Journal of Cloud Computing:
Advances, Systems and Applications

RESEARCH Open Access

ReactiveFnJ: A choreographed model for Fork-Join Workflow in Serverless Computing

Urmil Bharti[1*], Anita Goel[2] and S. C. Gupta[3]

Abstract

Function-as-a-Service (FaaS) is an event-based reactive programming model where functions run in ephemeral stateless containers for short duration. For building complex serverless applications, function composition is crucial to coordinate and synchronize the workflow of an application. Some serverless orchestration systems exist, but they are in their primitive state and do not provide inherent support for non-trivial workflows like, Fork-Join. To address this gap, we propose a fully serverless and scalable design model *ReactiveFnJ* for Fork-Join workflow. The intent of this work is to illustrate a design which is completely choreographed, reactive, asynchronous, and represents a dynamic composition model for serverless applications based on Fork-Join workflow. Our design uses two innovative patterns, namely, Relay Composition and Master-Worker Composition to solve execution time-out challenges. As a Proof-of-Concept (PoC), the prototypical implementation of Split-Sort-Merge use case, based on Fork-Join workflow is discussed and evaluated. The *ReactiveFnJ* handles embarrassingly parallel computations, and its design does not depend on any external orchestration services, messaging services, and queue services. *ReactiveFnJ* facilitates in designing fully automated pipelines for distributed data processing systems, satisfying the Serverless Trilemma in true essence. A file of any size can be processed using our effective and extensible design without facing execution time-out challenges. The proposed model is generic and can be applied to a wide range of serverless applications that are based on the Fork-Join workflow pattern. It fosters the choreographed serverless composition for complex workflows. The proposed design model is useful for software engineers and developers in industry and commercial organizations, total solution vendors and academic researchers.

Keywords Serverless computing, FaaS, Event-driven function composition, Choreography, Parallel computing, Distributed computing, Fork and Join, Orchestration

Introduction

Serverless computing is an emerging paradigm and is gaining popularity in the cloud owing to its simplicity, billing model, and inherent elasticity. This cloud computing execution model greatly simplifies the usage of cloud resources and suits well to highly scalable, event-driven applications in the cloud. Serverless architecture is especially effective at supporting modern applications with unpredictable scale and user demand [1].

The Function as a Service (FaaS) programming model of serverless allows programmers to develop cloud applications as individual functions that can run and scale independently. This model is event-driven since functions are activated in reaction to specific cloud events like, a state change in an object store, receipt of a message, a file upload, or insertion of a record in database. Though FaaS looks like a promising option for deploying cloud applications, it has few limitations also. Most notably, FaaS functions are stateless, short-lived, and

*Correspondence:
Urmil Bharti
urmil.bharti@rajguru.du.ac.in
[1] Department of Computer Science, Shaheed Rajguru College of Applied Sciences for Women, University of Delhi, Delhi, India
[2] Department of Computer Science, Dyal Singh College, University of Delhi, Delhi, India
[3] Department of Computer Science, Indian Institute of Technology, Delhi, India

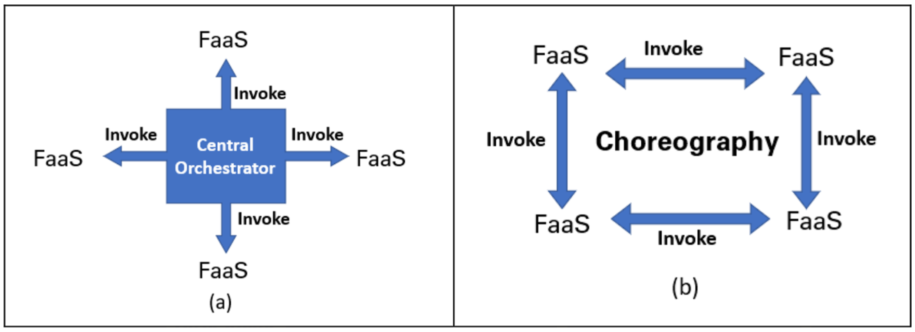

Fig. 1 Function Composition Approaches (**a**) Orchestration (**b**) Choreography

cannot communicate directly with each other [2, 3]. Thus, executing complex, burst-parallel jobs pose a significant challenge for serverless execution frameworks [4]. The composition of workflows in such jobs require extensive fine-grained communication and synchronization between independent functions that is challenging to implement in a serverless framework [5]. These FaaS challenges force developers to resort to alternate ways to establish function communication like publish-subscribe, passing data over some slow and expensive storage medium, or serverless orchestration services, but all these alternatives yield too high latency and cost [6].

At a high level, there are two approaches for function composition in a serverless application: orchestration and choreography [7], as shown in Fig. 1. In orchestration, a controller module orchestrates and controls the interaction between serverless functions. The controller module governs the flow according to the needs of the business logic. The choreography model is an event-driven paradigm in which every function works autonomously as a loosely coupled service. The functions work in a pipeline based on the triggered events. Each function performs its task, and its completion triggers the next function/s down the pipeline. In an event-driven architecture where each component plays a more architecturally aware role, the choreography model is used in the design of workflows instead of an orchestration model [8, 9].

The Fork-Join model is a programming method that exploits parallelism in applications based on inherent *divide and conquer* algorithms [10]. This execution model has already been successfully used for building parallel systems where an incoming task splits into subtasks that are processed by a set of parallel servers. The implementation of the Fork-Join model becomes more practical in serverless computing as these platforms are inherently scalable and do not need resource provisioning in advance. Since horizontal scaling in serverless is entirely automatic, elastic, and managed by its provider, dynamic parallel processing in Fork-Join can best exploit these characteristics.

Currently, Fork-Join workflow cannot be composed using any of the available serverless orchestration services [8] like Amazon Step Functions (ASF) (December 2016),[1] Azure Durable Functions (June 2017),[2] and IBM Composer (October 2017).[3] These services lack the ability to dynamically launch functions in parallel. Though ASF, the most mature and performant project [4], supports the *Parallel* state type to execute tasks in parallel. However, the application developers must list all the tasks to execute in parallel in an array in the state machine, thereby restricting the flexibility of the concurrency level [11].

In our research, we present ReactiveFnJ, an algorithm-based serverless design model for Fork-Join workflow. This serverless design is fully choreographed, vendor-neutral, and platform-independent serverless design model. The design solution uses innovative recursive and reactive design patterns for Fork-Join workflow. Our design is purely event-driven, trigger-based and satisfies all the three Serverless Trilemma (ST) constraints [12], and can handle hard execution time limits imposed by a serverless provider. The ReactiveFnJ handles embarrassingly parallel computations, but it does not depend on any external orchestration services, messaging services, and queue services. This model is generic, and several other use cases can be implemented by substituting specialized FaaS functions in our Proof-of-Concept (PoC) implementation. ReactiveFnJ model is the first work that exhibits all the characteristics mentioned above to the very best of our knowledge.

This manuscript is organized into several sections. Section 2 presents the technical background, motivation behind the proposed *ReactiveFnJ* serverless design model and summarizes our contributions. The detailed design

[1] https://aws.amazon.com/step-functions/
[2] https://docs.microsoft.com/en-us/azure/azure-functions/durable/
[3] https://cloud.ibm.com/functions/

and algorithms related to our serverless Fork-Process-Join pipeline are described in Section 3. Section 4 gives a detailed description of the implementation of our design using the AWS services. Section 5 shows the evaluation insights of the proposed models, followed by a discussion and lessons in section 6. The related works about the serverless applications implemented for parallel processing are described in Section 7. The concluding remarks and scope for future works are delineated in Section 8.

Background, motivation, and contribution

This section discusses current serverless challenges, the motivation behind this work, and the significant contributions of this research.

Serverless and its challenges

The serverless model was originally designed to execute event-driven, stateless functions in response to user actions or changes in the storage tier, e.g., uploading a photo to Amazon Simple Storage Service (S3), and inserting/updating a record in DynamoDB [13]. Some recent works have shown that the large-scale parallelism and auto-scaling features provided by serverless platforms make them well-suited for burst-parallel fine-grained tasks and parallel computation workflows [14]. In essence, the FaaS model is apt for embarrassingly parallel computing use cases such as linear algebra [15], optimization algorithms [16], data analytics [17], and real-time machine learning classifications [18].

Building a complex serverless application with numerous short-lived, concurrent functions requires new design guidelines. Beyond simple examples, serverless applications need to be designed as a composition of functions. In most cases, a serverless workflow composition needs an orchestration service that provides a coordination mechanism between FaaS functions [19]. These coordination services automatically trigger the execution of each function in the workflow and synchronize their behaviors and states. FaaS orchestration services such as AWS Step Functions or IBM Composer offer limited capabilities to coordinate serverless functions [20]. For instance, even in the AWS Step function, there is no provision for multiple functions to synchronize in parallel when the number of parallel instances is dynamic. So, in place of orchestration services, developers use some indirect ways to synchronize the dynamic parallel execution of functions via notification services, queue services, and in-memory data store/cache services [14] but each has its own limitations.

