

Alibaba Cloud

FunctionFlow Best Practices

Document Version: 20220117

Legal disclaimer

Alibaba Cloud reminds you to carefully read and fully understand the terms and conditions of this legal disclaimer before you read or use this document. If you have read or used this document, it shall be deemed as your total acceptance of this legal disclaimer.

1. You shall download and obtain this document from the Alibaba Cloud website or other Alibaba Cloud-authorized channels, and use this document for your own legal business activities only. The content of this document is considered confidential information of Alibaba Cloud. You shall strictly abide by the confidentiality obligations. No part of this document shall be disclosed or provided to any third party for use without the prior written consent of Alibaba Cloud.
2. No part of this document shall be excerpted, translated, reproduced, transmitted, or disseminated by any organization, company or individual in any form or by any means without the prior written consent of Alibaba Cloud.
3. The content of this document may be changed because of product version upgrade, adjustment, or other reasons. Alibaba Cloud reserves the right to modify the content of this document without notice and an updated version of this document will be released through Alibaba Cloud-authorized channels from time to time. You should pay attention to the version changes of this document as they occur and download and obtain the most up-to-date version of this document from Alibaba Cloud-authorized channels.
4. This document serves only as a reference guide for your use of Alibaba Cloud products and services. Alibaba Cloud provides this document based on the "status quo", "being defective", and "existing functions" of its products and services. Alibaba Cloud makes every effort to provide relevant operational guidance based on existing technologies. However, Alibaba Cloud hereby makes a clear statement that it in no way guarantees the accuracy, integrity, applicability, and reliability of the content of this document, either explicitly or implicitly. Alibaba Cloud shall not take legal responsibility for any errors or lost profits incurred by any organization, company, or individual arising from download, use, or trust in this document. Alibaba Cloud shall not, under any circumstances, take responsibility for any indirect, consequential, punitive, contingent, special, or punitive damages, including lost profits arising from the use or trust in this document (even if Alibaba Cloud has been notified of the possibility of such a loss).
5. By law, all the contents in Alibaba Cloud documents, including but not limited to pictures, architecture design, page layout, and text description, are intellectual property of Alibaba Cloud and/or its affiliates. This intellectual property includes, but is not limited to, trademark rights, patent rights, copyrights, and trade secrets. No part of this document shall be used, modified, reproduced, publicly transmitted, changed, disseminated, distributed, or published without the prior written consent of Alibaba Cloud and/or its affiliates. The names owned by Alibaba Cloud shall not be used, published, or reproduced for marketing, advertising, promotion, or other purposes without the prior written consent of Alibaba Cloud. The names owned by Alibaba Cloud include, but are not limited to, "Alibaba Cloud", "Aliyun", "HiChina", and other brands of Alibaba Cloud and/or its affiliates, which appear separately or in combination, as well as the auxiliary signs and patterns of the preceding brands, or anything similar to the company names, trade names, trademarks, product or service names, domain names, patterns, logos, marks, signs, or special descriptions that third parties identify as Alibaba Cloud and/or its affiliates.
6. Please directly contact Alibaba Cloud for any errors of this document.

Document conventions









Style	Description	Example
 Danger	A danger notice indicates a situation that will cause major system changes, faults, physical injuries, and other adverse results.	 Danger: Resetting will result in the loss of user configuration data.
 Warning	A warning notice indicates a situation that may cause major system changes, faults, physical injuries, and other adverse results.	 Warning: Restarting will cause business interruption. About 10 minutes are required to restart an instance.
 Notice	A caution notice indicates warning information, supplementary instructions, and other content that the user must understand.	 Notice: If the weight is set to 0, the server no longer receives new requests.
 Note	A note indicates supplemental instructions, best practices, tips, and other content.	 Note: You can use Ctrl + A to select all files.
>	Closing angle brackets are used to indicate a multi-level menu cascade.	Click Settings > Network > Set network type .
Bold	Bold formatting is used for buttons, menus, page names, and other UI elements.	Click OK .
<code>Courier font</code>	Courier font is used for commands	Run the <code>cd /d C:/window</code> command to enter the Windows system folder.
<i>Italic</i>	Italic formatting is used for parameters and variables.	<code>bae log list --instanceid</code> <i>Instance_ID</i>
[] or [a b]	This format is used for an optional value, where only one item can be selected.	<code>ipconfig [-all -t]</code>
{ } or {a b}	This format is used for a required value, where only one item can be selected.	<code>switch {active stand}</code>

Table of Contents

1.Poll for task status -----	05
2.Reliablely process distributed multi-step transactions -----	09
3.Integrate MNS topics to publish messages -----	16
4.Integrate MNS and use callback to orchestrate any type of tas...-----	21
5.Perform callbacks on asynchronous tasks -----	27
6.Schedule reserved resource functions or functions with specifie...-----	31
7.Create time-based schedules for a flow -----	33
8.Troubleshooting -----	36

1. Poll for task status

This topic describes how to poll for task status in Serverless workflow.

Overview

If no callback is specified after a long-running task is completed, developers usually poll the task status to check whether the task is completed. Reliable polling requires task status to be persistent.

Therefore, even if the current polling process fails, the polling continues after the process resumes. In this example, assume that a user calls Function Compute to submit a multimedia processing task that takes one minute to several hours. The task execution status can be queried by calling the corresponding API. This topic describes how to implement a common and reliable flow for polling task status in Serverless workflow.

Implementation in Serverless workflow

The following tutorial shows how to orchestrate two functions deployed in Function Compute as a flow for polling task status in the following three steps:

1. [Create a function in Function Compute](#)
2. [Create a flow in Serverless workflow](#)
3. [Execute the flow and view the result](#)

Step 1: Create a function in Function Compute

1. Create a service named `fnf-demo` in Function Compute, and create two functions (StartJob and GetJobStatus) in Python 2.7 in this service. For more information, see [Create a function in the Function Compute console](#).
 - The StartJob function is used to simulate calling an API to start a long-running task and return the task ID.

```
import logging
import uuid
def handler(event, context):
    logger = logging.getLogger()
    id = uuid.uuid4()
    logger.info('Started job with ID %s' % id)
    return {"job_id": str(id)}
```

- The GetJobStatus function is used to simulate calling an API to query the execution result of the specified task. It compares the value of the current time minus the time when the function is first executed with the value specified in `delay` and then returns the task status "success" or "running" accordingly.

```
import logging
import uuid
import time
import json
start_time = int(time.time())
def handler(event, context):
    evt = json.loads(event)
    logger = logging.getLogger()
    job_id = evt["job_id"]
    logger.info('Started job with ID %s' % job_id)
    now = int(time.time())
    status = "running"
    delay = 60
    if "delay" in evt:
        delay = evt["delay"]
    if now - start_time > delay:
        status = "success"
    try_count = 0
    if "try_count" in evt:
        try_count = evt["try_count"]
    try_count = try_count + 1
    logger.info('Job %s, status %s, try_count %d' % (job_id, status, try_count))
    return {"job_id": job_id, "job_status": status, "try_count": try_count}
```

Step 2: Create a flow in Serverless workflow

The following steps show the main logic of this flow:

1. StartJob: Serverless Workflow calls the `StartJob` function to start a task.
2. Wait10s: Serverless Workflow waits for 10s.
3. GetJobStatus: Serverless Workflow calls the `GetJobStatus` function to query the status of the current task.
4. CheckJobComplete: Serverless Workflow checks the result returned by the `GetJobStatus` function.
 - The result "success" indicates that the flow is completed.
 - If the polling requests are sent three or more times, Serverless Workflow considers that the task fails, and then the flow fails.
 - If the result "running" is returned, the system goes back to the `Wait10s` step.

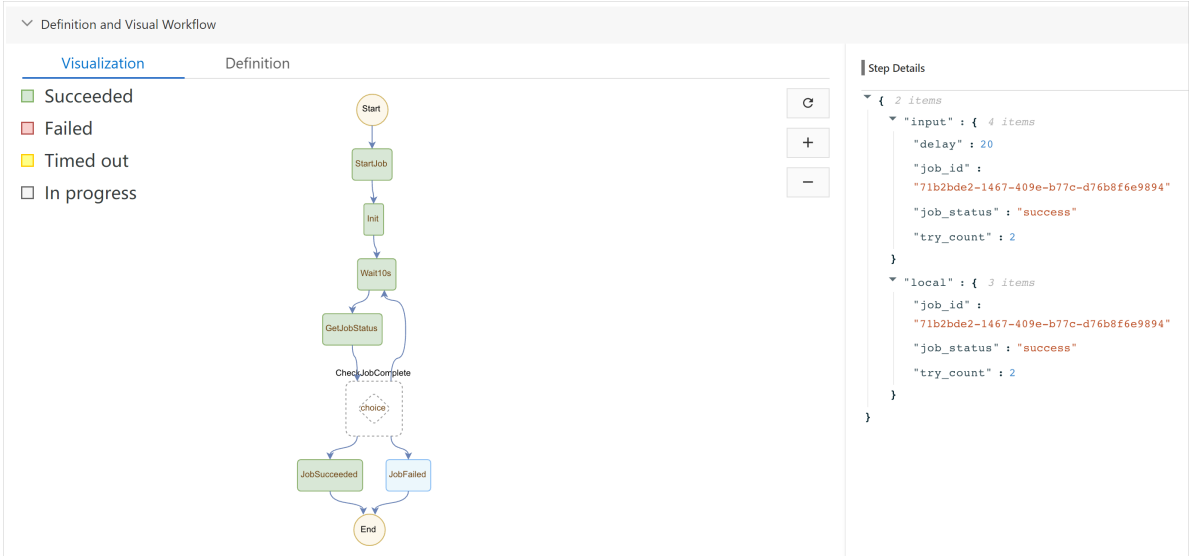
```
version: v1
type: flow
steps:
  - type: task
    name: StartJob
    resourceArn: acs:fc:cn-hangzhou:{accountID}:services/fnf-demo/functions/StartJob
  - type: pass
    name: Init
    outputMappings:
      - target: try_count
        source: 0
  - type: wait
    name: Wait10s
    duration: 10
  - type: task
    name: GetJobStatus
    resourceArn: acs:fc:cn-hangzhou:{accountID}:services/fnf-demo/functions/GetJobStatus
  - type: choice
    name: CheckJobComplete
    inputMappings:
      - target: job_id
        source: $local.job_id
      - target: delay
        source: $input.delay
      - target: try_count
        source: $local.try_count
    choices:
      - condition: $.status == "success"
        goto: JobSucceeded
      - condition: $.try_count > 3
        goto: JobFailed
      - condition: $.status == "running"
        goto: Wait10s
  - type: succeed
    name: JobSucceeded
  - type: fail
    name: JobFailed
```

