CHAPTER 9

Microservices Using Function-as-a-Service

Functions-as-a-Service (FaaS) is a "serverless" solution that has become increasingly popular in recent years. FaaS solutions enable individuals to build, manage, deploy, and scale application functionality without having to manage infrastructural overhead. They can provide significant value in event-driven systems as a means of implementing simple to moderately complex solutions.

A *function* is a piece of code that is executed when a specific triggering condition occurs. The function starts up, runs until completion, and then terminates once its work is completed. FaaS solutions can easily scale the number of function executions up and down depending on load, providing close tracking for highly variable loads.

It may be helpful to think of a FaaS solution as a basic consumer/producer implementation that regularly fails. A function will always end after a predetermined amount of time, and any connections and state associated with it will go away. Keep this in mind as you design your functions.

Designing Function-Based Solutions as Microservices

FaaS solutions may comprise many different functions, with the sum of their operations constituting the solution to the business bounded context. There are many ways to create function-based solutions, far more than this chapter can cover, but there are a few general design principles that will help guide you through the process.

Ensure Strict Membership to a Bounded Context

The functions and internal event streams composing a solution must strictly belong to a bounded context, such that the owner of the function and data is clearly identified. It is common for an organization to have ownership questions around functions, services, and event streams when implementing microservice solutions in large

numbers. While many microservice solutions map 1:1 to a bounded context, an *n*:1 mapping is not uncommon, as multiple functions may be used for a single bounded context. It's important to identify *which* function belongs to *which* bounded context, because the high granularity of functions can blur those lines.

Some practical ways of maintaining bounded contexts with functions include:

- Ensure that data stores are kept private from external contexts.
- Use standard request-response or event-driven interfaces when coupling to other contexts.
- Maintain strict metadata around which function belongs to which context (a 1:1 mapping of function to product).
- Maintain function code within a repository mapped to the bounded context.

Commit Offsets Only After Processing Has Completed

Offsets are committed at one of two times: when the function starts or when the function has completed its processing. Committing offsets only after processing has completed for a given event or batch of events is a FaaS best practice. It is important for you to understand how offsets for your specific function-based solutions are handled, so let's take a look at the implications of each approach.

When the function has completed its processing

The approach of committing offsets after processing has completed aligns with how other microservice implementations commit offsets, whether they're based on a basic producer/consumer or stream-processing framework. This strategy provides the strongest guarantee that events will be processed at least once and is equivalent to the offset management strategies used with non-FaaS solutions.

When the function has first started

Offsets may be committed once the batch of events to process has been passed off to the function. This simple approach is used in many FaaS frameworks that rely on framework-specific retry mechanisms and alerting to mitigate repetitive event processing failures. Functions that call other functions in a choreography pattern often rely on this strategy, as it greatly simplifies the tracking of event processing.

Committing offsets before processing has completed can be problematic, however. If the function is unable to successfully process an event, and numerous retries fail, then data loss is likely. The event will typically be shunted into a dead-letter queue or simply discarded. While many function-based microservices are not sensitive to data loss, those that are should not use this strategy.

Less Is More

An often-cited feature of FaaS frameworks is that they make it easy to write a single function and reuse it in multiple services. Following this approach, however, can lead to a highly fragmented solution that makes it difficult to discern exactly what is going on within the bounded context. Additionally, ownership of a given function becomes ambiguous, and it may not be clear if changes to a function could negatively affect other services. While versioning of functions can help with this issue, it can also lead to conflicts when multiple products need to maintain and improve different versions of a function.

FaaS solutions may incorporate multiple functions to solve the business requirements of the bounded context, and while this is not an uncommon or bad practice, a good rule of thumb for FaaS solutions is that fewer functions are better than many granular functions. Testing, debugging, and managing just one function is much easier than doing the same for multiple functions.