Few researchers proposed solutions for handling embarrassingly parallel computations in serverless. PyWren [21] uses its own ad-hoc external orchestration service, and ExCamera [22] relies on an external server to synchronize the parallel executions. These solutions add significant latency and cost as provisioning and configuration of external servers is required. Nevertheless, all the systems implemented so far for parallel computing do not comply with the four requirements claimed by Amazon for a serverless application: (i) No server management, (ii) Flexible scaling, (iii) Pay for value, and (iv) Automated high availability [8].

Barcelona-Pons. *et al.* argued that serverless functions follow a trigger-based model, so a FaaS composition should also be trigger-based [8]. This means that the termination of one or many functions should trigger the next stage (function) using asynchronous events in a workflow. From this perspective, any serverless composition achieving dynamic parallelism should also be trigger-based. Therefore, we emphasize that it is of utmost desire to build a serverless application as a complete reactive system of FaaS function compositions.

Any function composition is referred as ST-safe if it satisfies three main principles of Serverless Trilemma stated by Baldini et al. [12]: (1) Substitution - Each composition should behave like a function and could be substituted in any other pipeline, (2) Black-box - Each component of the workflow should be a black-box and abstracts from rest of the system i.e. implementation details of functions remain hidden from others, and (3) No Double billing - FaaS is a pay-per-use model, i.e., fine-grained resource measurement based on usage and there should not be double-billing of cloud function.

Fork-Join Model in serverless

The generic Fork-Join model of parallel processing splits a compute-intensive task into smaller sub-tasks to process them parallelly using the available CPU cores [23]. Therefore, the Fork-Join design model renders execution speed-up by running forked tasks in parallel and combining their results. This model can be best utilized in serverless implementation for use cases like, sorting [24], searching [25], matrix multiplication [26], string matching, MapReduce patterns like counting and summing, collating, and filtering [27]. In all the mentioned cases where the volume of data fluctuates, and resource requirements cannot be anticipated, the pay-per-use model of serverless is best used.

The above use cases can be best implemented using the generic Fork-Join workflow of parallel processing in serverless but designing this scalable workflow efficiently is a challenge in serverless frameworks. Currently, function composition for Fork-Join is not directly supported by the existing serverless orchestration services [8]. The available orchestration services are not designed for managing parallel Fork-Join workflows in a scalable and efficient way.

Motivation

Execution time limit is a major constraint of FaaS and hinders its implementation for applications where running time might go beyond the set time limits. We experimented on a serverless compute platform, AWS Lambda, provided as a part of Amazon Web Services. We found that a Single FaaS[4] function can only sort a file of size 560 KB in a default environment setting. If this serverless function gets an input file of size more than 560 KB, it ceases to complete due to execution time out.

It is just an example, but there are many compute-intensive scientific and business applications where constituent functions may time-out before their completion and hence, applications are unable to harness the power of serverless computing. There is a class of inherently parallel applications in any domain, where the initial task can be split into a large number of independent sub-tasks (Fork), and then each sub-task can be implemented as autonomous functions in serverless. This design brings up two challenges in serverless: (i) to provide some coordination mechanism to run in parallel a multitude of functions derived from a single task and (ii) to devise a synchronization mechanism to join the results of all split tasks and to prepare the aggregated output (Join).

One can attempt to use an existing orchestration service, but it may not be a viable solution as they do not support the execution of tasks in parallel when number of tasks are dynamically determined at runtime. As an example, Amazon Step Functions support the *Parallel* state type to execute tasks in parallel, but application developers have to provide all the tasks to execute in parallel in an array construct of the state machine [5]. This restricts the flexibility of concurrency level and hence, its usage in scenarios, where launching of functions in parallel is dynamic in nature.

The main downsides of currently available orchestration services are (i) Billing is based on the number of transitions happening during workflow execution (ii) Latency issues when working complex workflows (iii) Non adherence to serverless trilemma (violation of substitution and double billing principles) (iv) extra efforts are required to build a workflow in the orchestration services.

To handle these problems, we aim to design and implement a function composition mechanism for Fork-Join workflow which can be used into a broad spectrum of applications.

[4] Throughout this paper, we have referred to “Single FaaS” as a single serverless function instance that performs an operation without being time-out.

Contributions

The main contribution of this work is an algorithm-based design for serverless Fork-Join workflow. We have proposed a trigger-based serverless design model namely, *ReactiveFnJ,* for Fork-Join workflow. This design model can be utilized in compute intensive and burst-parallel applications. Our proposed model employs innovative design patterns that can process a file of any size without being time-out. In *ReactiveFnJ*, multiple component synchronization and coordination is crafted by *Relay Composition* and *Master-Worker Composition* design patterns thereby making it a choreographed and pure event-driven system. *ReactiveFnJ* is an asynchronous dynamic serverless composition design model that fully exploits the scalability, availability, and built-in fault tolerance of serverless infrastructure.

To prove the feasibility and viability of *ReactiveFnJ* design, we build a prototypical implementation to sort a large input file as a PoC. We call this sorting requirement as Split-Sort-Merge (SSM) use case throughout this research article. In SSM, the main aim is to sort a data file of any size (theoretically) where records are of variable length. There are several approaches to sort a large data file but the main challenge here is to develop an approach for a serverless architecture, where serverless functions are stateless and have a constrained execution environment. There exist multiple traditional serverful deployment frameworks for SSM, like, Map-Reduce and Apache Spark however, these frameworks suffer from cluster management, load balancing and task fairness issues [28]. Thus, developing/migrating these applications to serverless platforms illustrates unique opportunities.

After the successful implementation of SSM, it can be claimed that applications designed using *ReactiveFnJ* will not have dependency on any external orchestration service, will not time out and will be free from vendor lock-in problems.

The key contributions of this paper are-

- A pure event-driven choreographed design for Fork-Join workloads in serverless deployment.
- State-of-the-art ST-safe function composition model.
- Uses asynchronous push-based design exploiting recursive and parallel calling of functions.
- Self-driven function composition mechanism not relying on external orchestration services.
- Algorithm-level solution to handle execution time limitation imposed by serverless providers.

Our approach shall be a tipping point where a single machine/container is not big enough to perform a big computation and a serverless function fails for the same reason. Using our proposed novel algorithmic design approach, it shall be feasible to execute burst-parallel, compute intensive applications having large data-at-scale by leveraging the scalability benefit of FaaS.

ReactiveFnJ: proposed design

This section presents the design proposal for *ReactiveFnJ* in detail. Our design is inspired by external merge sort algorithm. The external merge sort is used when the data to be sorted do not fit into the main memory of a computing device. In this scenario, the data resides in an external memory (generally a hard drive). The external merge sort has two phases i.e., *Sort* and *Merge* [29]. In the sorting phase, a small chunk of data that can easily fit in main memory is read from external memory, sorted in main memory, and then written to a temporary file. This creates multiple sorted sub-files. In the merge phase, all sub-files are combined into a single sorted file.

Our design approach also aims to sort a large data file that cannot be handled by resources allocated to a single FaaS function. However, our approach differs from the external merge sort because we divide a large data file into small sub-files, and use the autoscaling feature of serverless environment to sort and merge the sub-files in parallel.

ReactiveFnJ design has three main components i.e., 1) *Fork*, 2) *Process* and 3) *Join,* as shown in Fig. 2. Using the well-established divide and conquer principle, the Fork component divides the main task into smaller subtasks for parallel processing. Once the main task is subdivided, each subtask is processed independently by *Process* component and intermediate results are produced. Responsibility of the *Join* module is to combine these intermediate results to produce the result.

The *Fork* component

The *Fork* is the first component of *ReactiveFnJ.* It has been designed to create small files from a data file of theoretically any size having records of variable length. To regulate size of the small files, we define a configurable parameter, namely, *MaxSplitSize (MSS).* The *MaxSplitSize* conveys the number of bytes that can be read and written by an instance of function responsible for forking. The value of *MSS*, may be set depending on the settings of the serverless execution environment.