Step 3: Execute the flow and view the result

In the Serverless Workflow console, find the target flow, click **Start Execution**, and then enter the following JSON object as the input of this execution. The value of `delay` indicates the time that the task takes to run. In this example, it is set to 20, which means that the `GetJobStatus` function returns "success" 20s later after the task is started, before which "running" is returned. You can change the value of `delay` to view different execution results.

```
{
  "delay": 20
}
```

- The following figure is the visual display of the polling flow from start to end.



- As shown in the following figure, the task takes 20s to run. When the `GetJobStatus` function is called for the first time, "running" is returned. Therefore, when `CheckJobComplete` is called, the system proceeds to the `Wait10s` step to wait 10s before the next query is initiated. The "success" result is returned for the second query, and the flow ends.

Execution History		Input/Output			
ID		Type	Step	Timestamp	Relative time (ms)
+	18	TaskSucceeded	GetJobStatus	Aug 14, 2020, 12:28:58	29046
-	19	StepExited	GetJobStatus	Aug 14, 2020, 12:28:59	29902
<pre>{ "local": { "job_id": "71b2bde2-1467-409e-b77c-d76b8f6e9894" "job_status": "running" "try_count": 1 } }</pre>					
+	20	StepEntered	CheckJobComplete	Aug 14, 2020, 12:29:00	30906
+	21	StepStarted	CheckJobComplete	Aug 14, 2020, 12:29:01	31911
+	22	StepSucceeded	CheckJobComplete	Aug 14, 2020, 12:29:02	32917
+	23	StepExited	CheckJobComplete	Aug 14, 2020, 12:29:03	33922
+	24	StepEntered	Wait10s	Aug 14, 2020, 12:29:04	34927
+	25	StepStarted	Wait10s	Aug 14, 2020, 12:29:05	35932

2. Reliably process distributed multi-step transactions

This topic describes how to use Serverless workflow to guarantee that distributed transactions are reliably processed in a complex flow, helping you focus on your business logic.

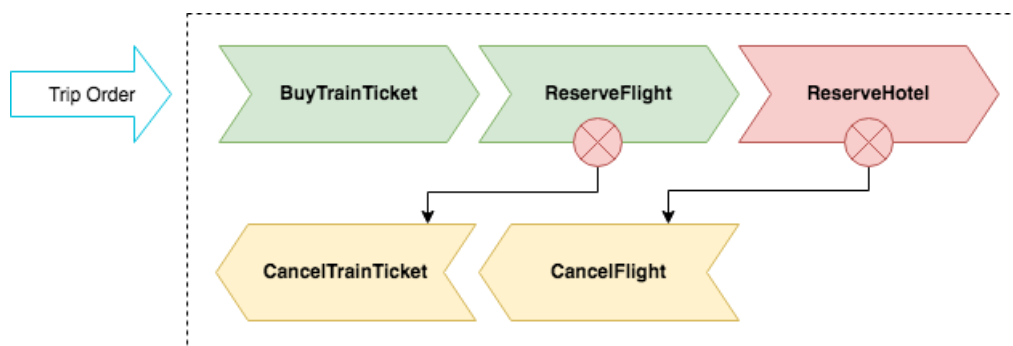
Overview

In complex scenarios involving order management, such as e-commerce websites, hotel booking, and flight reservations, applications need to access multiple remote services, and have high requirements for the operational semantics of transactions. In other words, all steps must succeed or fail without intermediate states. In applications with small traffic and centralized data storage, the atomicity, consistency, isolation, durability (ACID) properties of relational databases can guarantee that transactions are reliably processed. However, in large-traffic scenarios, distributed microservices are usually used for high availability and scalability. To guarantee reliable processing of multi-step transactions, the service providers usually need to introduce queues and persistent messages and display the flow status to the distributed architecture. This brings additional development and O&M costs. To resolve the preceding problems, Serverless workflow provides guarantee on reliable processing of distributed transactions in complex flows.

Scenarios

Assume that an application provides the train ticket, flight, and hotel booking feature and ensures that the transactions are reliably processed in three steps. Three remote calls are required to implement this feature (for example, you must call the 12306 API to book a train ticket). If all the three calls are successful, the order is successful. However, any of the three remote calls may fail. Therefore, the application must have compensation logic for different failure scenarios to roll back completed operations. The following figure shows the details.

- If BuyTrainTicket is successful but ReserveFlight fails, the application calls CancelTrainTicket and notifies the user that the order failed.
- If both BuyTrainTicket and ReserveFlight are successful but ReserveHotel fails, the application calls CancelFlight and CancelTrainTicket and notifies the user that the order failed.



Implementation in Serverless workflow

In the following example, a function deployed in Function Compute is orchestrated into a flow in Serverless workflow to implement a reliable multi-step complex flow in three steps:

1. Create a function in Function Compute.
2. Create a flow.

3. Execute the flow and view the result.

Step 1: Create a function in Function Compute to simulate the BuyTrainTicket, ReserveFlight, and ReserveHotel operations

1. Create a function in Python 2.7. For more information, see [Create a function in the Function Compute console](#). We recommend that you name the service and function in Function Compute to the following names respectively:

- Service: fnf-demo
- Function: Operation

The Operation function simulates the operations such as ReserveFlight, and ReserveHotel. The Operation result (success or failure) is determined by the input.

```
import json
import logging
import uuid

def handler(event, context):
    evt = json.loads(event)
    logger = logging.getLogger()
    id = uuid.uuid4()
    op = "operation"
    if 'operation' in evt:
        op = evt['operation']
    if op in evt:
        result = evt[op]
        if result == False:
            logger.info("%s failed" % op)
            exit()
    logger.info("%s succeeded, id %s" % (op, id))
    return '{"%s": "success", "%s_txnID": "%s"}' % (op, op, id)
```

Step 2: Create a flow

In the [Serverless workflow console](#), perform the following steps to create a flow:

1. Configure a **Resource Access Management (RAM)** user for the flow.

```
{
  "Statement": [
    {
      "Action": "sts:AssumeRole",
      "Effect": "Allow",
      "Principal": {
        "Service": [
          "fnf.aliyuncs.com"
        ]
      }
    }
  ],
  "Version": "1"
}
```

2. Define the flow.

```

version: v1
type: flow
steps:
  - type: task
    resourceArn: acs:fc:{region}:{accountID}:services/fnf-demo/functions/Operation
    name: BuyTrainTicket
    inputMappings:
      - target: operation
        source: buy_train_ticket
      - target: buy_train_ticket
        source: $input.buy_train_ticket_result
    catch:
      - errors:
          - FC.Unknown
            goto: OrderFailed
  - type: task
    resourceArn: acs:fc:{region}:{accountID}:services/fnf-demo/functions/Operation
    name: ReserveFlight
    inputMappings:
      - target: operation
        source: reserve_flight
      - target: reserve_flight
        source: $input.reserve_flight_result
    catch: # When the FC.Unknown error thrown by the ReserveFlight task is captured, S
erverless Workflow jumps to the CancelTrainTicket task.
      - errors:
          - FC.Unknown
            goto: CancelTrainTicket
  - type: task
    resourceArn: acs:fc:{region}:{accountID}:services/fnf-demo/functions/Operation
    name: ReserveHotel
    inputMappings:
      - target: operation
        source: reserve_hotel
      - target: reserve_hotel
        source: $input.reserve_hotel_result
    retry: # Serverless Workflow retries the task step up to three times in the expone
ntial backoff mode upon an FC.Unknown error. The initial retry interval is 1s, and the
next retry interval is twice the previous retry interval for the rest of the retries.
      - errors:
          - FC.Unknown
            intervalSeconds: 1
            maxAttempts: 3
            multiplier: 2
    catch: # When the FC.Unknown error thrown by the ReserveHotel task is captured, Se
rverless Workflow jumps to the CancelFlight task.
      - errors:
          - FC.Unknown
            goto: CancelFlight
  - type: succeed
    name: OrderSucceeded
  - type: task
    resourceArn: acs:fc:{region}:{accountID}:services/fnf-demo/functions/Operation
    name: CancelFlight

