Serverless Platform from Scratch

SUVASREE BISWAS (ROLL NUMBER: G32996728) , THARUN GANAPATHI RAMAN (ROLL NUMBER: G28521739), RAJESH REDDY, and MAORUI, George Washington University, USA

Additional Key Words and Phrases: serverless, distributed systems, function as a service

ACM Reference Format:
Suvasree Biswas (Roll Number: G32996728), Tharun Ganapathi Raman (Roll Number: G28521739), Rajesh Reddy, and Maorui. 2023. Serverless Platform from Scratch. 1, 1 (May 2023), 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

1.1 Abstract

Serverless platforms, also known as Function-as-a-Service (FaaS) platforms, are a cloud computing model where the cloud provider manages the infrastructure and dynamically manages the allocation of computing resources to execute code in response to specific events or triggers, without the need for the user to provision or manage servers. Serverless computing is inherently distributed, with functions running on multiple nodes across a network. This distributed architecture can introduce latency, communication overhead, and other performance issues. It may experience a cold start, causing a delay in the function's initial response time, and need to be able to scale and allocate resources as needed to handle varying loads, but scaling can also introduce additional costs. We will implement the serverless platform from scratch and address three key challenges of the distributed systems – Scalability, Fault tolerance, and Concurrency. For each challenge, we're going to use separate approaches to address it - BFT for the fault tolerance, Dependable compute cloud (DC2) based Kalman's filtering for the autoscaling, and DAG for the concurrency. To implement those algorithms and techniques, we're going to use Kubernetes (also known as "K8s") an open-source container orchestration platform that we use to manage pods to handle functions that come from the requests. In addition to it, some more components will be used to mitigate the challenges.

1.2 Notation Table

Here is a list of the notations we have referred to in table 1

1.3 Problem

When the user makes a request in the form of an arithmetic expression to the FAAS API, it intends to exploit the unbounded resources at the disposition and serve the reply at the quickest.

Authors' address: Suvasree Biswas (Roll Number: G32996728) , suvasree@gwmail.gwu.edu; Tharun Ganapathi Raman (Roll Number: G28521739), tharungr@gwu.edu; Rajesh Reddy, rajesh.thiyyagura@gwu.edu; Maorui, , George Washington University, 2121 I St NW, Washington, DC 20052, Washington DC, DC, USA, 20052.

Table 1. Notations vs Meaning table

Notation	Meaning
k	set of arithmetic functions. For us $k = \{+, *, -, \}$
w_k	function worker number k
f_i^j	fault f of type i and index number j
f_1^j	fault at function worker node with index number j
f_2^j	fault at leader replica node with index number j
f_3^j	fault at follower replica node with index number j
f_4^j	fault at control place with index number j

We emphasize that in this document we emphasize on the low level challenges and solutions, since the algorithmic level challenges and their corresponding solution algorithms have already been mentioned in the prior documents. Also the low level challenges and solutions are manifestation of their higher level algorithmic counterparts as stated in the prior submitted documents. Section 1.3.1 has a list of low level challenges, for each of these challenges the corresponding mapped solutions are in section 1.4.1.

1.3.1 ***Low Level Challenges.*** Below are the low level exact low level version of the above challenges faced::

(1) **Fault type I**:
Fault type I refers to a functional node being down.
(2) **Fault type II**:
Fault type II refers to a Leader Replica of a worker function node being down. We are not mitigating this issue
(3) **Fault type III**:
Fault type III refers to a follower replica of Leader Replica being down.
(4) **Fault type IV**:
Fault type IV refers to the case where a fault has taken place at the control plane. We are not mitigating this issue
(5) **Dependent Arithmetic Operation**:
When a request comes from the user having dependent arithmetic operations.
(6) **Request Burst** When a burst of request arrives from the user.

1.4 Proposed Solution

Solving this user level problem translates to implementing a simplified Function as a Service platform. A serverless platform dynamically starts and stops containers or VMs as requests arrive. The basic components include a gateway load balancer, a queueing system that can buffer requests, and an autoscaler that decides when to add or remove nodes.