Challenge in design

In a conventional way, data records of the input file can be read to create small size files referred as split files in this paper. Maximum number of records in a split file can be passed as an input parameter. This design for reading and splitting a file will eventually fail in case the input data file is too big. In other words, reading and creating

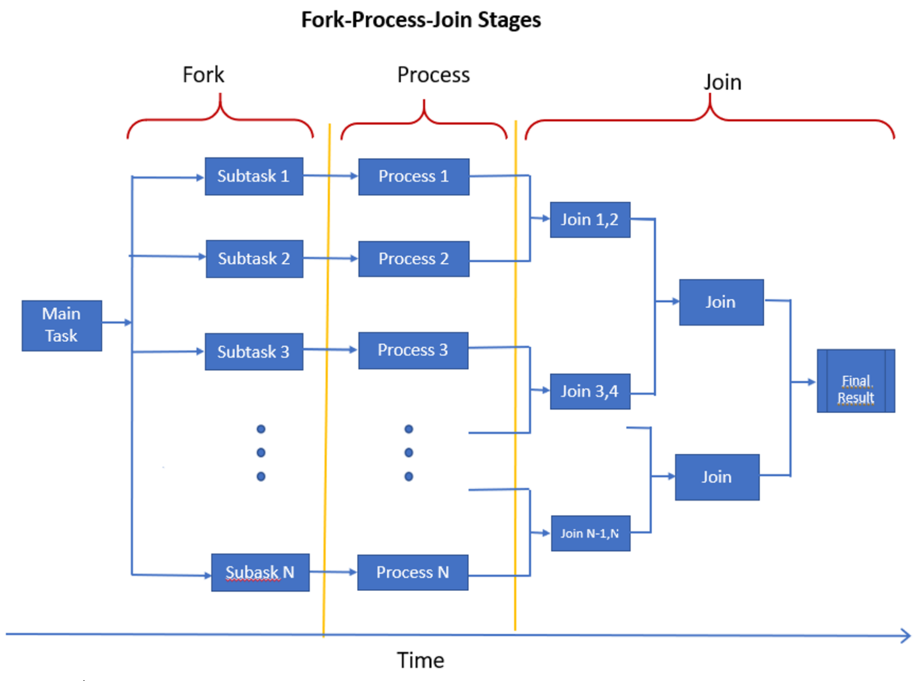

Fig. 2 Fork-Process-Join pipeline

smaller files will continue till the function itself does not surpass the memory usage and execution time-out limits imposed by service providers. This conventional design challenge for serverless is alleviated by a *Relay Composition* pattern *for Fork* as described below.

Relay composition for *Fork*

In this design, a big file is read, and split files are created by initiating a recursive *Relay Composition* pattern. This pattern can handle an input file of any size, theoretically, without being execution time-out. To make this composition more efficient, data is read and written in byte chunks of almost fixed number of bytes. The *MaxSplitSize* indicates maximum byte chunk that can be read/written by a single function without execution time-out. MSS is an estimated value and can be calculated empirically to know the maximum number of bytes that a *Single FaaS* function can read/write.

The number of split files N can be calculated as follows:

$$N = \lceil \left(\frac{InputFileSize}{MSS} \right) \rceil$$

For $1, 2 \ldots, N$ there are $F_1, F_2, \ldots, F_N$ split files where $|F_i| \leq MSS, \forall F_i;\ i \in Z^+$

The *Relay Composition* starts reading the first byte chunk and writes it to a new file and triggers the next instance asynchronously. New instance starts reading from the byte position in the file where the previous instance had stopped. The last record in a byte chunk may be incomplete due to variable length records and fixed *MSS* value. The composition design takes care of this case by discarding the partial read record and adjusting *StartByteLocation* parameter value as given in Algorithm 1. *StartByteLocation* works as a relay baton and is used to pass the next read position of the input file to the subsequent instance. Hence, the *StartByteLocation* is being relayed in every successive recursive instance till the end of the file is reached. This recursive style in *Relay Composition* where *StartByteLocation* being passed in successive calls as shown in Fig. 3. Thus, a file of any size, having variable length records can be forked in serverless infrastructures without being time-out.

Algorithm 1: Read and Split file of any size

Input: InputFilePath, Byte Chunk Size , OutputFilePath

Output: Split Files

```
1.  # Reads and splits input file into multiple files of specified size
2.  function splitFile(FilePath inputFile, Number numBytesToRead, FilePath outputFile)
3.    open inputFile in read mode,  open outputFile in write mode
4.    inputFileSize = get number of bytes in inputFile
5.    bytesTillNewLine = seek newLine/EOF after numBytesToRead bytes from startPosition
6.    while (startPosition + numBytesToRead + bytesTillNewLine < inputFileSize )
7.      bytesData = readChunk from startPosition till (startPosition +
8.                                   numBytesToRead + bytesTillNewLine) position
9.      write bytesData to outputFile, close outputFile,open new outputFile
10.     startPosition = (startPosition + numBytesToRead + bytesTillNewLine)
11.     if (startPosition + numBytesToRead + bytesTillNewLine < inputFileSize ) then
12.       bytesTillNewLine = seek newLine/EOF after numBytesToRead bytes from
    startPosition
13.     else
14.       #remaining file size is less than the bytes to read
15.       bytesData = readChunk from startPosition till file end
16.       write bytesData to outputFile, close outputFile
17.     end if
18.   end while
19. end
```

Each recursive instance creates a new split file that can be processed by the Process component in the design pipeline.

The design of the Fork component is for use in the use cases where same operation is to be performed on different data splits parallelly (data parallelism), as there is no

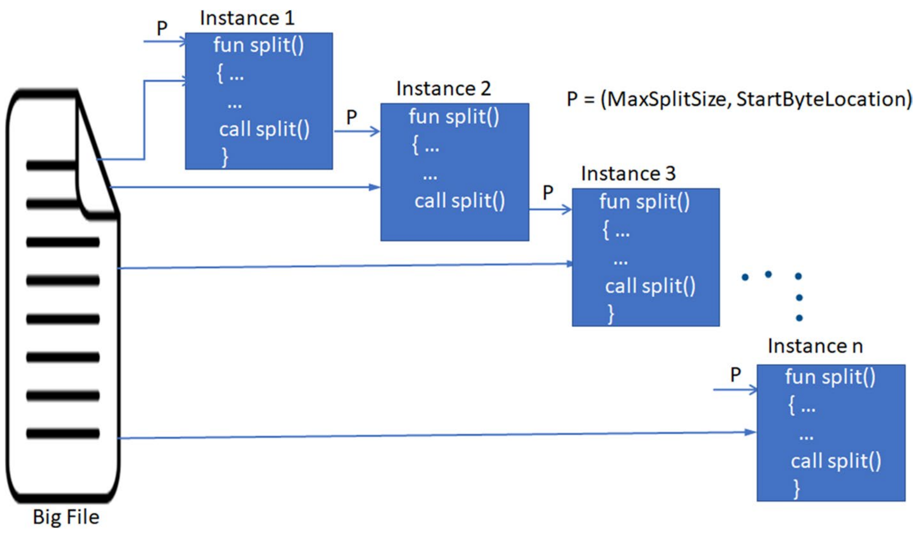

Fig. 3 *Relay Composition* used in spitting a big file

dependency among parallel tasks. Also, the design is only dependent on the value of *MaxSplitSize* that has to be calculated for the serverless environment configuration. It may be noted that the Fork component is suited for the forking of tasks that have no limitations on the minimum split size for processing.

The *Process* component

The *Process* is the second component of *ReactiveFnJ.* This component is designed to perform computation on all the split files carved in the previous step. All the split files are processed in parallel by independent functions. In essence, the design of *Process* component harnesses the power of scalability offered by a serverless infrastructure. Hence, parallel processing of sub-parts of initial data file helps in reducing the overall execution time in serverless. This component is the realization of a serverless idea of dynamically creating a compute cluster on demand without any overhead of cluster management. It can implement use cases where computation is data independent and therefore, can be executed in parallel. Some exemplary use cases include eCommerce, clickstream analytics, contact centre, legacy app modernization, and DevOps functions. In Algorithm 2, in-memory sort on a split file has been demonstrated but it could be any computation required to run in parallel.

```
Algorithm 2: In-memory sort of an Individual file
Input: InputFilePath, OutputFilePath
Output: Sorted OutputFile
1.  #Sorts the input file in ascending order
2.  function sortFile(FilePath inputFile, FilePath outputFile)
3.  Open inputFile in read mode
4.  For each Row in inputFile do
5.      Add row to dataList
6.  For End
7.  Sort data in dataList in ascending order
8.  Open outputFile in write mode
9.  For each datarow in dataList do
10.     Write datarow to outputFile
11. For End
12. Delete inputFile
13. End
```

The *Process* component delete the input split file (unsorted) at the end as its sorted copy is available for *Join* component.

The *Join* component

The *Join* is the final component of *ReactiveFnJ.* It is the core component of this design model. The results of the *Process* component are combined in the *Join* component to converge the result. The main job of this component is to join the processed split files in parallel. As per the design, in the first iteration, all available split files will be paired first and then joined to create a single file. The join component keeps on iterating this till a single file is left as shown in Fig. 2. Design of *Join* component is extremely efficient as number of iterations are growing in a binary logarithmic fashion as shown in Fig. 4 below. The number of iterations i required to join N number of files can be determined using our formula $2^{i-1} < N \leq 2^{i}$ where $i \in Z^{+}$.