Choosing a FaaS Provider

Just like the event brokers and container management systems (CMSes), FaaS frameworks are available as both free-to-use open source solutions and paid third-party cloud providers. An organization that runs its own in-house CMS for its microservice operations can also benefit from using an in-house FaaS solution. There are a number of free open source options available, such as OpenWhisk, OpenFaaS, and Kubeless, that can leverage existing container management services. Apache Pulsar offers its own built-in FaaS solution that runs alongside its event broker. By leveraging a common resource-provisioning framework, your FaaS solution can align with your microservice solution.

Third-party service providers such as Amazon Web Services (AWS), Google Cloud Platform (GCP), and Microsoft Azure also have their own proprietary FaaS frameworks, each of which offers attractive features and functionality but remains tightly integrated with its provider's proprietary event broker. This is a significant issue only because currently all three providers limit retention within their event brokers to seven days. Integrations between cloud providers and open source event brokers do exist (such as Kafka Connect), but may require additional effort to set up and manage. That being said, if your organization is already a subscriber to AWS, GCP, or Azure services, then the overhead to start experimenting is low.

Building Microservices Out of Functions

There are four main components you must consider when working with function-based solutions, regardless of your FaaS framework or event broker:

- The function
- Input event stream
- Triggering logic
- Error and scaling policies, with metadata

The first component of a FaaS implementation is the function itself. It can be implemented in any code supported by the FaaS framework.

```
public int myfunction(Event[] events, Context context) {
  println ("hello world!");
  return 0;
}
```

The `events` parameter contains the array of individual events to be processed, with each event containing a `key`, `value`, `timestamp`, `offset`, and `partition_id`. The `Context` parameter contains information about the function and its context, such as its name, the event stream ID, and the function's remaining lifespan.

Next, you need to wire up some triggering logic for the function. This will be covered in greater detail in the next section, but for now, say that the function is triggered whenever a new event arrives in one of its subscribed event streams. The triggering logic is often associated with a function by way of a *function-trigger map*, which is usually concealed behind the scenes of your FaaS framework. Here is an example:

Function	Event stream(s)	Trigger	Policies and metadata
`myFunction`	`myInputStream`	`onNewEvent`	`< ... >`

You can see that `myFunction` is set to trigger when a new event is delivered to `myInputStream`. You'll also notice that there is a column named Policies and Metadata, which is a bit of a catch all that includes configurations such as the following:

- Consumer group
- Consumer properties, such as batch size and batch window
- Retry and error handling policies
- Scaling policies

Once the triggers, metadata, and policies are established, the function is ready to process incoming events. When a new event arrives in its input event stream, the function will be started by the FaaS framework, get passed a batch of events, and begin processing. Upon completion, the function will terminate and wait for more events to come in. This is a typical implementation of the event-stream listener pattern, which is discussed more in the next section.

Each function-based microservice implementation must have its own independent consumer group, just as with other non-FaaS microservices.

Now keep in mind that this is just a logical representation of the components needed to successfully trigger and operate a function. A FaaS framework's function coding requirements, function management, and triggering mechanisms vary by provider and implementation, so be sure to refer to the documentation for your FaaS framework.

There is also a moderately complex interplay between triggering mechanisms, event consumption, consumer offsets, nested functions, failures, and at-least-once event processing. These are the subject of the remainder of this chapter.

Cold Start and Warm Starts

A *cold start* is the default state of the function upon starting for the first time, or after a sufficient period of inactivity. A container must be started and the code loaded, event broker connections may need to be created, and any other client connections to external resources need to be established. Once everything is stable and ready to process, the function is now in a warm state and ready to process events. The warm function begins processing events, and upon expiry or completion, is suspended and put into hibernation.

Most FaaS frameworks attempt to reuse terminated functions whenever possible. In many scenarios, a function processing a steady stream of events will hit the timeout expiry and be briefly terminated, just to be brought back a moment later by a triggering mechanism. The suspended instance is simply reused, and provided that the connections to the event broker and any state stores haven't expired during the interim, processing can resume immediately.