1.4.1 ***Low Level Solution.***

(1) **PBFT Admin node- Solution to Fault type I** For fault type I, i.e. for a fault at the function pod, the PBFT Admin is responsible for handling the fault at the function node. Here is a list of the execution steps that follow such event
  (a) Admin node, sitting with the control plane, misses a liveliness probe
  (b) Admin pod talks to the ETCD to find out that the liveliness probe missed is at the worker pod
  (c) Dispatcher starts sleeping with a time interval of 10 sec.

   (d) Admin enqueues f_1^j into the event bus
   (e) scheduler polls the event bus before dispatching
   (f) invokes cluster API responsible for creating node
   (g) pod created and placed at worker k
   (h) Dispatcher wakes up, finds the event bus empty, then dispatches next message

   In PBFT paper, the solution to fault type II is given as view change. However we are doing a quick fix by the admin pod responsible for liveliness probe at the admin pbft pod.

(2) **PBFT Admin node- Solution to Fault type II** For fault type II, i.e. for a fault at the Leader Replica pod, the PBFT Admin pod is responsible for handling the fault at the replica node. Here is a list of the execution steps that follow such event
   (a) Admin node, sitting with the control plane, misses a liveliness probe at a pod.
   (b) Admin pod talks to the ETCD and finds out that the liveliness probe is lost at the leader replica.
   (c) Dispatcher starts sleeping with a time interval of 10 sec.
   (d) Admin enqueues f_1^j into the event bus
   (e) scheduler polls the event bus before dispatching
   (f) invokes cluster API responsible for creating node
   (g) pod created and placed at worker k
   (h) Dispatcher wakes up, finds the event bus empty, then dispatches next message

   In PBFT paper, the solution to fault type II is given as view change. However we are doing a quick fix by the admin pod responsible for liveliness probe at the admin pbft pod.

(3) **Quorum certificate - solution to Fault type II** For fault type I, i.e. for a fault at the replica node has taken place.

   Normal Execution - $Algo_{normal}$
   (a) worker W_k gets a request m
   (b) W_k sends it to Replica leader R_L
   (c) R_L has the following sequence of messages with each of the replica R_n
      (i) broadcast to R_n
      (ii) commit to R_n
      (iii) reply from R_n
      (iv) Quorum certificate from R_n
   (d) if the R_L has $2f + 1$ number of quorum certificates from the replicas, R_L replies back the reply to W_k
   (e) W_k replies back reply

   Event Fault at Replica Consider a scenario where, upon message request m at R_L, R_L did not receive $2f + 1$ quorum certificates from the replicas
   (a) R_L informs W_j
   (b) W_j raises fault f_2^j and enqueues into the event bus
   (c) scheduler before dispatching and also periodically polls the event bus
   (d) dequeues f_2^j
   (e) invokes cluster API which creates new pod and places it
   (f) W_j is informed by sufficient number of replicas
   (g) again invokes $algo_{normal}$

(4) **Pod creation** We did not resolve the issue of fault at the control plane as of now.
(5) **DAG - solution to dependent arithmetic operation** Consider the scenario where a request comes from the user with function operations which are dependent on each other. The parser generates dependent and independent sub-expressions of 2 operands. It then enqueues them into the queue.
(6) **Multi-Layered Message Queue- solution to message burst** The message queue of ours has layers equivalent to the number of unique arithmetic operations that are there in the system. This scenario maps to the challenge of **autoscaling** in the distributed system backdrop. The steps of the solution algorithm is enumerated below:
    (a) Since in our system we have 4 types of arithmetic operations hence we have 4 SQS queues as our message queue.
    (b) Also for the event of message burst, our pods are not sets but only k8 pods.
    (c) Above the prior setup is done, let us consider a message burst is served by the user
    (d) The HTTP service enqueues each of this into the incoming request queue.
    (e) The scheduler and Parser service (PR-SC) service is triggered.
    (f) The parser service after parallel parsing of the dependent and independent subexpression, enqueues all of these into the respective layer of the 4 Layered Message Queue, according to the functionality of each subexpression.
    (g) The scheduler Service periodically polls this message queue on each layer
    (h) if the length of any layer > 4, then the scheduler service invokes the cluster service for creating a pod of that particular functionality.
    (i) the scheduler is informed by the control panel about this creation.
    (j) the scheduler then dispatches the next incoming request from the incoming request queue.
    (k) the scheduler makes grpc calls to the pod.
    (l) the pod returns back the result to the scheduler.
    (m) the scheduler enqueues them into the outbound reply queue
    (n) All subresults of the same request id is consolidated into the final result
    (o) The final result is then thrown to the user by the HTTP service.