```

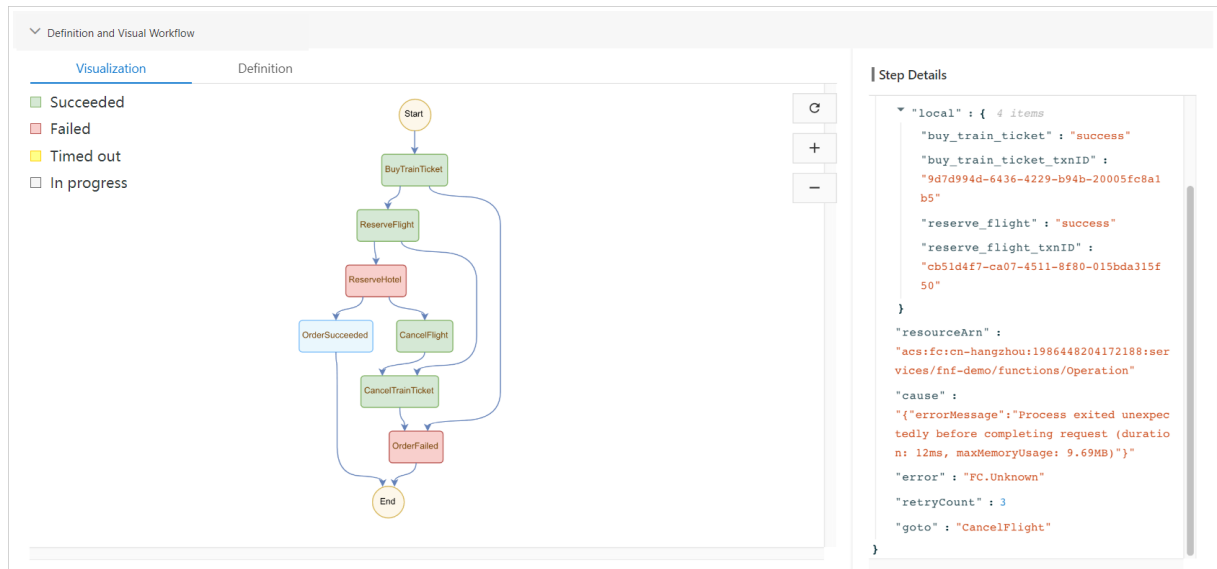
```
inputMappings:
  - target: operation
    source: cancel_flight
  - target: reserve_flight_txnID
    source: $local.reserve_flight_txnID
- type: task
  resourceArn: acs:fc:{region}:{accountID}:services/fnf-demo/functions/Operation
  name: CancelTrainTicket
  inputMappings:
    - target: operation
      source: cancel_train_ticket
    - target: reserve_flight_txnID
      source: $local.reserve_flight_txnID
- type: fail
  name: OrderFailed
```

Step 3: Execute the flow and view the result

Execute the flow you created in the console. The inputs for the StartExecution operation must be in JSON format. The following JSON objects can simulate the success or failure of each step. For example, "reserve_hotel_result": "fail" indicates a failure to reserve a hotel. StartExecution is an asynchronous operation. After the operation is called, Serverless workflow returns an execution name for you to query the flow execution status.

```
{
  "buy_train_ticket_result": "success",
  "reserve_flight_result": "success",
  "reserve_hotel_result": "fail"
}
```

After the flow execution starts, in the Serverless workflow console, click the target execution name. On the page that appears, view the execution process and results in the Definition and Visual Workflow section. As shown in the following figure, due to "reserve_hotel_result": "fail", ReserveHotel fails, and Serverless workflow calls CancelFlight and CancelTrainTicket in sequence based on the flow definition. In Serverless workflow, each step is persistent. In this way, failures such as network interruption or unexpected process exits do not affect the transactions in the flow.



An execution event is generated for each flow execution. You can call the `GetExecutionHistory` operation to query the execution events in the console or by using the SDK or command-line interface (CLI).

Execution History		Input/Output		
ID	Type	Step	Timestamp	Relative time (ms)
+	1	ExecutionStarted	Aug 14, 2020, 19:57:17	0
+	2	StepEntered	BuyTrainTicket	23
+	3	TaskStarted	BuyTrainTicket	31
+	4	TaskSucceeded	BuyTrainTicket	40
+	5	StepExited	BuyTrainTicket	48
+	6	StepExited	BuyTrainTicket	57
+	7	StepEntered	ReserveFlight	65
+	8	TaskScheduled	ReserveFlight	73
+	9	TaskStarted	ReserveFlight	73
+	10	TaskSucceeded	ReserveFlight	73
+	11	StepExited	ReserveFlight	73
+	12	StepEntered	ReserveFlight	82

Error handling and retries

1. In the preceding example, remote calls of `ReserveFlight` and `ReserveHotel` fail due to network or service errors. Retry upon transient errors can improve the success rate of the ordering flow. Serverless workflow automatically retries task steps. For example, define the `ReserveHotel` step based on the following code to retry the step in exponential backoff mode after the `FC.Unknown` is captured. If `ReserveHotel` still fails after the maximum number of retries, based on the `catch` definition of the step, Serverless Workflow captures the `FC.Unknown` error thrown by the `ReserveHotel` function and then jumps to the `CancelFlight` operation and implements the defined compensation logic.

```

- type: task
  resourceArn: acs:fc:{region}:{accountID}:services/fnf-demo/functions/Operation
  name: ReserveHotel
  inputMappings:
  - target: operation
    source: reserve_hotel
  retry: # Serverless Workflow retries the task step up to three times in the exponential backoff mode upon an FC.Unknown error. The initial retry interval is 1s, and the next retry interval is twice the previous retry interval for the rest of the retries.
  - errors:
    - FC.Unknown
    intervalSeconds: 1
    maxAttempts: 3
    multiplier: 2
  catch: # When the FC.Unknown error thrown by the ReserveHotel task is captured, Serverless Workflow jumps to the CancelFlight task.
  - errors:
    - FC.Unknown
    goto: CancelFlight

```

2. The following figure shows that, after the retry parameter is defined, the ReserveHotel task step is retried the specified maximum number of times.

Execution History		Input/Output			
ID	Type	Step	Timestamp	Relative time (ms)	
+ 12	StepEntered	ReserveHotel	Aug 16, 2020, 12:15:34	88	
+ 13	TaskScheduled	ReserveHotel	Aug 16, 2020, 12:15:34	98	
+ 14	TaskStarted	ReserveHotel	Aug 16, 2020, 12:15:34	102	
+ 15	TaskFailed	ReserveHotel	Aug 16, 2020, 12:15:34	108	
+ 16	TaskScheduled	ReserveHotel	Aug 16, 2020, 12:15:34	113	
+ 17	TaskStarted	ReserveHotel	Aug 16, 2020, 12:15:34	118	
+ 18	TaskFailed	ReserveHotel	Aug 16, 2020, 12:15:34	123	
+ 19	TaskScheduled	ReserveHotel	Aug 16, 2020, 12:15:34	127	
+ 20	TaskStarted	ReserveHotel	Aug 16, 2020, 12:15:34	131	
+ 21	TaskFailed	ReserveHotel	Aug 16, 2020, 12:15:34	138	
+ 22	TaskScheduled	ReserveHotel	Aug 16, 2020, 12:15:34	145	
+ 23	TaskStarted	ReserveHotel	Aug 16, 2020, 12:15:34	152	

Data transfer between steps

1. After ReserveHotel fails, CancelFlight and CancelTrainTicket are called. To cancel these two tasks, the transaction IDs (txnID) returned by ReserveFlight and BuyTrainTicket are required. The following section describes how to use the `inputMapping` object to pass the outputs of the previous steps to the `CancelFlight` step.

```

- type: task
  resourceArn: acs:fc:{region}:{accountID}:services/fnf-demo/functions/Operation
  name: CancelFlight
  inputMappings:
  - target: operation
    source: cancel_flight
  - target: reserve_flight_txnID
    source: $local.reserve_flight_txnID

```

2. Outputs of each step of the flow are stored in the local object of EventDetail in the `StepExited` event.

```

{
  "input":{
    "operation":"reserve_hotel",
    "reserve_hotel_result":"fail"
  },
  "local":{
    "buy_train_ticket":"success",
    "buy_train_ticket_txnID":"d37412b3-bb68-4d04-9d90-c8c15643d45e",
    "reserve_flight_result":"success",
    "reserve_flight_txnID":"024caecf-cfa3-43a6-b561-9b6fe0571b55"
  },
  "resourceArn":"acs:fc:{region}:{accountID}:services/fnf-demo/functions/Operation",
  "cause":{"errorMessage":"Process exited unexpectedly before completing request (duration: 12ms, maxMemoryUsage: 9.18MB)"},"",
  "error":"FC.Unknown",
  "retryCount":3,
  "goto":"CancelFlight"
}

```

3. Based on `EventDetail` and `inputMappings`, the inputs of the `CancelFlight` step are converted into the following JSON object. In this way, the inputs of the `CancelFlight` function contain the `reserve_flight_txnID` field.

```

"input":{
  "operation":"cancel_flight",
  "reserve_flight_txnID":"024caecf-cfa3-43a6-b561-9b6fe0571b55"
}

```

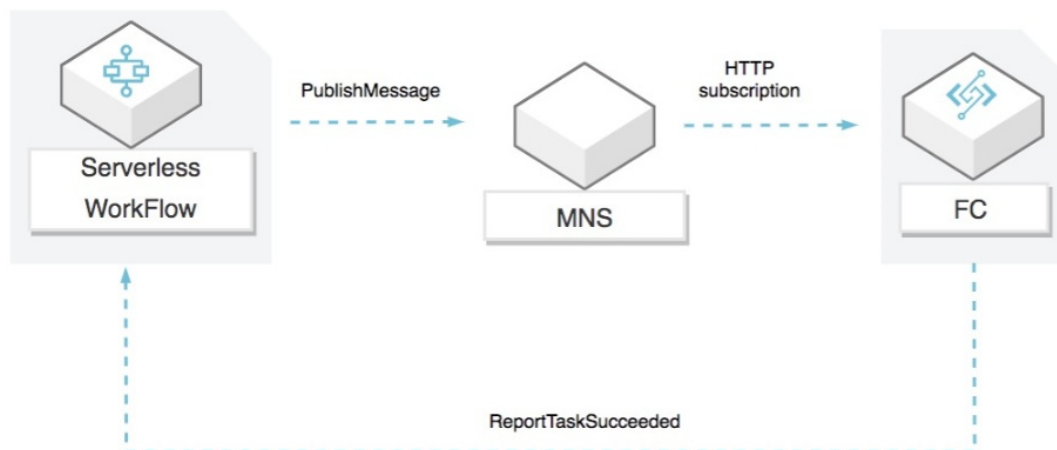
3.Integrate MNS topics to publish messages

This topic describes how to integrate a topic of Message Service (MNS) in the wait-for-callback mode of a task step and publish messages to the topic. After the MNS topic receives a message, the ReportTaskSucceeded or ReportTaskFailed operation is called to call back the task status.