Starting Functions with Triggers

Triggers are used to tell a function to start up and begin processing. The supported triggers vary depending on your FaaS framework, but tend to all fall into the same general categories, as described shortly. For now, let's take a look at which signals can be used to kick off functions, to give you an idea of when you may want to use them.

Triggering Based on New Events: The Event-Stream Listener

Functions can be triggered when an event is produced into an event stream. The *event-stream listener trigger* isolates event consumption behind a predefined consumer, reducing the amount of overhead code that a developer must write. Events are

injected directly into the function in a form of an array of events, in sequential order from an event stream, or as a cluster of unordered events if consuming from a queue. You can create multiple mappings from event streams to functions, such that a function can consume events from many different streams.

FaaS solutions from Google, Microsoft, and Amazon provide this trigger for usage with their proprietary event brokers, but currently do not support triggering directly from open source brokers. The generalized structure of this approach is shown in Figure 9-1.

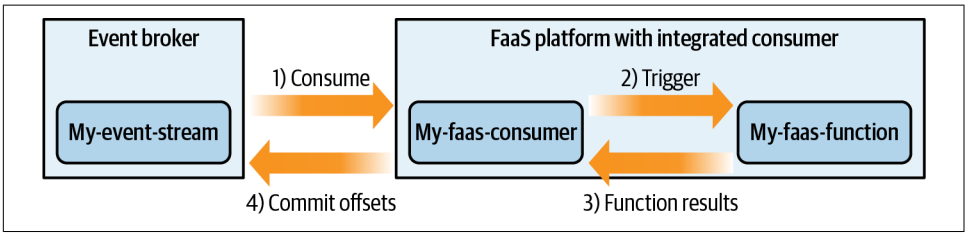

Figure 9-1. Integrated event-stream listener with FaaS framework

Conversely, open source solutions such as OpenFaaS, Kubeless, Nuclio, and others provide a variety of triggering plug-ins with various event brokers, such as Kafka, Pulsar, and NATS, to name a few. For instance, Apache Kafka Connect allows you to trigger the functions of third-party FaaS frameworks. Since Kafka Connect runs outside of the FaaS framework, you would end up with an event-stream listener as per Figure 9-2.

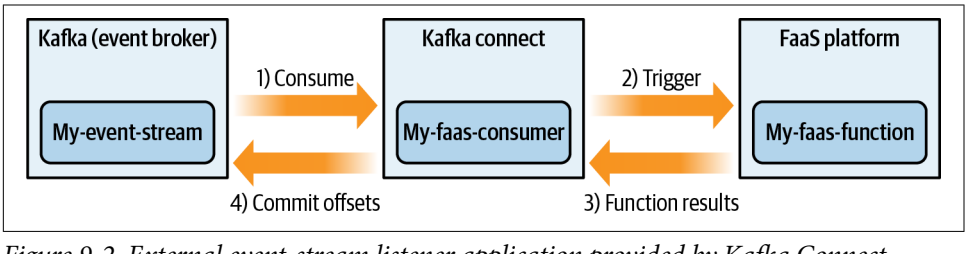

Figure 9-2. External event-stream listener application provided by Kafka Connect

Though not shown in the previous examples, function results can be output to their own event streams, not just for the purpose of outputting data but also for tracking the function's success.

Synchronous triggers require the function to complete before they issue the next events. This is particularly important for maintaining the processing order and is limited by the parallelism of the event stream being processed. Conversely, *asynchronous* triggering can issue multiple events to multiple functions, each one reporting

back as it is completed. This will *not* maintain the processing order, however, and should be used only when processing order is not important to the business logic.

Batch size and *batch window* are two important properties to consider in stream-listener triggers. The batch size dictates the maximum number of events to dispatch for processing, while the batch window indicates the maximum amount of time to wait for additional events, instead of triggering the function immediately. Both of these parameters are used to ensure that the overhead of starting the function is spread among the batch of records to reduce costs.