The design of *Join* component was the biggest challenge we faced. Being a serverless component, the *Join* design should adhere to the serverless principles: (i) complete decentralized scheduling, (ii) reactive inter-module communication, (iii) pure asynchronous push-based communication approach. To claim a component to be truly serverless, it should not use any external orchestration service to synchronize its execution workflows along with the principles mentioned above.

Challenge in design

The conventional way of joining N processed files in parallel is not a viable design in a serverless environment. In the traditional design, all processed files are joined in parallel in pairs. Each pair of files initiates the event-driven merging process, and a sorted file is created. Newly built joined files are again ready for join and this process will continue till a single file is left as shown in Fig. 5.

This conventional method works well when the size of a joined file can be handled under the limits of a serverless environment. But it shall eventually fail when file size starts growing as the joining process exceeds the serverless execution time limit. This design challenge is resolved by a reactive *Master-Worker Composition* (MWC) pattern as described below.

Master-Worker Composition for *Join*

MWC is an innovative design solution to handle the exponential increase in file size, the main cause of execution time-out, during the joining process. The Master is responsible for initiating joining of two files and *Worker* takes the responsibility of merging two sorted files. The *Worker* can handle files of any size. So the main highlight of this design is that it can join any number of files without any constraint on file size. The detailed working of this composition is described below.

In MWC, a parallel recursive join process is formulated which is devoid of time-out constraint. In this composition, we have two important modules, first is MasterReactiveMerge (MRM) and other one is *WorkerRecursiveMerge* (WRM). The MRM is responsible for initiating a join between a pair of split files. As MRM is reactive so whenever a pair of files is available for join, it is invoked automatically. Event-driven characteristic

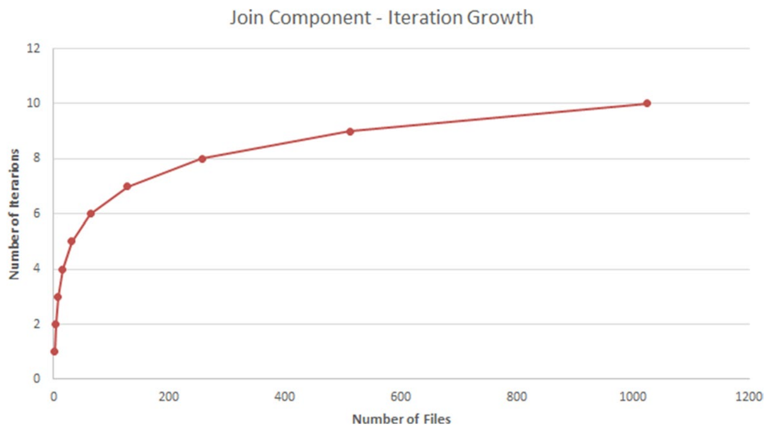

Fig. 4 *Join* Component – Binary Logarithmic Growth Curve

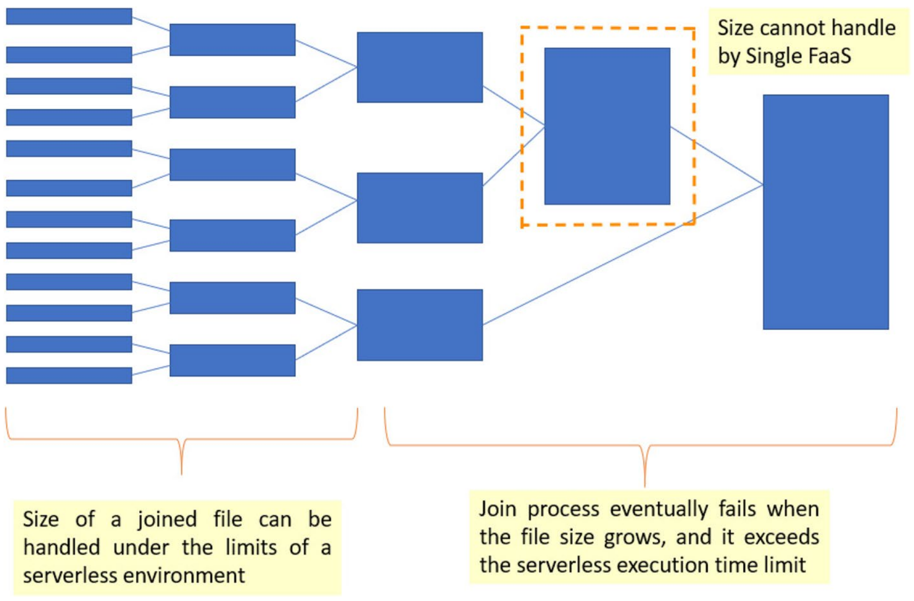
Size cannot handle by Single FaaS
Size of a joined file can be handled under the limits of a serverless environment
Join process eventually fails when the file size grows, and it exceeds the serverless execution time limit

Fig. 5 Conventional method to join N files in Parallel

of serverless architecture helps MRM to run in parallel for each pair of files. Every instance of the MRM module calls a WRM module, responsible for joining two files. First invocation of this module keeps on joining the two files till the size of the joined file reaches a limit called *JoinSafeLimit* which is passed as a parameter to this module. This parameter regulates the data size that can be joined by a single serverless function instance. The *JoinSafeLimit* is similar to the *MaxSplitSize* and can be calculated empirically. Files joined by WRM could be of any size so its multiple sequential instances may be required to join files of big size. As the file size increases, the number of instances increases proportionally. One instance of WRM joins two files upto *JoinSafeLimit* and before getting time-out calls next WRM instance with appropriate parameters like read positions for both the files. Two files are read and joined to build a single joined file by one or more WRM instances, where the resultant file is appended by each instance.

The key design features of *Join* component are as follows:

- Join process is automatic where the number of files and their size are not known in advance.
- No central controller for joining the files. It uses decentralized scheduling.

- No additional messaging-service or shared-memory for inter-module communication.
- Event-driven architecture and makes use of trigger-based communication.
- Supports dynamic sequential and parallel composition of functions and synchronizes burst parallelism.
- Ensures that executions are not double-billed as design is based on pure asynchronous push-based communication approach.
- No use of external rendezvous server, ad-hoc orchestrator service and current serverless orchestration systems for task scheduling and synchronization.
- Based on reactive programming model that is highly recommended for serverless.

Implementation

The *ReactiveFnJ* is implemented using cloud computing services provided by Amazon Web Services. The serverless functions are developed using AWS Lambda. It is a mature FaaS platform, so we opted it for our experimental implementation.

To validate the design of *ReactiveFnJ*, we implement Split-Sort-Merge (SSM) case study. In this study, the main goal is to sort a data file of any size having variable length records. SSM is a perfect case study based on the generic Fork-Process-Join model of parallel processing. In SSM, *Fork* component divide input file into split files, *Process* component does in-memory sorting of individual split files (in parallel), and *Join* component merges the sorted files (in parallel). We have typically chosen this use case to sort a very big text file that conventionally cannot be sorted using the resources limitations of a single serverless function container. There are many solutions available for handling these scenarios like Cloud IaaS, Hadoop Map Reduce, on-premises cluster etc. But we understand, serverless Function as a Service (FaaS) is the most attractive and methodical option because of its simplicity, billing flexibility and inherent elasticity. Study and implementation of this prototype aims to leverage existing event-based technology of serverless architectures to enable triggered compositions in complex workflows.

General overview of SSM

In the implementation of SSM, the assorted AWS services that have been used are as follows-

(1) *Amazon S3* to store input file, split files, intermediate sorted files and final sorted file,
(2) *AWS Lambda* for execution of split, sort and merge sub-tasks of workflow,
(3) *Amazon S3 Event Notifications* to send event messages for coordination of Lambda functions,
(4) *AWS Identity and Access Management (IAM)* to manage/access AWS resources [30], and
(5) *AWS CloudWatch* to monitor/observe logs, metrics and events for Fork-Join pipeline.

We implemented the AWS Lambda functions using Python 3.8 because it offers library support to manage critical operations like, creation, deletion of S3 bucket folders at runtime, setting/retrieving input/output file path, and read/write CSV format files. Additionally, we have used few important Python modules-*Botocore, Boto3, S3FS, JSON, CSV, and OS* in our implementation.

Source code of all the components developed for Split-Sort-Merge pipeline implementation is available at https://github.com/anitagoel/ReactiveFnJ.

Specifics of AWS Lambda functions

In this section, AWS Lambda functions of the SSM pipeline have been discussed. For the implementation of the SSM, four functions have been developed – (i) BL_ReadAndSplit (λBR&S) – Lambda function responsible for creating split files, (ii) Sort (λS) - Lambda function responsible for sorting a split file, (iii) MasterReactiveMerge (λMRM) – Lambda function responsible for initiating join between two files and, (iv) WorkerRecursiveMerge (λWRM) - Lambda function responsible for merging two sorted files.