2 RELATED WORK

Some of the related Faas research platforms that are OpeenFaas [1], OpenLambda [2], vHive [3]. Even though all of these solve different problem statements, they all present solutions to specific optimisations in the serverless function as a service platforms. Like them we also address 3 issues of distributed systems. For the problem of fault tolerance, [4] is referred. For the challenge of autosclaing [5] is referred. For the problem of concurrency, [6] is referred.

3 PROJECT DESIGN

3.1 Overview

On an abstract level, we broadly divide our system as two separate distributed system platforms

(1) **Platform 1** :
    (a) The abstract architectural overview of this platform is that here all pods in the cluster are plain pods and not pod sets.
    (b) A key component of this architecture is the multi layered message queue which the scheduler probes before dispatching.

   (c) used to capture the event of message burst and event of dependent expression request.
(2) **Platform 2** :
   (a) The abstract architectural overview of this platform is that here all pods in the cluster are pod sets in which each set has one functional worker node and a set of replica pods associated with each worker, refer to figure 1
   (b) The information about neighbours inside each set is stored in the ETCD key value store.
   (c) In this architecture the component of ADMIN pod also plays a crucial role of performing health check probes.
   (d) The event bus is an active component of this architecture.
   (e) The above platform is used to capture the event of fault at the follower replica and the worker replica.

3.2 Services

Internally inside the FAAS platform we have we have the following modular services along with the corresponding functionalities exposed:

(1) **Service Name - HTTP** the HTTP service has the following functionalities exposed:
   (a) calculate expression
   (b) get expression
(2) **Service name - Parser and Scheduler** - the parser and scheduler service has the following functionalities exposed:
   (a) Trigger Parsing
   (b) grpc calls
(3) **Service name - Cluster** - the cluster service has the following routes created for it:
   (a) Create Pod
   (b) Get Pod
   (c) Get all Pod
   (d) Delete Pod
   (e) create Service
   (f) Get Service
   (g) Get all Service
   (h) Delete Service

3.3 Honest Execution Flow

Overview of honest working, without any fault handling, without any autoscaling.

(1) user makes a function request in the form of a parameterised arithmetic expression
(2) the HTTP service takes the expression.
   (a) the CalculateExpression function inside the HTTP service is responsible for allocating the unique request id
   (b) the CalculateExpression function then enqueues the request in the inbound Request queue(SQS)
(3) The Cluster Service is independently responsible for doing the following setup:
   (a) pulls the images from the docker repository
   (b) makes the pods with these images at static port
   (c) metadata about the set worker and its replica information gets stored at the ETCD
(4) The Parser and the Scheduler (PR-SC) handler does the following:

   (a) the PR-SC handler dequeues the request from the inbound request queue.
   (b) breaks the request expression into independent and dependent subexpression
   (c) enqueues the independent and dependent subexpressions into the respective layer of the 4 layered message queue(4MQ) (-SQS) for autoscaling
   (d) for each of the independent subexpressions, the scheduler makes immediate GRPC calls to the port of the respective K8 Load balancer port id
   (e) for each of the dependent subexpression, the scheduler sleeps and wakes up periodically before making GRPC calls for each dependent subexpression at their respective Load Balancer port id.
   Hence concurrency is achieved
(5) upon getting a GRPC call, Load Balancer for operation f (say LB_f) does the following:
   (a) allocates f to any $Worker_f$
   (b) $Worker_f$ computes f.
   (c) $Worker_f$ sends the f to its Leader Replicas say R_{L_f}.
   (d) R_{L_f} computes f. R_{L_f} also does the following:
      (i) R_{L_f} broadcasts preprepare message to all the follower replicas R_F
      (ii) R_{L_f} broadcasts prepare message with f to all the follower replicas R_F
      (iii) each of the R_F computes f and returns Quorum Certificate message (Commit) message with the result of f to the R_{L_f}
      (iv) R_{L_f} receives 3 such commit messages with the same result of f computation.
      Consensus reached assuming 1 fault in 4 replicas
      (v) R_{L_f} sends result of f to $Worker_f$
   (e) $Worker_f$ sends the result of f to LB_f
   (f) LB_f sends it to the result of f to the Scheduler
(6) The scheduler puts the result of the expression is put into the outbound reply queue(SQS).
(7) the HTTP service takes the result of the full expression.
   (a) the GetExpression function inside the HTTP service is responsible for dequeueing the result of a certain request id from the outbound reply queue(SQS)
   (b) the GetExpression then outputs the dequeued result to the end user.