A function executed by a stream-listener trigger looks something like the following:

```
public int myEventfunction(Event[] events, Context context) {
  for(Event event: events)
      try {
        println (event.key + ", " + event.value);
      } catch (Exception e) {
        println ("error printing " + event.toString);
      }
  //Indicates to the FaaS framework that batch processing was completed.
  context.success();
  return 0;
}
```

Much like a containerized microservice, triggers for the event-stream listener pattern can be configured to start processing events from a stream's latest offsets, earliest offsets, or anywhere in between.

Triggering Based on Consumer Group Lag

A consumer group's lag metric is another way to trigger functions. You can detect lag by periodically polling the offsets of an individual application's consumer groups and computing the delta between the current consumer offset and the head offset of the stream (see "Consumer Offset Lag Monitoring" on page 246 for more on lag monitoring). While similar to the stream listener trigger, lag monitoring can also be used for scaling non-FaaS microservices.

Lag monitoring typically involves computing and reporting lag metrics to your monitoring framework of choice. The monitoring framework can then call the FaaS framework to tell it to start up the functions registered on the event stream. A high lag value can indicate that multiple function instances can be started to more quickly process the load, while a low lag value may require only a single function instance to process the backlog. You can tailor the relationship between lag quantity and function startup on a microservice-by-microservice basis, ensuring compliance with SLAs.

One of the major differences between the previously mentioned event-stream listener trigger and this one is that with lag triggering, the function does *not* consume the events until *after* it is started. Functions started by the lag trigger have a wider domain of responsibilities, including establishing a client connection with the event broker, consuming the events, and committing back any offset updates. This makes functions triggered by lag much more similar to basic producer/consumer clients, albeit with a limited lifespan. The following example function illustrates this workflow:

```
public int myLagConsumerfunction(Context context) {
  String consumerGroup = context.consumerGroup;
  String streamName = context.streamName;

  EventBrokerClient client = new EventBrokerClient(consumerGroup, ...);

  Event[] events = client.consumeBatch(streamName, ...);

  for(Event event: events) {
      // Do event processing work
      doWork(event);
  }
  //Commit the offsets back to the event broker
  client.commitOffsets();

  //Indicates to the FaaS framework that the function succeeded.
  context.success();

  //Return, letting the lag-triggering system know it was a success
  return 0;
}
```

The consumer group and stream name are passed in as parameters in the context. The client is created, events are consumed and processed, and the offsets are committed back to the event broker. The function indicates a `success` result back to the FaaS framework and then returns.

If the function is frequently triggered by the lag monitor, there is a good chance that it will still be warm from the last iteration, and the overhead of connecting to the event broker client may not be a factor. This, of course, depends on the timeouts used by the client and event broker configurations. For longer periods of inactivity, consumer group rebalancing and client cold starts will slightly reduce the amount of work that a function instance can process.

Triggering on a Schedule

Functions can also be scheduled to start up periodically and at specific datetimes. The scheduled functions start up at the specified interval, poll the source event-streams for new events, and process them or shut down as necessary. The polling period

should be kept low so that SLAs are maintained, but polling too frequently may put undue load on both the FaaS framework and event broker.

Client code for a time-based trigger looks identical to that of the consumer group lag trigger example.

Triggering Using Webhooks

Functions can also be triggered by direct invocation, allowing custom integration with monitoring frameworks, schedulers, and other third-party applications.

Triggering on Resource Events

Changes made to resources can also be a source of triggers. For instance, creating, updating, or deleting a file in a filesystem can trigger functions, as can the same modifications made to a row in a data store. Since most of the events in the event-driven microservice domain are generated via event streams, this particular resource trigger isn't often used in most business workflows. It is, however, quite useful when you are integrating with external sources of data that require an FTP or other file service to drop their files into.

Performing Business Work with Functions

FaaS approaches work particularly well for solutions that can leverage the on-demand, flexible nature of processing resource provisioning. Simple topologies are a great candidate, as are those that are stateless, those that do not require deterministic processing of multiple event streams, and those that scale very wide, such as queue-based processing. Anything with a highly variable volume can benefit from FaaS solutions, as their horizontal scaling capabilities and on-demand characteristics allow for rapid provisioning and release of compute resources.