How it works

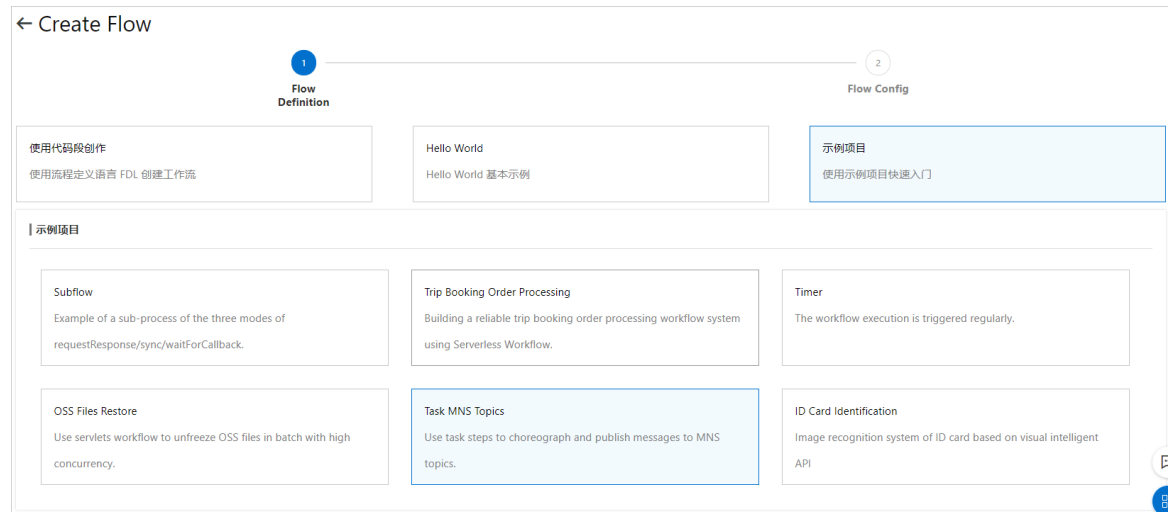
After an application is deployed, the application is executed based on the following steps:

1. Execute the flow. The task step publishes a message to the MNS topic. The `TaskToken` of the task step is placed in the message body and sent to the topic.
2. The task step of the flow is suspended and waits for the task callback.
3. After the MNS topic receives the message, the message and the `TaskToken` are pushed to the HTTP trigger of the function in Function Compute over HTTP to trigger the execution.
4. The function in Function Compute obtains the `TaskToken` and calls `ReportTaskSucceeded` to report the task status.
5. Then, the flow continues.

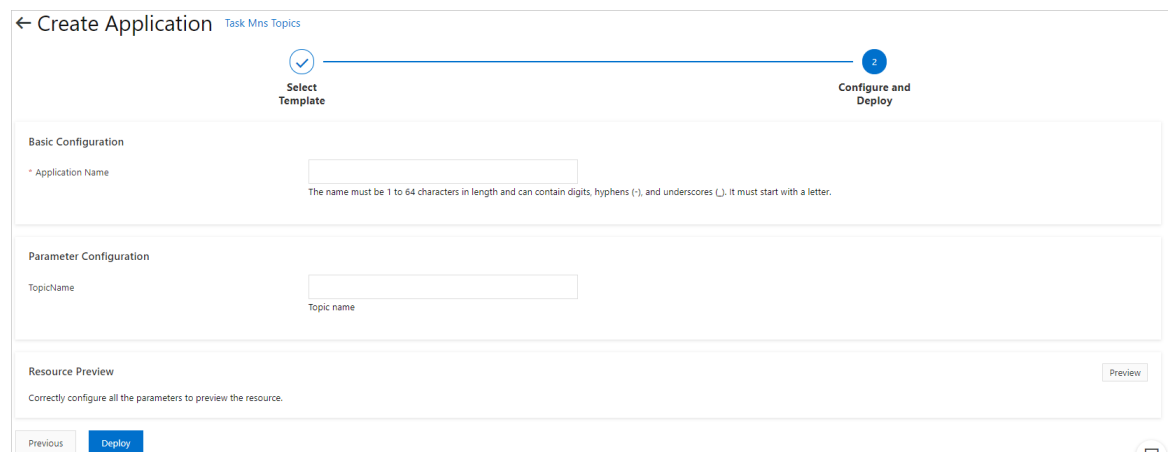


Deploy an application

1. In the [Serverless workflow console](#), click Create Flow. On the page that appears, select **Sample Project** and **Task MNS Topics**, and then click **Next Step**.



2. On the **Create Application** page, create an application corresponding to the template, and then click **Deploy**.



Where:

Application Name: Enter a name for the application. The name must be unique in the same account.

TopicName: Enter a name for the topic. If the specified MNS topic does not exist, the system automatically creates it.

After you click **Deploy**, all resources that you created in the application are displayed.

← fnd-name

Overview

Deploy

Monitoring

Logical Resource Name	Physical Resource Name	Resource Type	Resource Status	Updated At
		Role	Creating	Dec 10, 2020, 17:47:38
lowRole		ToRole	Initialized	Dec 10, 2020, 17:47:38
Role		Role	Initialized	Dec 10, 2020, 17:47:38
		Service	Initialized	Dec 10, 2020, 17:47:38
		Function	Initialized	Dec 10, 2020, 17:47:38
Flow		Flow	Initialized	Dec 10, 2020, 17:47:38
viceRole		ToRole	Initialized	Dec 10, 2020, 17:47:38
		Topic	Creating	Dec 10, 2020, 17:47:38
viceRole		ToRole	Initialized	Dec 10, 2020, 17:47:38
ackhttp		Trigger	Initialized	Dec 10, 2020, 17:47:38
ion		Description	Initialized	Dec 10, 2020, 17:47:38

3. Execute the flow.

Input of the execution

```
{
  "messageBody": "hello world"
}
```

Details

Name: a9645443-c24f-bd97-941d-fbef496b512f

Started Time: Aug 16, 2020, 13:56:19

Status: Succeeded

End Time: Aug 16, 2020, 13:56:22

Definition and Visual Workflow

Visualization

Definition

Succeeded

Failed

Timed out

In progress

Start

mns-topic-task

End

Step Details

Select a step to view the details

Application code

1. Orchestrate a flow of the MNS topic.

Encapsulate the `TaskToken` called back in the task step into `MessageBody` of the message for subsequent callback. Read `output` specified in `ReportTaskSucceeded` from `outputMappings`.

```
version: v1
type: flow
steps:
  - type: task
    name: mns-topic-task
    resourceArn: acs:mns::/topics/<topic>/messages
    pattern: waitForCallback
    inputMappings:
      - target: messageBody
        source: $input.messageBody
      - target: taskToken
        source: $context.task.token
    outputMappings:
      - target: status
        source: $local.status
    serviceParams:
      MessageBody: $
```

2. Call back the function of the task step that is deployed in Function Compute.

Read the `TaskToken` that is encapsulated in `MessageBody`, set the TaskToken callback status to set `output`, and then set TaskToken to `{"status": "success"}`.

```
def handler(enviro, start_response):
    # Get request body
    try:
        request_body_size = int(enviro.get('CONTENT_LENGTH',
0))
    except ValueError:
        request_body_size = 0
    request_body =
enviro['wsgi.input'].read(request_body_size)
    print('Request body:
{}'.format(request_body))
    body = json.loads(request_body)
    message_body_str =
body['Message']
    # Read MessageBody and TaskToken from
message body
    message_body =
json.loads(message_body_str)
    task_token =
message_body['taskToken']
    ori_message_body =
message_body['messageBody']
    print('Task token: {} \norigin message
body: {}'.format(task_token, ori_message_body))
    # Init fnf client use sts token
    context = enviro['fc.context']
    creds = context.credentials
    sts_creds =
StsTokenCredential(creds.access_key_id, creds.access_key_secret, creds.security_token)
    fnf_client =
AcsClient(credential=sts_creds, region_id=context.region)
    # Report task succeeded to serverless
workflow
    req =
ReportTaskSucceededRequest()
    req.set_TaskToken(task_token)
    req.set_Output('{"status":
"success"}')
    resp =
fnf_client.do_action_with_exception(req)
    print('Report task response:
{}'.format(resp))
    # Response to http request
    status = '200 OK'
    response_headers = [('Content-type',
'text/plain')]
    start_response(status,
response_headers)
    return [b'OK']
```

References

For more information about how to use task steps to orchestrate MNS topics, see [task-mns-topics](#).