---

4 DISTRIBUTED SYSTEM CHALLENGES

4.1 Heterogeneity - implied

Distributed applications are typically heterogeneous having different hardware, different software, diverse networks and protocols. A distributed system is said to be robust with respect to heterogeneity if the user remains oblivious to the differences in the underlying differences in resources. We have EC2 instances of T2.micro, SQS instances, EKS instances. These all have different hardware specifications whose difference in execution times is obfuscated from the user by the efficient load balancing techniques at every level.

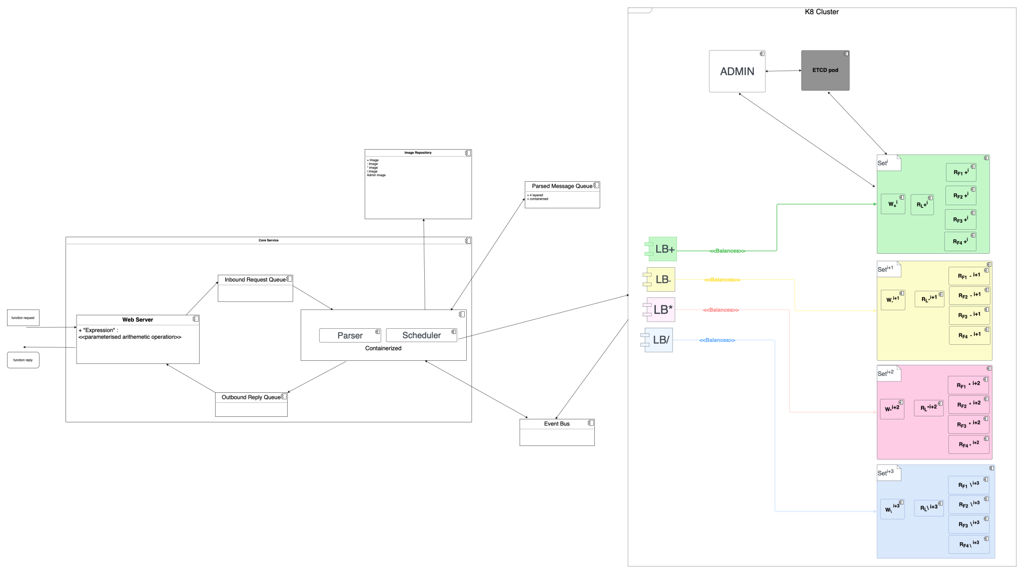

Fig. 1. Component Diagram

4.2 Openness - implied

An open distributed system is a system that offers services according to standard rules that describe the syntax and semantics of those services. We take as input an expression in the following form :

$$\text{"}string\text{"} : paranthesized\ arithmetic\ expression \tag{1}$$

An example of this expression is: $\text{"}expression\text{"} : ((3+5)*(4/2))$. This will be parsed as three expressions $f1 = 3+5$, $f2 = 4/2$, and $f3 = f1 * f2$. Therefore, there are two independent expressions and one dependent expression. Our parser technique makes the dependent function $f3$. It would periodically wake up and keep probing whether the independent sub expressions are evaluated or not. Only if it is evaluated, will then the dependent expression will be dispatched to the k8 cluster by the load balanced scheduler. The user just needs to give input a string followed by a colon and a paranthesized arithmetic expression.