FaaS solutions can perform extremely well when concurrency and determinism are not concerns. However, once determinism comes into play, you must take great care to ensure correctness and consistency in the event stream processing. Much like the basic consumer solutions in the next chapter, FaaS solutions require that you provide an event scheduler to ensure consistent processing results. Copartitioned data can only be successfully and consistently processed by a single function at a time, similar to how the full-featured lightweight and heavyweight frameworks must use only a single thread.

Maintaining State

Given the short lifespan of functions, most stateful FaaS-based solutions require the use of an external stateful service. Part of the reason is the goal of many FaaS providers to supply quick, highly scalable units of processing power independent of the data's location. Having functions that require local state from previous executions limits current execution to the nodes that have that state co-located. This greatly reduces the flexibility of FaaS providers, so they often enforce a "no local state" policy and require that everything stateful be stored external to the executors.

Although previous local state may be available if a function starts in a warm state, there is no guarantee that this will always be the case. Functions connect to external state stores exactly as any other client would—by creating a connection to the state store and using the corresponding API. Any state must be persisted and retrieved explicitly by the function.

Be sure to use strict access permissions for your function's state, such that nothing outside of its bounded context is allowed access.

Some FaaS frameworks have added durable stateful function support, such as Microsoft Azure's Durable Functions, which abstracts away the explicit management of state and allows you to use local memory, which is automatically persisted to external state. This allows developers to suspend functions and bring them back to life without needing to write code to explicitly store and retrieve the state. This greatly simplifies stateful workflows and provides the option to standardize state management across function implementations.

FaaS frameworks will continue to grow and include new features. Simple management of state is a common need in function-based solutions, so keep a lookout for state handling improvements in the FaaS frameworks of your choice.

Functions Calling Other Functions

Functions are often used to execute other functions and may also be used for both choreographed and orchestrated workflows. Communication between functions can be facilitated asynchronously through events, via request-response calls, or with a combination of both. These choices depend highly on the FaaS framework and the problem space of the bounded context. It's common to use choreography and orchestration design patterns when implementing a multifunction solution.

To avoid out-of-order processing issues, ensure that all processing is completed for one event before processing the next event.

Event-Driven Communication Pattern

The output of one function can be produced into an event stream for another consuming function. A bounded context may be made up of many functions and many internal event streams, with varying triggering and scaling logic for each function definition. Each function processes incoming events at its own rate, with events being consumed, work being performed, and outputs produced accordingly. An example of this design is shown in Figure 9-3.

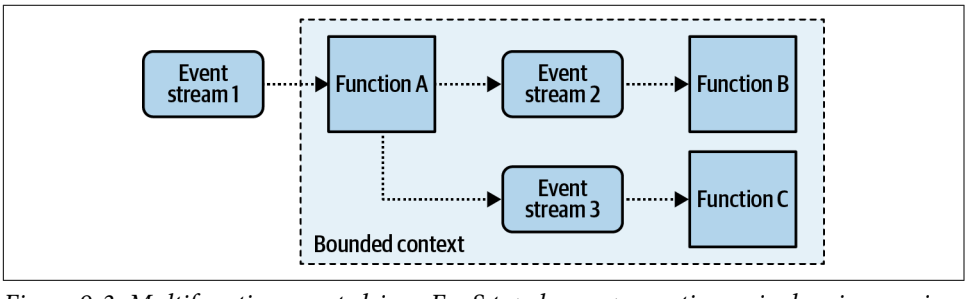

Figure 9-3. Multifunction event-driven FaaS topology representing a single microservice

In this example, function A is triggered independently of the triggers for functions B and C. Event streams 2 and 3 are considered internal event streams, with access to their contents completely restricted to any functions outside of the bounded context. Each function consumes events from its source stream using the same consumer group, as the functions are all colocated within the same bounded context. This ensures that the functions are effectively operating in the same way as a non-FaaS-based microservice.