4.Integrate MNS and use callback to orchestrate any type of tasks

Serverless workflow provides the service integration feature to simplify the interaction between users and cloud services. In this topic, the Message Service (MNS) queues are used with callback to orchestrate tasks that do not involve functions in Function Compute.

Overview

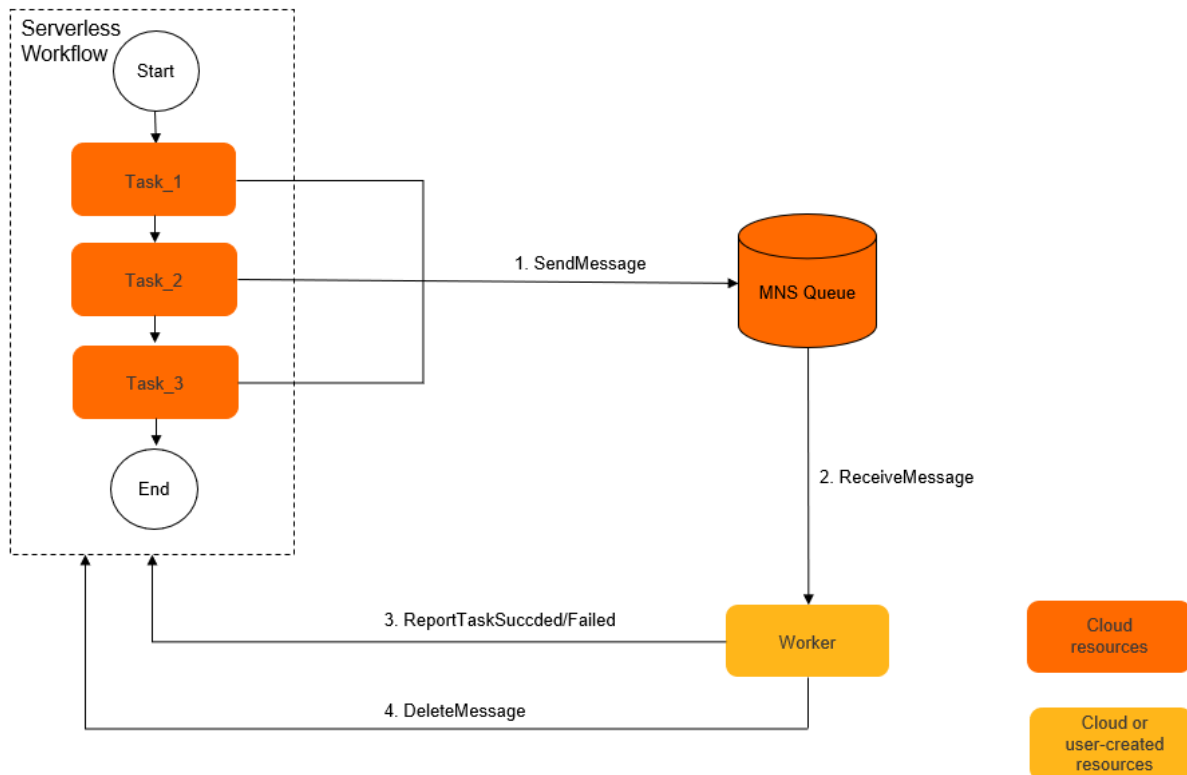
Serverless workflow not only allows you to orchestrate functions that are deployed in Function Compute in Function as a Service (FaaS) mode into flows, but also allows you to orchestrate other computing tasks into flows. The topic [Perform callbacks on asynchronous tasks](#) under Best Practices describes how to use functions in Function Compute to send messages to MNS queues. In custom environments, after a task executor (worker) receives a message, it notifies Serverless workflow of the task execution result based on the callback. This topic describes how to use MNS queues, a new feature of Serverless workflow. MNS queues further simplify the orchestration of custom task types. Serverless workflow allows you to directly send messages to MNS queues. In this way, you do not need to develop, test, and maintain the function that is deployed in Function Compute for sending the messages, improving the availability and reducing the latency. Compared with sending messages to MNS topics by using functions in Function Compute, using the integrated MNS service to send messages to specified MNS queues has the following benefits:

- You do not need to develop a function in Function Compute to send messages. This reduces the cost of development, testing, and maintenance.
- The message delivery delay is reduced, a remote access process is eliminated, and the cold start of Function Compute is avoided.
- Service dependency is removed and fault tolerance is improved.

Serverless workflow will support more cloud services in the future to make it easier to orchestrate flows that consist of different types of tasks.

Service integration

In the following figure, the three serial tasks are sent by Serverless workflow to the specified MNS queue in sequence. After the messages are sent, Serverless workflow waits for the callback in this step. You can call the `ReceiveMessage` operation of MNS to pull messages in the worker in a custom environment, such as an Elastic Compute Service (ECS) instance, a container, or a server in an on-premises data center. After the worker receives the messages, it executes the corresponding task based on the message content. After the task ends, the worker calls the `ReportTaskSucceeded/Failed` operation of Serverless workflow. Serverless workflow continues the step after receiving the task result. After the worker reports the success result, the message is deleted from the MNS queue.



Procedure

Perform the following step to use this feature:

1. Prepare for using this feature
2. Define a flow
3. Define a worker
4. Execute the flow and view the result

Step 1: Prepare for using this feature

1. In the [MNS console](#), create an MNS queue. For more information, see [Create a queue](#).
2. Serverless workflow assumes the [Create execution roles](#) (the role of the RAM user) that you specify in the flow to send messages to the MNS queue in your Alibaba Cloud account. Therefore, you must add **MNS SendMessage** policies for the role of the RAM user. The following example shows a fine-grained policy. If you do not need the fine-grained policy, you can log on to the [Serverless workflow console](#), and add `AliyunMNSFullAccess` in System Policy to Flow RAM Role.

```
{
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "mns:SendMessage"
      ],
      "Resource": [
        "acs:mns:$region:$account_id:/queues/$queue_name/messages"
      ]
    }
  ],
  "Version": "1"
}
```

Step 2: Define a flow

The following code in Flow Definition Language (FDL) defines a task step that can send messages to the MNS queue named fnf-demo and wait for the callback.

```
version: v1
type: flow
steps:
- type: task
  name: Task_1
  resourceArn: acs:mns:::/queues/fnf-demo/messages # This task step sends messages to the
MNS queue fnf-demo that is under the same account in the same region.
  pattern: waitForCallback # The task step suspends after the message is sent to the MNS
queue and waits until it receives the callback.
  inputMappings:
    - target: task_token
      source: $context.task.token # Serverless Workflow queries the task token from the
context object.
    - target: key
      source: value
  serviceParams: # The service integration parameters.
    MessageBody: $ # The mapped input is used as the body of the message you want to send.
    Priority: 1 # The priority of the MNS queue.
```

Step 3: Define a worker

The following Python 2.7 code simulates a worker that executes a task. It can run in any environment that can access Serverless workflow and MNS. The worker calls the MNS `ReceiveMessage` operation for long polling. When it enters a task step with an MNS configuration, Serverless workflow sends a message to the `fnf-demo` queue. After the worker executes the task, it calls back the `ReportTaskSucceeded/Failed` operation of Serverless workflow. After Serverless workflow receives the task execution result, it continues the current task step. The worker deletes the message from the queue.

1. In a virtual environment, install Serverless Workflow, MNS, and Python SDK.

```
cd /tmp; mkdir -p fnf-demo-callback; cd fnf-demo-callback
virtualenv env; source env/bin/activate
pip install -t . aliyun-python-sdk-core -t . aliyun-python-sdk-fnf -t . aliyun-mns
```

2. Compile the code for the local task executor worker.py.


```

import json
import os
from aliyunsdkcore.client import AcsClient
from aliyunsdkcore.acs_exception.exceptions import ServerException
from aliyunsdkcore.client import AcsClient
from aliyunsdkfnf.request.v20190315 import ReportTaskSucceededRequest
from mns.account import Account # pip install aliyun-mns
from mns.queue import *

def main():
    region = os.environ['REGION']
    account_id = os.environ['ACCOUNT_ID']
    ak_id = os.environ['AK_ID']
    ak_secret = os.environ['AK_SECRET']
    queue_name = "fnf-demo"
    fnf_client = AcsClient(
        ak_id,
        ak_secret,
        region
    )
    mns_endpoint = "https://%s.mns.%s.aliyuncs.com" % (account_id, region)
    my_account = Account(mns_endpoint, ak_id, ak_secret)
    my_queue = my_account.get_queue("fnf-demo")
    my_queue.set_encoding(False)
    wait_seconds = 10
    try:
        while True:
            try:
                print "Receiving messages"
                rcv_msg = my_queue.receive_message(wait_seconds)
                print "Received message %s, body %s" % (rcv_msg.message_id, rcv_msg.message_body)

                body = json.loads(rcv_msg.message_body)
                task_token = body["task_token"]
                output = "{\"key\": \"value\"}"
                request = ReportTaskSucceededRequest.ReportTaskSucceededRequest()
                request.set_Output(output)
                request.set_TaskToken(task_token)
                resp = fnf_client.do_action_with_exception(request)
                print "Report task succeeded finished"
                my_queue.delete_message(rcv_msg.receipt_handle)
                print "Deleted message " + rcv_msg.message_id
            except MNSEExceptionBase as e:
                print(e)
            except ServerException as e:
                print(e)
                if e.error_code == 'TaskAlreadyCompleted':
                    my_queue.delete_message(rcv_msg.receipt_handle)
                    print "Task already completed, deleted message " + rcv_msg.message_id
            except ServerException as e:
                print(e)
    if __name__ == '__main__':
        main()

```

3. Run the worker to long poll the fnf-demo queue. After the worker receives the message, it

performs callback to report the result to Serverless workflow.

```
# Run the worker process.
export REGION={your-region}
export ACCOUNT_ID={your-account-id}
export AK_ID={your-ak-id}
export AK_SECRET={your-ak-secret}
python worker.py
```

Step 4: Execute the flow and view the result

In the Serverless workflow console, execute the flow and run the worker. The result shows that the flow is successful.