4.3 Security - partially handled

The PBFT [7] algorithm uses secret key cryptography primitives like MAC and hashes. In particular we are using MD5 hash to create digest of the messages and RSA keypairs to allow worker pod, leader replica and follower replica to share key pairs. Each of the worker replica share RSA key pair with the Leader replica and subsequently the leader replica shares RSA key pairs with the each of the follower replica. Message exchange between the worker pod and the leader replica happens through these RSA keypair unique to each $(worker, replica_{leader})$ pair. Subsequently message exchange between the leader replica and the follower replica happens through RSA key pair mapped to each such $(replica_{leader}, replica_i), \forall i \in [n-1]$ where n is the total number replica set alloted to each worker pod. Each of the replicas are also responsible for signing each of the packets it exchanges, in order for the open channel to differentiate between the broadcasted messages.

$$Client \quad \underset{\beta_{RSA_{C,R}}=<Reply,v,t,c,i,r>}{\overset{\alpha_{RSA_{R,C}}=<request,o,t,c>}{\rightleftharpoons}} \quad Replica$$

where o is the state machine operation, t is the timestamp, c is the client id, v is the current view number, i is the replica number, and r is the result of executing the requested operation. We do not have the view notion in our system currently implemented because the fault at worker pod is handled explicitly by the admin node. This acts as the certificate of request and reply. Receiving these certificates ensures that the quorum has consensus and then the Byzantine agreement would guarantee correct proactive recovery.

4.4 Failure Handling - handled

Byzantine Generals Problem states that if f nodes can be arbitrarily malicious, you need 3f+1 replicas to guarantee a correct response (stateful or stateless). We follow PBFT algorithm [7].

Under normal circumstances, the algorithm goes as follows:

(1) A client sends a request to invoke a service operation to the primary (Leader Replica)
(2) The Replica Leader multicasts the request to the backups
(3) Follower replicas execute the request and send a reply to the client
(4) The client waits for $2f + 1$ replies from different replicas with the same result; this is the result of the operation. In effect, the leader replica on receiving $2f + 1$ replies from the followers, send the reply back to the function worker node.

Upon a fault happening at the replica as well as the worker node, what our system does is described in section 2 and 3.

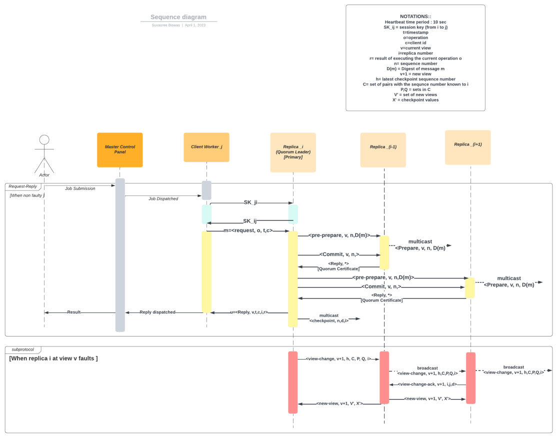
Sequence diagram
Suvasree Biswas | April 1, 2023
NOTATIONS::
Heartbeat time period : 10 sec
SK_ij = session key (from i to j)
t=timestamp
o=operation
c=client id
v=current view
i=replica number
r= result of executing the current operation o
n= sequence number
D(m) = Digest of message m
v+1 = new view
h= latest checkpoint sequence number
C= set of pairs with the sequnce number known to i
P,Q = sets in C
V' = set of new views
X' = checkpoint values
Actor
Master Control Panel
Client Worker_j
Replica_i (Quorum Leader) [Primary]
Replica_(i-1)
Replica_(i+1)
Request-Reply
[When non faulty]
Job Submission
Job Dispatched
SK_ji
SK_ij
m=<request, o, t,c>
<pre-prepare, v, n,D(m)>
multicast <Prepare, v, n, D(m)>
<Commit, v, n,>
<Reply, *> [Quorum Certificate]
Result
Reply dispatched
u=<Reply, v,t,c,i,r>
multicast <checkpoint, n,d,i>
subprotocol
[When replica i at view v faults]
<view-change, v+1, h, C, P, Q, i>
broadcast <view-change, v+1, h,C,P,Q,i>
<view-change-ack, v+1, i,j,d>
<new-view, v+1, V', X'>

Fig. 2. Sequence Diagram depicting the action taken at (i) fault at replica (ii) fault at function worker node

4.5 Concurrency - handled

Concurrency considers the checkpoints. It refers to how the tasks are divided in the same CPU. Whereas in contrast, Parallelism considers time of progresses. Thereby it refers to the problem of handling parallel tasks in multiple CPUs.