Further, for implementation of the SSM, five Amazon S3 folders have been used – (i) input – stores input file, (ii) to_*process* – for storing file splits, (iii) *to_join* – to store sorted split files, intermediate merged files and final merged file, (iv) *to_merge* – for temporarily storing pairs of files undergoing merge, and (v) *archive* – to archive input file after successful processing.

The interaction among all Lambda functions of the SSM pipeline is based on S3 triggers set on *PUT* event of the above-mentioned folders having ".csv" filter.

Figure 6 shows the complete deployment diagram of SSM.

BL_ReadAndSplit

A large file can be read recursively to overcome the restriction of execution time-out for an AWS Lambda function. The recursive Lambda function, λBR&S, reads a byte chunk and writes it to a new file. It is invoked when an input file is uploaded to the folder "/input" of S3 bucket. Data byte chunks are read from the input file and split files are created in a new folder "to_process" (created at runtime). A configurable

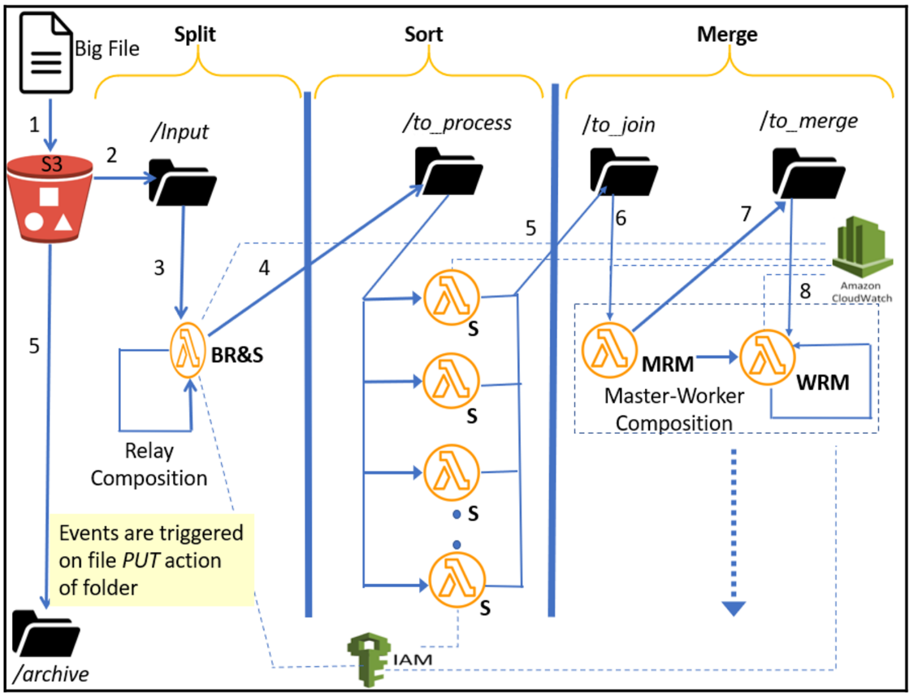

Fig. 6 The deployment diagram of SSM

parameter *ByteChunkSize* is used by this function that determines the size of split files. Just before getting time-out, this function asynchronously invokes itself by passing the new read byte position in the input file. First instance of λBR&S creates the first split file and invokes the next instance to create the next split file and so on. This process continues till the end of the file. This recursive implementation works perfectly for any size of input file. Invoking a function asynchronously using *lambda.invoke* and setting *Invocation-type* flag as "Event", place invoke requests in Lambda service queue for processing the requests as they arrive. Keeping *lambda.invoke* as the last statement in this function's code eliminates double billing as it would not force the Lambda to wait for each invocation to finish. Thus, recursive invocations of this function where each invocation communicates and coordinates with each other, attains "No Double Billing" condition for the serverless function composition mechanism of serverless trilemma.

After the successful completion of this function, the input file is deleted from its folder and is moved to a "\input_archive" folder in S3 bucket. Naming convention of a split file carries two important attributes of information i.e., (i) Total number of Splits and (ii) File Split Number as shown in Fig. 7. Split file name and achieve folder helps in detecting and debugging failures, if any, via AWS CloudWatch log.

Sort

Sort is the AWS Lambda function implemented for sorting a split file. All split files invoke λS on their creation and resultant sorted files are stored in the "/to_join" folder of S3 bucket. Hence, λS instances will run in parallel. The coordination between *BL_ReadAndSplit* and *Sort* is done through S3 event notification. In the timeline, one *ReadAndSplit* Lambda function reads a large file and creates multiple small files. Small files will be created in sequence one after another and each of these files will also get sorted and stored in a new S3 bucket folder as shown in Fig. 6. Choreography of all these tasks is fully event-driven.

Parallel Reactive Merge

Parallel Reactive Merge is a serverless solution to initiate merging of two sorted files of any size. This implementation solves the challenge of merging files when the size of sorted files starts growing and the execution time-out limit reached before the merging process completes. So Parallel Reactive Merge overcomes the serverless execution time-out constraint. For this, two Lambda functions are implemented: (1) MasterReactiveMerge(λMRM) (2) WorkerRecursiveMerge(λWRM).

In this implementation, a parallel reactive merging process is devised that will complete the merging without being time-out. A Lambda function MasterReactiveMerge is invoked whenever a sorted file is dropped in

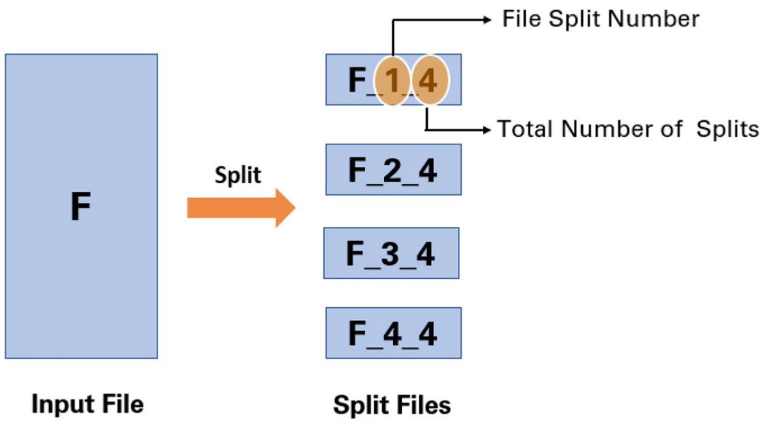

Fig. 7 Split File Naming convention

the *to_join* folder of S3 bucket. This Lambda function picks up another file available in *to_join* for merging, moves these two files to the *to_merge* folder and invokes the first instance of WRM. The WRM merges records from two sorted files and writes them to a new file. The WRM takes input parameters as: i) Working Directory, ii) File1, iii) File2 iv) Start Position in File1, v) Start Position in File2, vi) Output File, and vii) Byte Threshold Value. First invocation of WRM carries Start Position for both the input files as zero. Before getting time-out, WRM calls itself and the next instance starts appending records to the same file created by the first instance. Hence, λWRM, recursively calls itself till the single merged file is created as depicted in Fig. 8. Last instance of λWRM moves the merged file to the "/to_join" folder and deletes both the input files. One instance of λMRM followed by one or more instances of λWRM is initiated for every pair of files in "/to_join" and this parallel merging process continues till a single file is left in the "/to_join" folder.

Evaluation

In this section, we describe the quantitative assessment for our proposed design model. This will facilitate us to demonstrate the feasibility of our algorithmic approach in designing fully choregraphed Fork-Join workflow in a serverless architecture.