The screenshot displays the 'Execute' page for a workflow named 'scheduled-exec-20200816T064912Z-613ade66-473f-4846-a1b9-6c1bdf999647'. The status is 'Succeeded' and the execution time is 'Aug 16, 2020, 14:49:12'. The workflow definition shows a simple flow: Start -> Task_1 -> End. The 'Step Details' for 'Task_1' show the input payload as a single item with a key 'value' and a trigger name 'schedule-test'.

← Execute scheduled-exec-20200816T064912Z-613ade66-473f-4846-a1b9-6c1bdf999647

Start Execution Monitoring and Alerts Share Edit Flow

Details

Name: scheduled-exec-20200816T064912Z-613ade66-473f-4846-a1b9-6c1bdf999647 Started Time: Aug 16, 2020, 14:49:12

Status: Succeeded End Time: Aug 16, 2020, 14:49:12

Definition and Visual Workflow

Visualization Definition

Succeeded Failed Timed out In progress

Start Task_1 End

Step Details

```
{
  "input": {
    "payload": {
      "key": "value"
    }
  },
  "triggerName": "schedule-test",
  "triggerTime": "2020-08-16T06:49:12Z"
}
```

5. Perform callbacks on asynchronous tasks

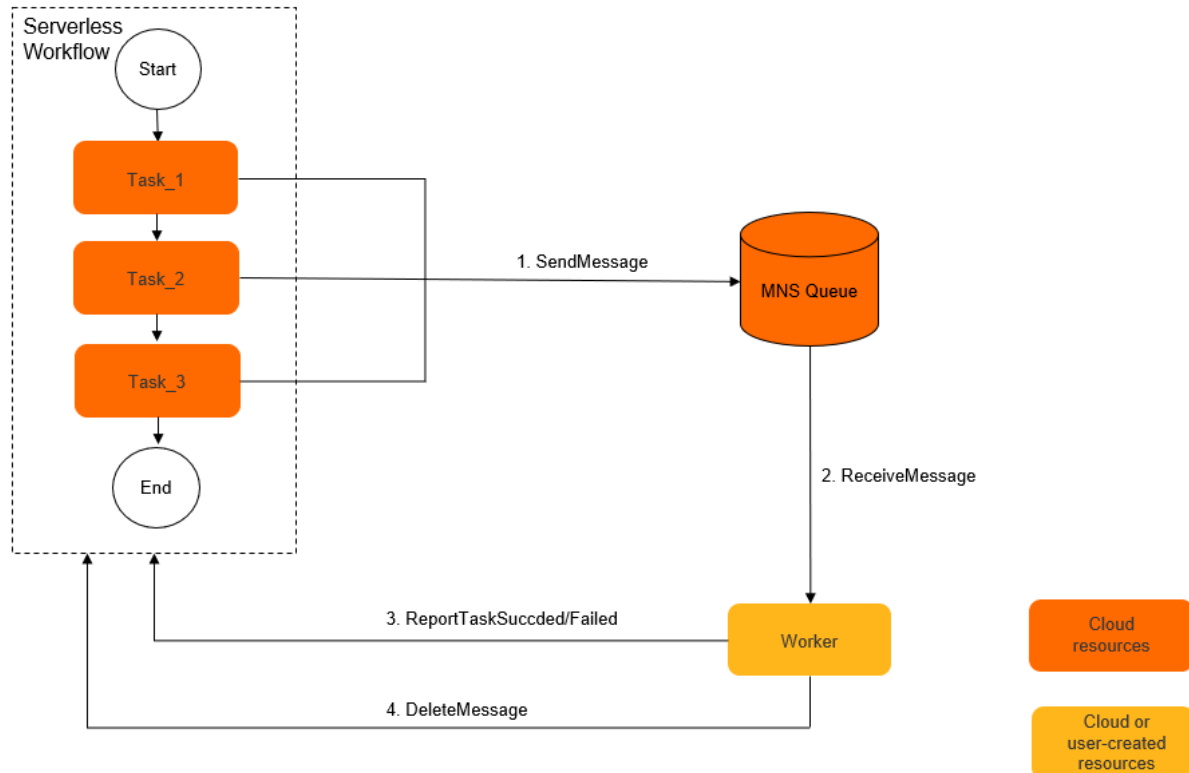
This topic describes the callback feature of Serverless workflow. Compared with polling, a callback effectively reduces the delay and unnecessary pressure on the server caused by polling. In addition, callback can be used with queues to orchestrate non-Function Compute tasks. In this way, Serverless workflow allows you to orchestrate any type of computing resources.

Overview

Long-running tasks are asynchronously submitted and a task ID is returned. You can use either polling or callback to check whether an asynchronous task ends. The [Poll for task status](#) topic describes how to use polling to check whether a task ends. The callback feature of Serverless workflow has the following benefits:

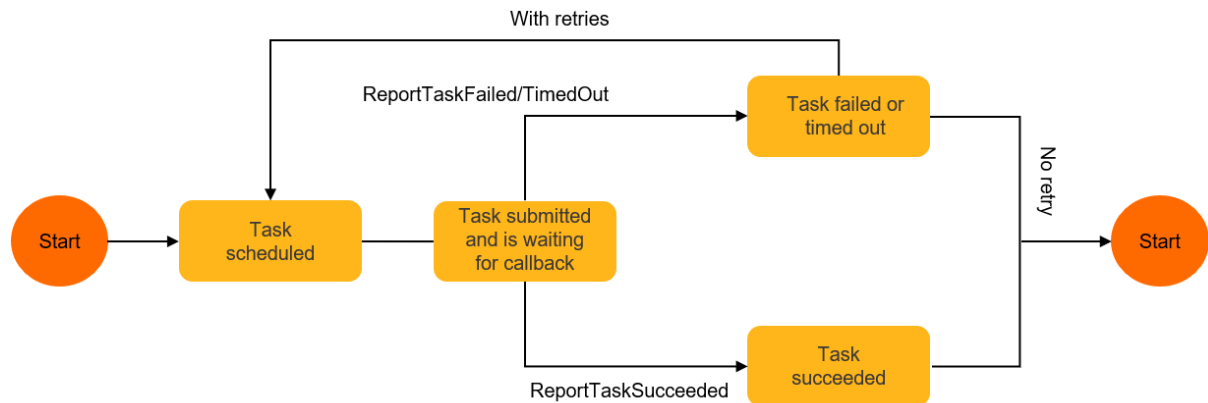
- Eliminate unnecessary delay caused by long polling.
- Eliminate unnecessary pressure on and waste of server resources caused by highly concurrent polling in large-traffic scenarios.
- Orchestrate tasks that do not involve functions in Function Compute, such as processes running in an on-premises data center or an Elastic Compute Service (ECS) instance.
- Automate steps that require manual intervention, such as notifying that a task has been approved.

The following figure shows how to use [Message Service \(MNS\) queues](#) with the callback API to orchestrate user-created resources in Serverless workflow.



Callback usage

In the **task step**, specify `pattern: waitForCallback`. As shown in the following figure, after the task, such as Function Compute call, specified in `resourceArn` is submitted, this step stores `taskToken` to the `context` object of the step and is suspended until Serverless workflow receives that the callback or the specified task times out. When `taskToken` is passed to the `ReportTaskSucceeded` or `ReportTaskFailed` operation for callback, this step continues.



```

- type: task
  name: mytask
  resourceArn: acs:fc:::services/{fc-service}/functions/{fc-function}
  pattern: waitForCallback # Enables the task step to wait for callback after the task
                           is submitted.
  inputMappings:
    - target: taskToken
      source: $context.task.token # Uses taskToken in the context object as an input f
or the function that is specified in resourceArn.
  outputMappings:
    - target: k
      source: $local.key # Maps output {"key": "value"} in ReportTaskSucceeded to {"k"
: "value"} and uses the mapped data as the output of this step.
  
```

Example

This example consists of the following three steps:

1. Prepare a task function.
2. Start a flow.
3. Perform callback.

Step 1: Prepare a task function

1. Create a simple function. The function directly returns the input.
 - Service: fnf-demo.
 - Function: echo.
 - Runtime environment: Python 2.7.
 - Entry point: index.handler.

```
#!/usr/bin/env python
import json
def handler(event, context):
    return event
```

Step 2: Start a flow

1. In the [Serverless workflow console](#), create the following flow and execute it.
 - o Flow name: fnf-demo-callback.
 - o Flow role: a role with the Function Compute Invocation permission.

```
version: v1
type: flow
steps:
  - type: task
    name: mytask
    resourceArn: acs:fc::services/fnf-demo/functions/echo
    pattern: waitForCallback
    inputMappings:
      - target: taskToken
        source: $context.task.token
    outputMappings:
      - target: s
        source: $local.status
```

After the flow starts, the `mytask` step is suspended in the `TaskSubmitted` event and waits for the callback. The event `output` contains `taskToken` that identifies the callback task.