Some of the mechanisms for the mitigation of the problem of distributed coordination comes (i) Distributed Locking (ii) Consensus (iii) Elections (iv) State Machine Replication. The standard distributed locking mechanisms are distributed and centralised. Election enables us to appoint a central coordinator. But the algorithm must allow the leader to be replaced in a safe, distributed way. Bully algorithm and ring algorithm are the main schemes in the literature. Some of the key features that our system supports are as follows:

(1) **Election**Our system supports a election plus centralised locking architecture inside the replica cluster for each of the function worker node. In our implementation, the election algorithm implementation deployed is that of a randomised one.
(2) **Locking** standard avenues of locking mechanism in the literature is centralised locking and decentralised locking. We follow a centralised locking mechanism as simple as the lines of lamports's algorithm. All mutex shared in our system is via this simple logic.
(3) **Consensus Correctness**: Inside the K8 Control plane(CP), there lies the 4 PN nodes which are responsible for consensus correctness. Assume a simpleton architecture when we are trying to accommodate only 1 fault in any of the function of replica pod.
(4) **Parallel Parser**: The parallel parser is also helping in achieving concurrency by allowing dependent expressions to be executed in parallel.

4.6 Quality Of Service

QoS refers to the metrics used to gauge the performance metrics of a system. Performance takes into consideration the following parameters (i) Latency (ii) Bandwidth (iii) Throughput (iv) Response time. Our system is not really a performance driven project. comparison. Even though our system might take more time than its K8 contemporary our customised version gives it more freedom than the k8 system. We record the following:

(1) **K8 Fault Tolerance Vs Our Fault Tolerance**
(2) **K8 Autoscaling Vs our autoscaling**
(3) **K8 concurrency Vs Our Concurrency**

We hereby note that the above comparative analysis are for future scope and hold insignificance in our backdrop.

4.7 Scalability - handled

Scalability means you can increase or reduce the capacity, diversity, power or abilities of your system. One of the major avenues of mesauring scalability is with respect to size. A system is scalable with respect to its size means adding more resources to the system, upon a message burst – referred to as *Scaleup*.

One of the major ways to ensure scalability is through **replication**. Data are replicated to increase the reliability of a system. But replication demands trade-off of performance. But with replica, comes creeping in the problem of maintaining **consistency**. This originated framing of consistency models. A consistency model describes what can be expected when multiple processes concurrently operate on a set of data. The set is then said to be consistent if it adheres to the rules described by the model. Some of the common models are that of lamport, raft, praxo. What our system supports:

(1) In general our system follows a **client centric consistency model with monotonic reads and writes**. The scheduler of our system goes on to sleep and periodically wakes up after a period of 10 seconds. Upon waking up and also before dispatching any request to the cluster, the scheduler checks that the event bus containing the

fault events is empty or not. If it is not empty it goes back to sleep. Our scheduler dispatches iff event bus is empty. Hence our system supports client centric consistency model with monotonic reads and writes. This is one of the ways of ensuring consistency in our system. Our system does not handle coherence of data.

(2) Our system also supports **quorum certificates** in the following way. For each of the functional worker node, one of the replica is elected as the leader replica. Hence we support the **single replica system** supporting linearizability. For the exact sequence of message exchange between the replica, refer to first half of Figure 2. Only after the leader gets 2f+1 quorum certificates, will the leader replica reply back to the function worker node. Not only the quorum certificate ensures consistency, it allows ensures better performance, durability and availability.

(3) Our system also supports **autoscaling**. We intended to have predictive autoscaling of the message queue via kalman filtering [5]. This is yet to be done. We do accommodate a static autoscaling mechanism. We have a 4 layered message queue, one for each functionality. The scheduler while enqueing a message at a particular layer probes whether the length of the queue is greater than 4 or not, if yes, it invokes the service API which is responsible for creating worker pod of that particular functionality in the cluster.

4.8 Transparency - implied

Transparency in simple words is defined as the concealment from the user and the application programmer of the separation of components in a distributed system, so that the system is perceived as a whole rather than as a collection of independent components. In our system transparency is implied by the user having to enter only a parenthesized arithmetic expression (as described in section 4.2), and getting the correct result back after a stipulated period of time. The beauty lies in the fact that the user remains completely oblivious to the internal architecture and orchestration .

---

5 PROJECT OUTCOMES AND REFLECTION

We build two broad cases of our system.