We find that the serverless design using *Relay Composition* pattern can be successfully used for long running computation. Our Fork component allows us to conclude that use cases where processing time might exceed the execution time limit can also make use of serverless

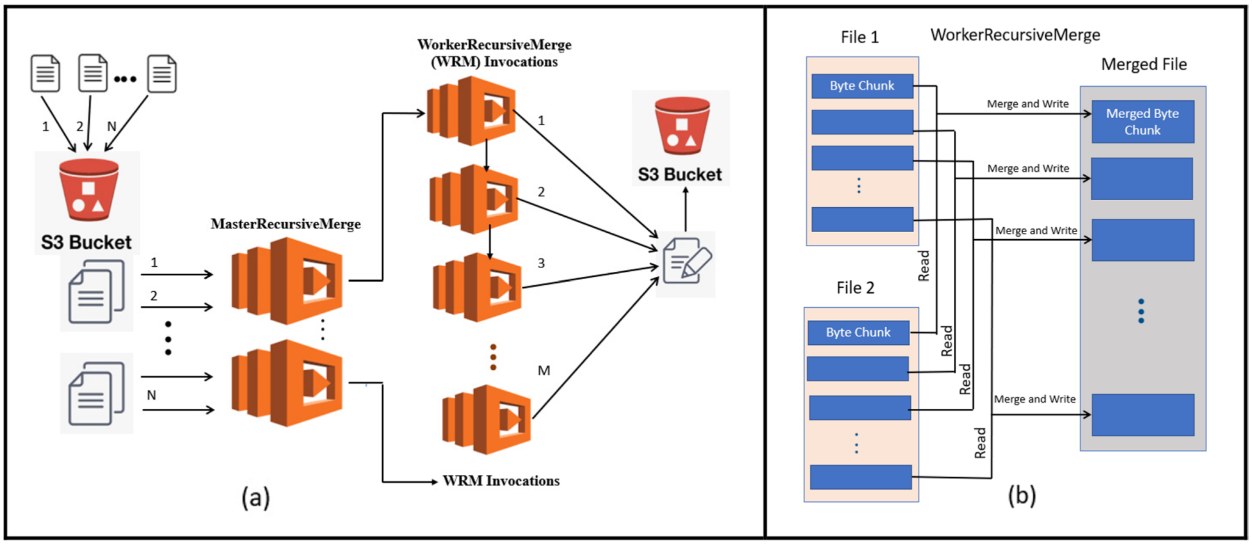

Fig. 8 **a** Parallel Reactive Merge **b** Worker Recursive Merge

Table 1 Experimental Results – Single Lambda v/s *Relay Composition* pattern

Input File Size (KB)	Design Used	Time Out	No. of Files	(No. of Files x *MaxSplitSize* (KB)) + Size of last split file (KB)
200	[a]SF	No	1	N/A
300	SF	No	1	N/A
600	**SF**	**Yes**	**1**	**N/A**
600	[b]RC	No	2	(1 × 450) + 150
700	RC	No	2	(1 × 450) + 250
800	RC	No	2	(1 × 450) + 350
1000	RC	No	3	(2 × 450) + 100
1200	RC	No	3	(2 × 450) + 300
1400	RC	No	4	(3 × 450) + 50
1600	RC	No	4	(3 × 450) + 250

[a] SF- Single FaaS

[b] RC- *Relay Composition*

technology. Results of our experiments are presented in Table 1 where *MaxSplitSize* is taken as 450 KB.

The execution of a complex, burst-parallel workflow can be addressed programmatically without using any external orchestration service. The algorithmic approach is a priori more powerful considering the availability of basic control flow instructions in any serverless runtime [17]. Our design model can handle burst-parallel workload by utilizing auto scalability of serverless along with algorithmic design. Successful implementation of Fork-Join workflow without using any external orchestration service proves this argument. Results of SSM pipeline based on *ReactiveFnJ* are given in Fig. 9.

We proved that it is possible to design a serverless system using pure event-driven architecture for complex workflows. The *ReactiveFnJ*, a pure event-driven system, using two novel serverless design patterns i.e., *Relay Composition* and *Master-Worker Composition*. These patterns are used in prototypical implementation of SSM pipeline that initiates when an input file gets uploaded in the S3 bucket. Subsequent steps like splitting, sorting, and merging are triggered automatically without any human arbitration. This shows that complex parallel systems can be designed in a pure event-driven architecture.

To summarize our results, the *ReactiveFnJ* is a choreographed design model for the development of distributed applications based on Fork-Join workflow with serverless architectures. This design can handle long running tasks by overcoming execution time-out constraint and is not dependent on any external orchestration service for its function composition.

As regards the cold start phenomenon during the execution of *Fork-Join* pipeline, mostly the initial containers will face cold start delay. The *Fork* and *Worker-RecusiveMerge* component of the pipeline uses the *Relay* pattern (one invocation initiates the next one), so all the containers except initial ones will have a warm start. In *Process* and *MasterReactiveMerge* components, initial function invocations will suffer cold start, but once their containers become available again, then rest of the invocations will have warm start. Hence, the *Relay* pattern and invocations of same functions again and again creates a pool of containers available for warm start in pipeline execution. So theoretically it can be inferred that initial containers will have cold start delay but it will not affect every invocation hence overall impact will not be high.[5]

Discussion and limitations

Our model for Fork-Join workflow can be best utilized in serverless environment as this technology provides auto-scalability, built-in fault tolerance, availability, and abstraction of underlying infrastructure. To make Fork-Join workflow available, we build *ReactiveFnJ*, a serverless composition model. The *ReactiveFnJ* uses an innovative design that is purely event-driven, reactive, ST-safe, choreographed composition model that conquers the hard time-limit forced by serverless environment. To achieve these characteristics, our model keeps two copies of data in the *Process* stage i.e., unsorted input split files and its corresponding sorted files. So, during *Process*, required storage capacity becomes twofold temporarily for a short duration of time. It starts declining as the *Process* component deletes unsorted input split file as soon as its corresponding sorted file is prepared. At the end of the Fork-Join workflow, only two files are available i.e., input file and output file. The model can be easily updated to keep only one final sorted file, if required. Our implementation uses S3 storage which is an AWS object storage service. No

[5] https://docs.aws.amazon.com/lambda/latest/operatorguide/execution-environments.html

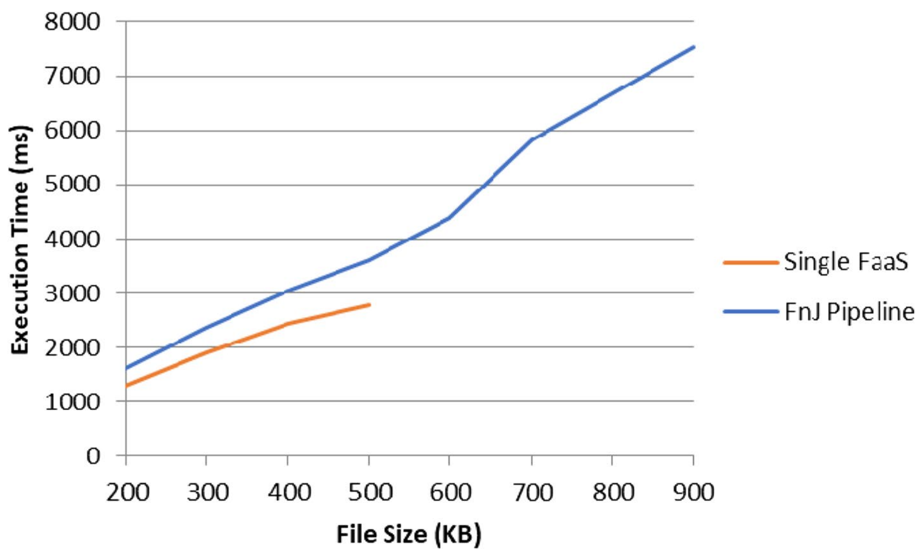

Fig. 9 Executions Results of Single Lambda vs FnJ Pipeline

additional infrastructure costs are involved in this storage option as it is managed by the provider and charged on per-use basis.

Our model is generic, and several other workflows/use cases can be implemented by substituting specialized FaaS functions in *ReactiveFnJ*. With the intent to develop other complex compute intensive and data parallel applications, patterns analogous to our innovative composition patterns i.e., *Relay Composition* and Master-*Worker Composition* should be discovered. We have articulated following general design guidelines that will help serverless system designers and architects to engineer their applications in an efficient and effective manner.

Identification of iterative patterns in a long-running task. An iterative pattern can be implemented as a recursive serverless function where each function asynchronously invokes the next instance by relaying the appropriate parameters till the task is completed. Recognition of computing patterns that are data independent and hence, can be executed in parallel. These patterns can be implemented as serverless functions runnable in parallel exploiting on-demand scale features of serverless infrastructure. For coordination between serverless functions *Master-Worker Composition* is applied. In this composition a master function is invoked automatically on an event and is responsible for invoking and dispatching parameters to Worker. The Worker recursively calls itself till the task is completed. In this composition Master is reactive in nature and Worker is recursive in its functioning.

The proposed design has some limitations too. Firstly, for problem domains, like ad-hoc and less common custom tasks that have limitations for the minimum possible split size, the proposed design may not be well-suited.

Secondly, our implementation of the proposed design uses Amazon Web Services like AWS Lambda and Amazon S3. So, *Fork-Join* workflow implementation primarily depends on cloud object storage. The co-ordination in workflow is handled either by the object storage event notifications or the asynchronous invocations via serverless functions. For this, Python modules like - *Boto3* and *S3FS* are used which are very specific to AWS environment. The implementation of our design in other serverless platforms will require the use of their environment specific Python libraries. Most of the leading serverless service providers have Python modules offering services similar to *Boto3* and *S3FS*. In IBM cloud,[6] Python support is provided through a fork of the boto3 library. In Microsoft Azure, the open-source Azure libraries for Python are available for using Azure resources.[7] The Python application *gsutil*, allows access to Google Cloud Storage[8] to do a wide range of bucket and object management tasks. In case similar Python modules are not available in a serverless environment, then the required services need to be developed to implement the proposed model.