Execution History		Input/Output			
ID	Type	Step	Timestamp	Relative time (ms)	
+	1	ExecutionStarted	Aug 16, 2020, 14:57:31	0	
+	2	StepEntered	mytask	Aug 16, 2020, 14:57:31	23
+	3	TaskScheduled	mytask	Aug 16, 2020, 14:57:31	31
+	4	TaskStarted	mytask	Aug 16, 2020, 14:57:32	801
-	5	TaskSubmitted	mytask	Aug 16, 2020, 14:57:32	826

```
{ 1 item
  "output": { 1 item
    "taskToken": "dJeJm5mLr1wBtY2F6bGJhY2ejNDWEYjI0NmQtMDZlYS1hNDYwLTk2NDItZDY4MWZhMWZkMWNhIzIjZlU0ZUx0ZC0RNRQRhMWN2VhJ5YU4tMWMpGUzZzPQ=="
  }
}
```

Step 3: Perform the callback

1. Use Serverless workflow [Python SDK](#) to run the `callback.py` script locally or in any environment where Python can run. Replace {task-token} with the value of the `TaskSubmitted` event.

```
cd /tmp
mkdir fnf-demo-callback
cd fnf-demo-callback
# Install Serverless Workflow Python SDK in a virtual environment.
virtualenv env
source env/bin/activate
pip install -t . aliyun-python-sdk-core
pip install -t . aliyun-python-sdk-fnf
# Run the worker process.
export ACCOUNT_ID={your-account-id}; export AK_ID={your-ak-id}; export AK_SECRET={your-ak-secret}
python worker.py {task-token-from-TaskSubmitted}
```

```
# The worker.py code:
import os
import sys
from aliyunsdkcore.acs_exception.exceptions import ServerException
from aliyunsdkcore.client import AcsClient
from aliyunsdkfnf.request.v20190315 import ReportTaskSucceededRequest
def main():
    account_id = os.environ['ACCOUNT_ID']
    akid = os.environ['AK_ID']
    ak_secret = os.environ['AK_SECRET']
    fnf_client = AcsClient(akid, ak_secret, "cn-hangzhou")
    task_token = sys.argv[1]
    print "task token " + task_token
    try:
        request = ReportTaskSucceededRequest.ReportTaskSucceededRequest()
        request.set_Output("{\"status\": \"ok\"}")
        request.set_TaskToken(task_token)
        resp = fnf_client.do_action_with_exception(request)
        print "Report task succeeded finished"
    except ServerException as e:
        print(e)
if __name__ == '__main__':
    main()
```

After the callback by using the preceding script is successful, the `mytask` step continues, and the `{"status": "ok"}` output specified in `ReportTaskSucceeded` is mapped by output Mappings to `{"s": "ok"}`.

6.Schedule reserved resource functions or functions with specified versions

This topic describes how a flow in Serverless workflow schedules reserved resource functions or functions with specified versions.

Overview

In actual production scenarios, functions scheduled by the task flow may frequently change due to changes in service scenarios. Therefore, you must avoid unexpected actions caused by the changes and control the stability of the task flow. In the following scenarios, functions of a fixed version will help in Serverless workflow task steps:

- Flow A orchestrates multiple functions f1, f2, and f3. The same task must execute the same version of the functions. For example, when flow A is under execution, function f1 has been executed, but the function is updated at this time. In this case, the latest versions of functions f2 and f3 may be executed in flow A, which may cause unexpected results. Therefore, the version of the functions that the flow executes must be fixed.
- A function needs to be rolled back. If you find that the flow failed due to a new change after the function is launched, you must roll back the flow to the previous fixed version.
- The function alias is used to call a reserved resource function, reduce the function cold start time, and [Best practice for cost optimization](#).

Functions of different [Introduction to versions](#) that are deployed in Function Compute can efficiently support continuous integration and release in similar scenarios. The following section uses an example to describe how to use the function alias in the flow to call a reserved resource function. Reserved resource functions depend on the functions of a specified version. You can see this example in scenarios where functions of specified versions are required.

Implementation in Serverless workflow

This operation consists of the following three steps:

1. [Create a reserved instance for a function](#).
2. [Create a flow](#).
3. [Execute the reserved function and check the execution result in the console or on a command line interface \(CLI\)](#).

Step 1: Create a reserved instance for a function

1. Create a service named `fnf-demo` in Function Compute. In this service, create a Python 3 function named provision and release its version and alias to generate a reserved instance. For more information, see [Introduction to supplementary examples](#).

Assume that the version of the created function is 1, the alias is online, and a reserved instance is generated. The following code shows the content of the function:

```
import logging
def handler(event, context):
    logger = logging.getLogger()
    logger.info('Started function test')
    return {"success": True}
```

Step 2: Create a flow

Serverless workflow natively supports the versions and aliases of functions in Function.

In the task step of Serverless workflow, enter the default value `acs:fc:{region}:` `{accID}:services/fnf/functions/test` in the resourceArn parameter. Based on the function execution rule, the function of the latest version is executed by default. You can release the [Manage versions](#) or [Manage aliases](#) and enter `acs:fc:{region}:{accID}:services/fnf.{ alias or version}/functions/test` in the resourceArn parameter of the task step in the flow to call the function of the specified version. Therefore, define the flow based on the following code:

```
version: v1
type: flow
steps:
  - type: task
    resourceArn: acs:fc:::services/fnf-demo.online/functions/provision
    # You can also use the version by defining resourceArn: acs:fc:::services/fnf-demo.1/fu
    nctions/provision.
    name: TestFCProvision
```

Step 3: Execute the reserved function and check the execution result in the console or on a CLI

1. Execute the flow. The following figure shows the execution details before the reserved mode is used.

+	4	TaskStarted	TestFCProvision	Aug 16, 2020, 15:05:11	760
+	5	TaskSucceeded	TestFCProvision	Aug 16, 2020, 15:05:11	585

2. The following figure shows the execution details after the reserved mode is used.

+	4	TaskStarted	TestFCProvision	Aug 16, 2020, 15:05:11	760
+	5	TaskSucceeded	TestFCProvision	Aug 16, 2020, 15:05:11	585

As shown in the figures, after the reserved mode is used, the flow execution time is reduced from 500 ms to 230 ms.

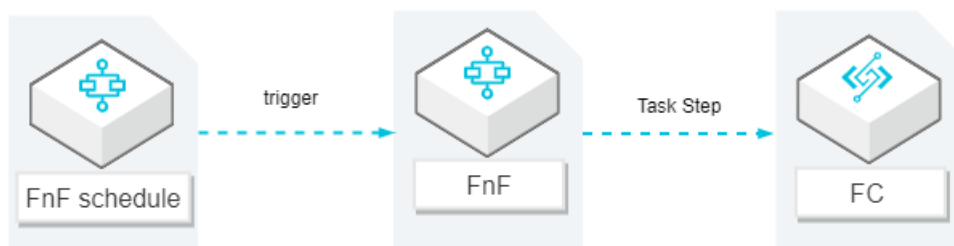
7. Create time-based schedules for a flow

This topic describes how to execute a flow in Serverless Workflow on a specified schedule to call functions deployed in Function Compute.

Execution process

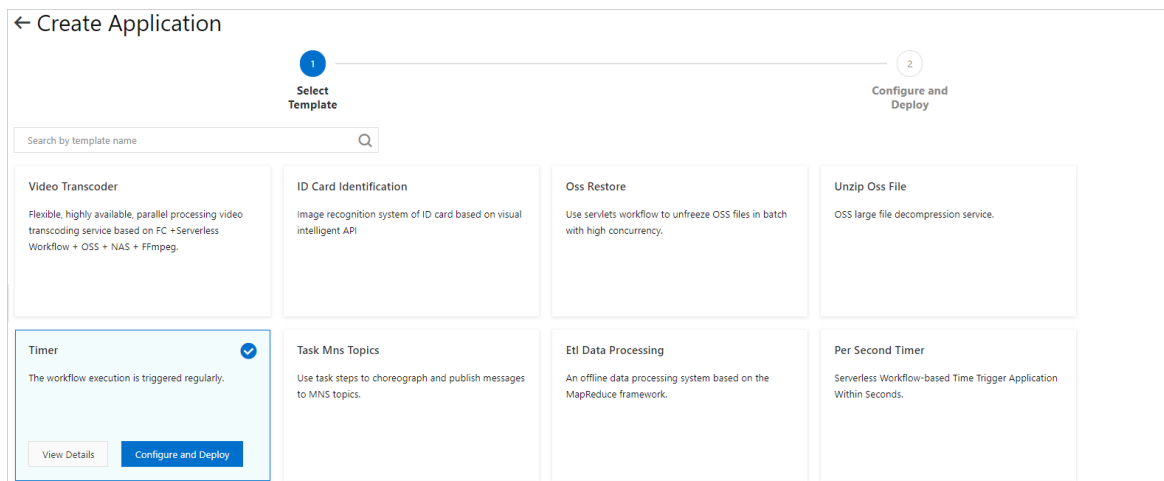
The process of a specified schedule for a flow consists of the following steps:

1. In Serverless Workflow, define a task step that calls a function in Function Compute.
2. In Serverless Workflow, create a time-based schedule. Both the flow and the function in the flow are executed on the schedule.