(1) **Case I : Autoscaled Serverless Platform for Arithmetic as a service system:** A request burst initiates this case, which is then handled by a load balanced parallel parser for concurrency, and the cluster sets are autoscaled as a function of the length of the incoming request burst.

(2) **Case II : Fault tolerant Serverless platform for Arithmetic as a service system:** This case tries to encompass two independent scenarios:
   (a) Scene I : when a message has arrived at the worker, and after that only 3 quorum certificates are got back from the replica set of size 4.
   (b) Scene II : when admin pod misses a liveliness check probe at the functional worker node.

All pods in our system are EC2 t2.medium instances. The parser scheduler of ours lies inside an ECS instance of size t2.medium, which is a cluster responsible for hosting these containers. Every queue system of ours are SQS instances. We also

5.1 Deploy application inside the pod

One of the major task was deploying application inside the pods.

(1) develop application
(2) do containerization

(3) store the image in the dockerhub repository
(4) deploy the cluster using the cluster service
(5) the cluster service has a go client which talks to the kube api server and deploy the containerized application inside the pod

5.2 Develop containerized application

Another major core task was developing containerized application

(1) write the code base for our application
(2) create the dockerfile
(3) use the docker command to containerize the application
(4) tag it with nomenclature and push it to the repository

5.3 Queue system

5.3.1 Inbound Request Queue And Outbound Reply Queue.

(1) create an SQS instance for the Inbound Request Queue
(2) the ECS containerized parser and scheduler interface dequeues from this inbound Request queue
(3) create an SQS instance for the Outbound Reply Queue
(4) the ECS containerized parser and scheduler interface enqueues reply inside the outbound reply queue

5.3.2 Autoscaling Queue.

(1) Forward Direction
  (a) We create 4 SQS instances for the autoscaling queue, one layer for each arithmetic functionality our system is serving
  (b) The ECS interface enqueues every sub-request into the respective layer of the autoscaling queue
  (c) the scheduler while enqueueing also probes and performs necessary autoscaling activities by invoking the Cluster API
(2) Backward Direction
  (a) when reply of a unique uuid of the request id is got by the scheduler, it probes the particular layer of SQS and dequeues it
  (b) the scheduler forwards the reply to the HTTPS service handler

5.3.3 Event Bus. Our Event Bus is also a SQS instance. The scheduler and the worker pod are only component in our system that interacts with the event bus.

5.4 Parsing

(1) the parser service has a parser expression route (REST API) which is being triggered by the SQS when there is an expression in the SQS
(2) this parser expression has a function which evaluates and parses the given expression and divides it into subexpression of size 2 operator and 1 operation
(3) the scheduler gets these subexpressions, scheduler triggers the worker GRPCs to do the computations

6 CONCLUSION AND FUTURE SCOPE

All the components of our system run successfully yielding the correct function execution results. One interesting future scope is analyzing the fault at control plane scenario. Another exciting future scope is that of analyzing the time complexity of the kalman filtering based autoscaling approach with that of our autoscaling approach.

REFERENCES

[1] D.-N. Le, S. Pal, and P. K. Pattnaik, “Openfaas,” *Cloud Computing Solutions: Architecture, Data Storage, Implementation and Security*, pp. 287–303, 2022.

[2] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “Serverless computation with openlambda,” in *8th {USENIX} workshop on hot topics in cloud computing (HotCloud 16)*, 2016.

[3] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot, “Benchmarking, analysis, and optimization of serverless function snapshots,” in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 559–572.

[4] R. Rodrigues, M. Castro, and B. Liskov, “Base: Using abstraction to improve fault tolerance,” *ACM SIGOPS Operating Systems Review*, vol. 35, no. 5, pp. 15–28, 2001.

[5] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang, “Adaptive, model-driven autoscaling for cloud applications,” in *11th International Conference on Autonomic Computing ({ICAC} 14)*, 2014, pp. 57–64.

[6] H. Shafiei, A. Khonsari, and P. Mousavi, “Serverless computing: a survey of opportunities, challenges, and applications,” *ACM Computing Surveys*, vol. 54, no. 11s, pp. 1–32, 2022.

[7] M. Castro and B. Liskov, “Practical byzantine fault tolerance and proactive recovery,” *ACM Transactions on Computer Systems (TOCS)*, vol. 20, no. 4, pp. 398–461, 2002.