There are several benefits of using an event-based communication pattern. Each function within the topology can manage its own consumer offsets, committing each offset once its work is done. No coordination between functions needs to occur outside of the event stream processing. The preceding figure shows a choreography-based design pattern, though orchestration can also be used. Additionally, any failures in the event processing will not result in any data loss, as the events are durably stored in the event broker and will simply be reprocessed by the next function instance.

Direct-Call Pattern

In the direct-call pattern, a function can call other functions directly from its own code. Direct invocation of other functions can be performed asynchronously, which is essentially a "fire-and-forget" approach, or synchronously, where the calling function awaits for a return value.

Choreography and asychronous function calling

Asychronous direct calls lead to a choreography-based FaaS solution. One function simply invokes the next one based on its business logic, leaving it up to that function and the FaaS framework to handle the next steps, including any failures or errors. An asynchronous direct-call function topology is a simple way to chain function calls together. Figure 9-4 illustrates an example.

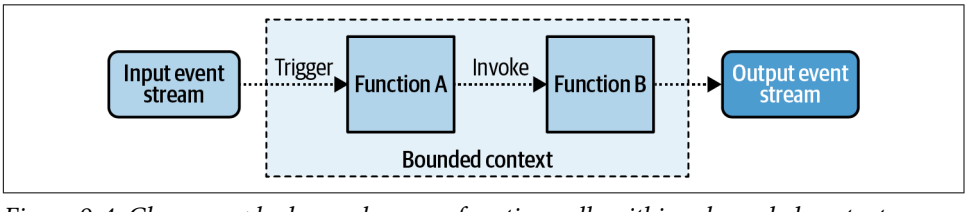

Figure 9-4. Choreographed asynchronous function calls within a bounded context

Function A invokes function B as it processes its batch of events, and once done, function A can simply update its consumer offsets and terminate. Meanwhile, function B continues its processing and produces any outputs to the output event stream.

One major downside to asynchronous direct calls is in ensuring that the consumer offsets are updated only in the case of successful processing. In the example, function B has no feedback to function A, and so only errors in function A will prevent the workflow from incorrectly committing the consumer group offsets. However, losing events may not matter to some workflows, and in those cases this problem can be dismissed.

Another potentially major issue is that events may be processed out of order due to multiple invocations of function B. Consider the code for function A:

```
public int functionA(Event[] events, Context context) {
  for(Event event: events) {
    //Do function A's processing work

    //Invoke function B asynchronously per event
    //Does not wait for return value
    asyncfunctionB(event);
  }
  context.success();
  return 0;
}
```

Function B is called inline with the work from function A. Depending on your FaaS framework, this may result in multiple instances of function B being created, each of which runs independently of the others. This will create a race condition where some executions of function B will finish before others, potentially leading to out-of-order processing of events.

Similarly, writing your code as follows will not solve ordering problems either. Processing will still happen out of order, as function A's processing work will be executed for all events in the batch prior to function B's execution.

```
public int functionA(Event[] events, Context context) {
  for(Event event: events) {
    //Do function A's processing work
  }
  //Invoke function B asynchronously with entire batch of events
  asyncFunctionB(events);

  context.success()
  return 0;
}
```

In-order processing requires strictly executing function A before B, for each event, before processing the next event. An event must be completely processed before the next one can be started; otherwise, nondeterministic behavior will likely result. This is particularly true if functions A and B are acting on the same external data store, as function B may rely on data written by function A.

In many cases, asynchronous calls are not sufficient for the needs of the bounded context. In these cases, consider whether using synchronous calls in an orchestrated manner is more suitable.

Orchestration and synchronous function calling

Synchronous function calls allow you to invoke other functions and await the results before proceeding with the remaining business logic. This allows for the implementation of the orchestration pattern, as covered in Chapter 8.

Event stream—triggered processing. In the following example, a single orchestration function is triggered when new events arrive in a partitioned event stream. The function starts up and begins processing the input batch of events, dispatching event sequentially for each function. Figure 9-5 shows an example of function-based orchestration within a single bounded context.