[6] https://cloud.ibm.com/docs/cloud-object-storage?topic=cloud-object-storage-python

[7] https://learn.microsoft.com/en-us/azure/developer/python/sdk/azure-sdk-overview

[8] https://cloud.google.com/storage/docs/gsutil

Related Work

Composition of serverless functions to implement workflows is sparsely covered in the current scientific literature but immensely important in practice [3]. Workflow management systems like Apache Airflow,[9] Oozie [31] and Camunda [32] exist but they depend on a dedicated long-running stateful execution engine to handle the orchestration. So, the serverless frameworks must pave the way for complex function composition mechanisms to build responsive, compute intensive, and burst-parallel distributed serverless applications [5].

In some previous studies, researchers have explored and harnessed the power of FaaS in parallel processing applications. For massively parallel computations on serverless, systems like PyWren [21] and ExCamera [22], have used their own external ad-hoc orchestration services to synchronize the parallel execution of cloud functions. PyWren utilized a polling mechanism for the Amazon S3 bucket to consolidate its results. In ExCamera implementation, an additional external server is configured to synchronize the interactions of parallel running executions. Shankar et al., developed a serverless system Numpywren for highly parallel linear algebra algorithms. For their implementation, authors introduced a new domain-specific language "LAmbdaPACK" [15]. Zhang et al., proposed a new serverless framework "Kappa", that used a check-pointing mechanism to control function execution time-out [33].

On the academic front, the authors of [26], proposed a design approach that exploits data parallelism in serverless infrastructure for massively parallel computations. A running prototype is implemented in AWS Lambda for distributed matrix multiplication using the design approach. Their design uses a central controller module outside serverless environment. The authors [34] in thoroughly discussed different sequential workflow compositions in varied language runtime environments. They also compared chaining composition in AWS serverless platform and IBM Cloud Function. Their experimental results show that a pure serverless composition is efficient in terms of execution time in comparison to external orchestration service. Witte et al., [35] illustrated a cloud specific serverless implementation for seismic imaging application. They used orchestration service AWS Step Function visual workflow for their implementation and exploited the mathematical properties of imaging optimization problem.

In a recent study, different FaaS platforms are analysed and compared to run highly parallel computing jobs. This research answers an important question: do all existing serverless platforms suitable for parallel computations? Their results clearly indicate that AWS and IBM provides better automatic elasticity for parallelism [36]. In another research, authors experimentally analysed the current support for serverless workflows in FaaS platforms and uncovered important weaknesses. They introduced Abstract Function Choreography Language to overcome some of the existing gaps [7]. Durable Functions allow developers to implement advanced serverless workflows but can create an IOps bottleneck. To handle it, a novel architecture "Netherite" is developed for executing serverless workflows on an elastic cluster [37]. One of the recent research works presents "DIFFUSE", a decentralized and distributed platform. It enables function composition in serverless environments, but it relies on pluggable middleware support for conveying messages among the platform components [38].

Serverless popular function orchestration services are- AWS Step Functions, Azure Durable Functions, IBM Cloud Functions, and Google Cloud Functions. They render help to create services involving compositions of serverless functions. But currently they do not support Parallel Fork-Join workflows natively [8]. As per L´opez et al., [1] serverless applications must follow a trigger-based interaction mechanism. They expressed that the FaaS orchestration system should also be event-triggered. It means, in a complex business workflow, the termination of one FaaS should trigger the next function in the pipeline using asynchronous events.

Concurrency and parallel computation of serverless are well suited for requirements like sorting a huge data file. Efficient sorting of large data sets is an extensively discussed area as it is central to large businesses, banks, and institutions. Sorting counts for roughly one-fourth of the total computer cycles [39]. Usually, if a data file is small and can fit into the memory limits of a FaaS container, it can be easily programmed for sorting. But in case a file is huge then sorting by a single FaaS will time-out before its completion. Due to resource limitations of a serverless environment, processing of very large files is not supported directly. Recently, Amazon Elastic File System (EFS) volume mounted to handle such large files [40]. However, latency and additional cost are some challenges around this way-out.

After investigating previous and recent research works, a clear gap surfaced regarding the non-existence of reactive Fork-Join workflow runnable for serverless infrastructures. To the best of our knowledge, ours is the first work that presents the design and implementation of *ReactiveFnJ* - a pure reactive, choreographed, ST-safe, algorithmic solution and answer the intrinsic constraint of serverless environments. This makes the best use of the scalability and fault tolerance feature of serverless.

[9] https://airflow.apache.org/

ReactiveFnJ is judicious to be used for large-sized datasets where divide and conquer strategies can be applied.

Conclusions and future work

Current serverless technology provides its users with fine billing granularity and affordability to run arbitrary functions on-demand. We can reap great benefits of serverless when it comes to the runtime scalability of functions. Subtle barriers of this architecture are resource limit of functions, designing complex applications that require state and composability of long running functions.

This paper is an academic endeavour that designs a choreography model *ReactiveFnJ* for Fork-Join workflow in serverless architecture. This model is pure event-driven, ST-Safe, and makes best use of scalability and fault tolerance features of serverless environment. This research proves the viability of a serverless design for a complex burst-parallel workflow where every serverless challenge is resolved at algorithmic level rather than using any external orchestration service, shared memory, or messaging services for its task scheduling and synchronization. This solution is platform independent as a result it is free from vendor lock-in problem also. Our model will render help to serverless architects and developers in fabricating compositions based on Fork-Join workflows. We contemplate that each component of the Fork-Join pipeline can be substituted by a variety of serverless functions to generate diverse workflows. The proof-of-concept and evaluation results authenticate that *ReactiveFnJ* can deliver sufficient performance and suggests its adoption in serverless frameworks for large-scale distributed computing. As future work, we would like to develop specific modules and libraries for Fork-Join components in serverless for the production grade applications.

Abbreviations

FaaS	Function as a Service
ST	Serverless Trilemma
SSM	Split-Sort-Merge
MSS	MaxSplitSize
MWC	Master-Worker Composition
MRM	MasterReactiveMerge
WRM	WorkerRecursiveMerge
IAM	AWS Identity and Access Management
PoC	Proof-of-Concept
ASF	Amazon Step Functions;
λBR&S	Lambda for BL_ReadAndSplit
λS	Lambda for Sort
λMRM	Lambda for MasterReactiveMerge
λWRM	Lambda for WorkerRecursiveMerge

Acknowledgements
Not applicable.

Authors' contributions
All authors take part in the discussion of the work described in this paper. Urmil Bharti contributed to the model design and validation experiment of this work. She drafted the manuscript and coordinated the review task among authors. The author(s) read and approved the final manuscript.

Authors' information
Ms. Urmil Bharti has done B.Sc. Computer Science (University of Delhi, India), M.Sc. Computer Science (DAVV Indore, India) and M.Phil. Computer Science (MKU, India). She has over 15 years of teaching experience as Assistant Professor in constituent colleges of University of Delhi. Earlier she worked in IT industry for more than 10 years. Her last designation in industry was Senior Quality Analyst. She is currently doing her research in Cloud and Distributed Computing. Her key research area is cloud computing, serverless technology and software engineering. She has authored several national and international research publications.
Dr. Anita Goel is currently a Professor with the Department of Computer Science, Dyal Singh College, University of Delhi, India. She has a work experience of more than 30 years. She is also a visiting faculty to several universities in India. She has been a fellow of Computer Science with the Institute of Life Long Learning (ILLL), University of Delhi. She has guided several students for their doctoral studies and has travelled internationally to present research papers. She is a serving member of program committee of several international conferences. She has authored 21 books in computer science. She has several national and international research publications. Her research interests include cloud computing, microservices, serverless computing, software engineering, and technology-enhanced education (MOOC).
Dr SC Gupta is B.Tech (EE) from IIT Delhi and has worked at Computer Group at Tata Institute of Fundamental Research and NCSDCT (now C-DAC Mumbai), Till recently, he worked as Deputy Director General, Scientist-G and Head of Training at National Informatics Centre, New Delhi and was responsible for keeping its 3000 scientists/engineers up-to-date in various technologies. He has extensive experience in design and development of large Complex Software Systems. Currently he is a Visiting Faculty at Dept of Computer Science and Engineering, IIT Delhi. His research interests include Software Engineering, Database and Cloud Computing. He has been teaching Cloud Computing at IIT Delhi, which includes emerging disruptive technologies like SDN and SDS. He has guided many M.Tech. & PhD Research students in these technologies and has many publications in Software Engineering and Cloud Technology in National and International Conferences and Journals.