Procedure

1. Log on to the [Serverless Workflow console](#).
2. In the left-side navigation pane, choose **Application Center > Create Application**.
3. On the **Create Application** page, click the **Select Template** tab. Select the **Timer** template and click **Configure and Deploy**.



4. On the **Configure and Deploy** tab, configure the parameters, and then click **Deploy**.

← Create Application
Timer

✓
Select Template

2
Configure and Deploy

Basic Configuration

* Application Name

The name must be 1 to 64 characters in length and can contain digits, hyphens (-), and underscores (_). It must start with a letter.

Parameter Configuration

Cron

cron expression

Input

target flow json input

Resource Preview

Correctly configure all the parameters to preview the resource.

Previous

Deploy

The following table describes the parameters.

Parameter	Description
Application Name	<p>The name of the application, which must be unique in the same account.</p> <div> ? Note Your application is a custom Resource Orchestration Service (ROS) resource. You can log on to the ROS console to view the application. </div>
Cron	<p>The Cron expressions to execute the flow on a specified schedule. For more information, see Parameters for time-based schedules.</p>
Input	<p>The input of the flow that is executed on a specified schedule. The input must be in JSON format and its default value is null. For more information, see Input format.</p>

After the application is deployed, all the resources created by this application are displayed.

← ceshi_08_16 [Start execution](#)

Overview Deploy Monitoring

Output

applicationName timer
No description given

entrypointFlowName FnTimer-ceshi_08_16
No description given

Logical Resource Name	Physical Resource Name	Resource Type	Resource Status	Updated At
service	ceshi_08_16-service-261C83DF2C57	ALYUN::FC::Service	Created	Aug 16, 2020, 3:34:13
schedule		ALYUN::FNF::Schedule	Created	Aug 16, 2020, 3:34:16
servicehello	hello	ALYUN::FC::Function	Created	Aug 16, 2020, 3:34:15
flow	FnTimer-ceshi_08_16	ALYUN::FNF::Flow	Created	Aug 16, 2020, 3:34:15
flowRole	ceshi-08-16-flowRole-3230DA0F786D	ALYUN::RAM::Role	Created	Aug 16, 2020, 3:34:13
AliyunFCInvocationAccessflowRole		ALYUN::RAM::AttachPolicyToRole	Created	Aug 16, 2020, 3:34:15

- Role of the RAM user: AliyunFCInvocationAccessflowRole with the function call permission and flowRole with the flow permission.
 - Function Compute resources: service and the servicehello function.
 - Serverless Workflow resources: flow and time-based schedule (ALYUN::FNF::Schedule).
5. Log on to the [Serverless Workflow console](#) and check whether the flow you created is executed on the specified schedule.

← Flows FnTimer-ceshi_08_16 [Edit](#) [Monitoring and Alerts](#) [Share](#) [Delete](#)

Details

Name: ceshi_08_16 Description: Fnf time trigger demo flow
RAM role: acsram:1880770869023420role/ceshi-08-16-flowrole-3230DA0F786D Created time: Aug 16, 2020, 15:34:13

Executions Definition Schedule

[Start Execution](#) Please select status Please input execution name

Name	Status	Started Time	End Time	执行时间	Action
scheduled-exec-20200816T073615Z-bde04769-e2e8-4047-bd39-7a8c4660a3de	Succeeded	Aug 16, 2020, 15:36:15	Aug 16, 2020, 15:36:17	1.727 s	
scheduled-exec-20200816T073515Z-deb010da-067c-4511-8721-10447b073e19	Succeeded	Aug 16, 2020, 15:35:15	Aug 16, 2020, 15:35:15	0.183 s	
879c446f-c909-31e9-9e5f-45bd5b0aae47	Succeeded	Aug 16, 2020, 15:35:03	Aug 16, 2020, 15:35:06	2.128 s	

Items per Page 10 20 50 < Previous 1 Next >

The following code shows the definition of a flow that calls the `hello` function of Function Compute in the [task step](#).

```
version: v1
type: flow
steps:
  # task step to invoke FC function hello
  - type: task
    name: hello
    resourceArn: acs:fc:::services/service-CD946B9A9F36/functions/hello
```

You can modify the definition of this flow to implement your business logic. For more information, see [Modify a flow](#).

8. Troubleshooting

You can integrate Serverless workflow with multiple Alibaba Cloud services. When cloud services are used as execution nodes of task steps in Serverless workflow, you can troubleshoot execution errors based on your business scenarios by catching errors or retrying tasks. This ensures the stable execution of your tasks in production scenarios. This topic describes how to troubleshoot errors in Serverless Workflow in different business scenarios.

Troubleshooting methods

You can use task steps in Serverless workflow to catch errors and retry or redirect tasks after errors are caught. For more information, see [Task steps](#).

- Retry a task after an error is caught.

```
steps:
  - type: task
    name: hello
    resourceArn: acs:fc:{region}:{accountID}:xxx
    retry:
      - errors:
          - FnF.ALL
        intervalSeconds: 10
        maxIntervalSeconds: 300
        maxAttempts: 3
        multiplier: 2
```

Parameters for retrying a task after an error is caught

Parameter	Description
<code>retry</code>	Specifies that the task is retried after an error is caught.
<code>errors</code>	The list of errors to be caught.
<code>intervalSeconds</code>	The initial interval between retry attempts. Maximum value: 86400. Default value: 1. Unit: seconds.
<code>maxAttempts</code>	The maximum number of retry attempts. Default value: 3.
<code>multiplier</code>	The multiplier by which the retry interval increases during each attempt. Default value: 2. In the preceding sample code, the second retry attempt is performed after 20 seconds, and the third retry attempt is performed after 40 seconds.

- Redirect a task after an error is caught.

```

steps:
  - type: task
    name: hello
    resourceArn: acs:fc:{region}:{accountID}:xxx
    errorMappings:
      - target: errMsg
        source: $local.cause # This value is reserved for the system and can be directly
        used when an error occurs in this step.
      - target: errCode
        source: $local.error # This value is reserved for the system and can be directly
        used when an error occurs in this step.
    catch:
      - errors:
        - FnF.ALL
      goto: final

```

Parameters for redirecting a task after an error is caught

Parameter	Description
<code>errorMappings</code>	The error fields in this step that can be passed in the redirection.
<code>catch</code>	The policy based on which errors are caught in the task.
<code>errors</code>	The list of errors to be caught.
<code>goto</code>	The object to which the task is redirected after the task throws an error.

Use Function Compute as an execution node of a task in Serverless workflow

When Function Compute serves as an execution node of a task in Serverless workflow, take note of the following error types:

- Exceptions prompted by Function Compute
- Function code errors

To catch these errors, you can specify `errors` in a task in Serverless workflow.


The following code describes common error types:

```

- errors:
  - FC.ResourceThrottled
  - FC.ResourceExhausted
  - FC.InternalServerError
  - FC.Unknown
  - FnF.TaskTimeout
  - FnF.ALL

```

Common error types

Error type	Description
<code>FC.</code> <code>{ErrorCode}</code>	<p>Function Compute returns HTTP status codes other than 200. The following common error types are included:</p> <ul style="list-style-type: none"> <code>FC.ResourceThrottled</code> : Your functions are throttled due to high concurrency. All your functions are controlled by a total concurrency value. Serverless workflow invokes Function Compute when the task node is executed. The total concurrency value is combined with the concurrency values of other invocation methods. You can apply to modify the value. <code>FC.ResourceExhausted</code> : Your functions are throttled due to insufficient resources. Contact us when errors of this type occur. <code>FC.InternalServerError</code> : A system error occurs on Function Compute. Execute the flow again. <p> Note <code>{Error code}</code> indicates the error code of Function Compute. For more information, see Error codes.</p>
<code>FC.Unknown</code>	Function Compute has invoked the function, but an error occurred during the function execution and the error was not caught. Example: <code>UnhandledInvocationError</code> .
<code>{CustomError}</code>	Function Compute has invoked the function, but the function threw an exception.
<code>FnF.TaskTimeout</code>	The execution of a step in Serverless workflow times out.
<code>FnF.ALL</code>	All errors in Serverless workflow are caught.
<code>FnF.Timeout</code>	The overall execution in Serverless workflow times out.

In addition to common errors in Function Compute and Serverless workflow, you can also customize error types. You can edit function code to throw an exception and pass the state or error of an execution to Serverless workflow. Then, Serverless workflow retries or redirects the task based on the flow. The following function code shows how to customize an error type in Python and specifies how to retry tasks that throw this type of error in Serverless workflow. To handle an error of a custom type, perform the following operations:

1. Customize an error type in function code.

```
...
class ErrorNeedsRetry(Exception):
    pass
def handler(event, context):
    try:
        # do sth
    except ServerException:
        raise ErrorNeedsRetry("custom error message")
```

2. Modify the task step in Serverless workflow to catch the error and retry the task.

```
retry:
  - errors:
    - ErrorNeedsRetry
  intervalSeconds: 10
  maxAttempts: 3
  multiplier: 2
```

Common system errors prompted by Function Compute or Serverless Workflow

Custom error types

Use another cloud service such as MNS as an execution node of a task

If a cloud service of a third party is used as an execution node of a task, Serverless workflow directly calls an API operation of the service to distribute the task.

When Message Service (MNS) is used as the execution node, Serverless workflow calls the `SendMessage` operation of MNS to send messages. For more information, see [SendMessage](#). In most cases, you can call API operations to execute such tasks. Results of function executions are not expected. After an error is caught, retry attempts are performed in Serverless workflow for up to the specified number of times. When you use cloud services such as MNS and Visual Intelligence API as execution nodes, you do not need to handle errors in the flow.