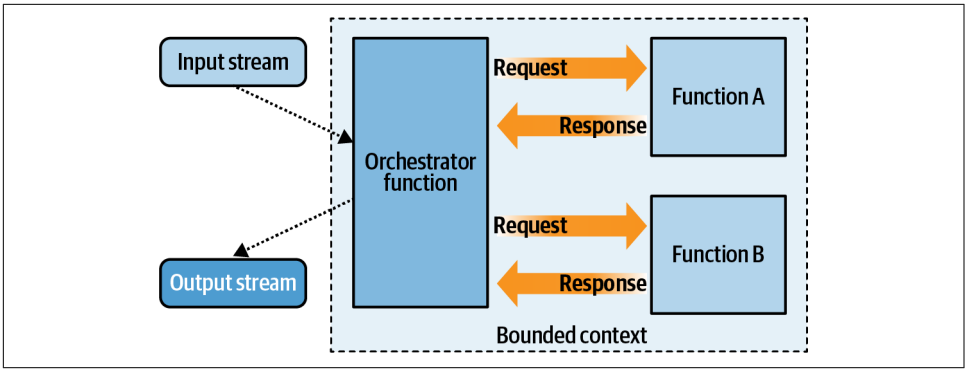

Figure 9-5. Orchestrated synchronous function calls within a bounded context

Here is an example of the orchestration code:

```
public int orchestrationFunction(Event[] events, Context context) {
  for(Event event: events) {
    //Synchronous function Calls
    Result resultFromA = invokeFunctionA(event);
    Result resultFromB = invokeFunctionB(event, resultFromA);
    Output output = composeOutputEvent(resultFromA, resultFromB);
    //Write to the output stream
    producer.produce("Output Stream", output);
  }
  //This will tell the FaaS framework to update the consumer offsets
  context.success();
  return 0;
}
```

The orchestration function invokes functions A and B in order and awaits the results from each function. The output of function A can be sent to B if needed. Each event is fully processed through the workflow before the next one is started, ensuring that offset order is respected. Once the consumer function has completed processing each event in the batch, it can issue a success message and update the offsets accordingly.

Queue-triggered event processing. If you use a queue with individual commit capabilities, the triggering mechanism can simply trigger an individual orchestration function for each event. The orchestrator will need to commit the processing confirmation back to the queue after it has completed its work. In the case that the orchestrator fails to process the work, it will simply be picked up by the next orchestrator instance created.

Termination and Shutdown

A function is terminated once it has completed its work or it reaches the end of its allocated lifespan, generally in the range of 5–10 minutes. The function instance is suspended and enters a state of hibernation, where it may be immediately revived. The suspended function may also eventually be evicted from the hibernation cache due to resource or time constraints.

You will need to decide how to handle any open connections and resource assignments made to a function prior to its termination. In the case of a consumer client, the function instance may be assigned specific event stream partitions. Terminating the function instance without revoking these assignments may result in processing delays, as ownership in the consumer group won't be reassigned until a timeout is reached. Events from those partitions won't be processed until a consumer group rebalance is performed, or the terminated function instance comes back online and resumes processing.

If your function is almost always online and processing events, closing connections and rebalancing the consumer group may not be necessary. The function will likely only be momentarily suspended at the end of it lifespan, go briefly into hibernation, and be restored to runtime immediately. Conversely, for a consumer function that runs only intermittently, it is best to close down all connections and relinquish assignment of the event stream partitions. The next function instance will have to recreate the connection regardless of whether it's a warm start or cold start. When in doubt, cleaning up connections is generally a good idea; it lightens the load on external data stores and on the event broker and reduces the chances that suspended functions are laying claim to partition ownership.

Tuning Your Functions

Each function has specific needs based on its workload. Optimizing the resources that a function uses during its execution can ensure that performance remains high while costs remain low. There are a few things to consider when establishing the resources and tuning the parameters of your functions.