Funding
Not applicable.

Availability of data and materials
Not applicable.

Declarations

Competing interests
The authors declare that they have no competing interests.

Received: 26 September 2021 Accepted: 20 March 2023
Published online: 24 April 2023

References
1. Arjona A, López PG, Sampé J, Slominski A, Villard L (2021) Triggerflow: Trigger-based orchestration of serverless workflows. Futur Gener Comput Syst 124:215–229. https://doi.org/10.1016/j.future.2021.06.004
2. Hassan HB, Barakat SA, Sarhan QI (2021) Survey on serverless computing. J Cloud Comput 10:1–29
3. Leitner P, Wittern E, Spillner J, Hummer W (2019) A mixed-method empirical study of Function-as-a-Service software development in industrial practice. J Syst Softw 149:340–359
4. López PG, Sánchez-Artigas M, Par\'is G, Pons DB, Ollobarren ÁR, Pinto DA (2018) Comparison of FaaS orchestration systems. 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion). pp 148–153
5. Carver B, Zhang J, Wang A, Anwar A, Wu P, Cheng Y (2020) Wukong: A scalable and locality-enhanced framework for serverless parallel

computing. In Proceedings of the 11th ACM Symposium on Cloud Computing, pp. 1-15.
6. Pu Q, Venkataraman S, Stoica I (2019) Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In NSDI, vol. 19, pp. 193-206.
7. Ristov S, Pedratscher S, Fahringer T (2021) AFCL: An abstract function choreography language for serverless workflow specification. Futur Gener Comput Syst 114:368–382
8. Barcelona-Pons D, Garc\'\ia-López P, Ruiz Á, Gómez-Gómez A, Par\'\is G, Sánchez-Artigas M (2019) Faas orchestration of parallel workloads. Proceedings of the 5th International Workshop on Serverless Computing. pp 25–30
9. Leite LAF, Oliva GA, Nogueira GM, Gerosa MA, Kon F, Milojicic DS (2013) A systematic literature review of service choreography adaptation. Serv Oriented Comput Appl 7:199–216
10. Landset S, Khoshgoftaar TM, Richter AN, Hasanin T (2015) A survey of open source tools for machine learning with big data in the Hadoop ecosystem. J Big Data 2:1–36
11. Yu T, Liu Q, Du D, Xia Y, Zang B, Lu Z, Yang P, Qin C, Chen H (2020) Characterizing serverless platforms with serverlessbench. Proceedings of the 11th ACM Symposium on Cloud Computing. pp 30–44
12. Baldini I, Cheng P, Fink SJ, Mitchell N, Muthusamy V, Rabbah R, Suter P, Tardieu O (2017) The serverless trilemma: Function composition for serverless computing. Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software. pp 89–103. https://doi.org/10.1145/3133850.3133855
13. Kuhlenkamp J, Werner S, Tai S (2020) The ifs and buts of less is more: A serverless computing reality check. 2020 IEEE International Conference on Cloud Engineering (IC2E). pp 154–161
14. García-López P, Sánchez-Artigas M, Shillaker S, Pietzuch P, Breitgand D, Vernik G, Sutra P, Tarrant T, Juan-Ferrer A, París G (2022) Trade-Offs and Challenges of Serverless Data Analytics. In: Curry E, Auer S, Berre AJ, Metzger A, Perez MS, Zillner S (eds) Technologies and Applications for Big Data Value. Springer International Publishing, Cham, pp 41–61
15. Shankar V, Krauth K, Vodrahalli K, Pu Q, Recht B, Stoica I, Ragan-Kelley J, Jonas E, Venkataraman S (2020) Serverless linear algebra. Proceedings of the 11th ACM Symposium on Cloud Computing. pp 281–295
16. Dai D, Chen Y, Kimpe D, Ross RB (2018) Trigger-based incremental data processing with unified sync and async model. IEEE Trans Cloud Comput 9:372–385
17. Sampé J, Vernik G, Sánchez-Artigas M, Garc\'\ia-López P (2018) Serverless data analytics in the IBM cloud. Proceedings of the 19th International Middleware Conference Industry. pp 1–8
18. Christidis A, Davies R, Moschoyiannis S (2019) Serving machine learning workloads in resource constrained environments: A serverless deployment example. 2019 IEEE 12th Conference on Service-Oriented Computing and Applications (SOCA). pp 55–63
19. Hellerstein JM, Faleiro J, Gonzalez JE, Schleier-Smith J, Sreekanti V, Tumanov A, Wu C (2018) Serverless computing: One step forward, two steps back. arXiv preprint arXiv:1812.03651.
20. Barcelona-Pons D, Sánchez-Artigas M, Par\'\is G, Sutra P, Garc\'\ia-López P (2019) On the faas track: Building stateful distributed applications with serverless architectures. Proceedings of the 20th international middleware conference. pp 41–54
21. Jonas E, Pu Q, Venkataraman S, Stoica I, Recht B (2017) Occupy the cloud: Distributed computing for the 99\%. Proceedings of the 2017 symposium on cloud computing. pp 445–451
22. Fouladi S, Wahby RS, Shacklett B, Balasubramaniam KV, Zeng W, Bhalerao R, Sivaraman A, Porter G, Winstein K (2017) Encoding, Fast and Slow:$\{$Low-Latency$\}$ Video Processing Using Thousands of Tiny Threads. 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17). pp 363–376
23. Rizk A, Poloczek F, Ciucu F (2015) Computable bounds in fork-join queueing systems. Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems. pp 335–346. https://doi.org/10.1145/2796314.2745859
24. Klimovic A, Wang Y, Kozyrakis C, Stuedi P, Pfefferle J, Trivedi A (2018) Understanding ephemeral storage for serverless analytics. 2018 USENIX Annual Technical Conference (USENIX ATC 18). pp 789–794
25. Holubiev V, Ihnatiuk B, Voytyuk I (2018) Next-generation serverless system for contextual search based on rich media content
26. Bharti U, Bajaj D, Goel A, Gupta SC (2021) A novel design approach exploiting data parallelism in serverless infrastructure. In Advances in Computing and Network Communications: Proceedings of CoCoNet 2020, Volume 1, pp. 247-260. Springer Singapore.
27. Giménez-Alventosa V, Moltó G, Caballer M (2019) A framework and a performance assessment for serverless MapReduce on AWS Lambda. Futur Gener Comput Syst 97:259–274
28. Arfat Y, Usman S, Mehmood R, Katib I (2020) Big data for smart infrastructure design: Opportunities and challenges. Smart Infrastructure and Applications: Foundations for Smarter Cities and Societies 491-518.
29. Zheng L, Larson P-A (1996) Speeding up external mergesort. IEEE Trans Knowl Data Eng 8:322–332
30. Zahoor E, Asma Z, Perrin O (2017) A formal approach for the verification of AWS IAM access control policies. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Springer, Cham, pp 59–74
31. Islam M, Huang AK, Battisha M, Chiang M, Srinivasan S, Peters C, Neumann A, Abdelnur A (2012) Oozie: towards a scalable workflow management system for Hadoop. Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies. pp 1–10
32. Wiemuth M, Burgert O (2019) A workflow management system for the OR based on the OMG standards BPMN, CMMN, and DMN. p 79
33. Zhang W, Fang V, Panda A, Shenker S (2020) Kappa: A programming framework for serverless computing. Proceedings of the 11th ACM Symposium on Cloud Computing. pp 328–343
34. Bharti U, Bajaj D, Goel A, Gupta SC (2021) Sequential Workflow in Production Serverless FaaS Orchestration Platform. Proceedings of International Conference on Intelligent Computing, Information and Control Systems. pp 681–693
35. Witte PA, Louboutin M, Modzelewski H, Jones C, Selvage J, Herrmann FJ (2020) An event-driven approach to serverless seismic imaging in the cloud. IEEE Trans Parallel Distrib Syst 31:2032–2049
36. Barcelona-Pons D, Garc\'\ia-López P (2021) Benchmarking parallelism in FaaS platforms. Futur Gener Comput Syst 124:268–284
37. Burckhardt S, Chandramouli B, Gillum C, Justo D, Kallas K, McMahon C, Meiklejohn CS, Zhu X (2022) Netherite: efficient execution of serverless workflows. Proc VLDB Endow 15:1591–1604
38. Sabbioni A, Rosa L, Bujari A, Foschini L, Corradi A (2022) DIFFUSE: A DIstributed and decentralized platForm enabling Function composition in Serverless Environments. Comput Networks 210:108993
39. Leu F-C, Tsai Y-T, Tang CY (2000) An efficient external sorting algorithm. Inf Process Lett 75:159–163
40. Obrutsky S (2016) Cloud storage: Advantages, disadvantages and enterprise solutions for business. Conference: EIT New Zealand

Publisher's Note