Allocating Sufficient Resources

Each function can be allocated a specific amount of CPU and memory. It is important to tune these parameters to the needs of your function; overallocation can be expensive, while underallocation may result in your functions crashing or taking too long to complete.

Maximum execution time is another factor, as it limits how long a function may run. This parameter is closely related to the batch size, as the time a function needs to process events is very often linearly related, on average, with the number of events to process. Set the maximum execution time higher than the maximum expected time to process a specific batch size of events to avoid unnecessary function timeout errors.

Lastly, you must consider any external I/O to state stores belonging within the bounded context of the function-based solution. The workload of a function varies with the flow of input events, with some workloads requiring consistent I/O to external state, and other workloads requiring only sporadic I/O. A failure to provide sufficient I/O resources can result in degraded throughput and performance.

Batch Event-Processing Parameters

If a function is unable to process its assigned batch of events during its execution lifespan, the execution of the function is considered to have failed and the batch must be processed again. However, barring any changes to the function's allocated execution time or the batch size of input events, it is likely to simply fail again. Therefore, one of two things must occur:

- Increase the maximum execution time of the function.
- Decrease the maximum batch size of events processed by the function.

Functions that establish their own connections to the event broker and manage the consumption of events can also periodically commit offsets during execution, ensuring partial completion of the batch. This does not work when the function is passed the batch of consumed events, as it has no way to update the consumer offsets during processing.

Additionally, some event-listener triggering systems, such as those provided by Amazon and Microsoft, give you the option to automatically halve the batch size on failure and re-execute the failed function. Subsequent errors result in the input batch being halved again and the function re-executed, until it reaches the point where it can complete its processing on time.

Scaling Your FaaS Solutions

FaaS solutions provide exceptional capabilities for the parallelization of work, especially for queues and event streams where the order in which data is processed is not important. For partitioned event streams, if the order of events is indeed important,

the maximum level of parallelization is limited by the number of partitions in your event streams, just as it is for all microservice implementations.

Scaling policies are typically the domain of the FaaS framework, so check your framework documentation to see what options are offered. Typical options involve scaling based on consumer input lag, time of day, processing throughput, and performance characteristics.

For functions that instantiate and manage their own event broker connections, beware the impact of partition assignment rebalancing when a consumer enters or leaves the consumer group. A consumer group can end up in a state of near-constant rebalancing if consumers are frequently joining and leaving the consumer group, preventing progress from being made. In extreme circumstances it is possible to get stuck in a virtual deadlock of rebalancing, where the functions spend their life cycle repeatedly having partitions assigned and removed. This problem can occur when you use many short-lived functions with small consumer batch sizes and can be made worse by scaling policies that are overly sensitive to delay. Instituting a step-based scaling policy or using a hysteresis loop can provide sufficient scaling responsiveness without putting the consumer group into a state of excessive rebalancing.

Static partition assignments eliminate the rebalancing overhead of dynamically assigned consumer groups and can also be used to copartition event streams. Functions will start with the foreknowledge of which partitions they will consume from; there will be no rebalancing, and events can simply be consumed whenever the function is triggered. This approach does require a more careful consideration of the work that your function is performing, as you need to ensure that each partition is being consumed.

Be careful about thrashing triggers and scaling policy. Frequent rebalancing of partition assignments can be expensive for event brokers. Try to scale your function count up or down at most once every few minutes.

Summary

Function-as-a-Service is an area of cloud computing that is growing rapidly. Many FaaS frameworks offer a variety of function development, management, deployment, triggering, testing, and scalability tools that allow you to build your microservices using functions. Functions can be triggered by new events in an event stream, consumer group lag status, wall-clock time, or custom logic.

Function-based solutions are particularly useful in handling stateless and simple stateful business problems that do not require event scheduling. The orchestration pattern allows multiple functions to be called in strict sequential order, while also

respecting the order of events from the event stream. Since the FaaS framework space is growing and evolving rapidly, it's important to keep up with the newest features of the platforms of interest to you and your organization.