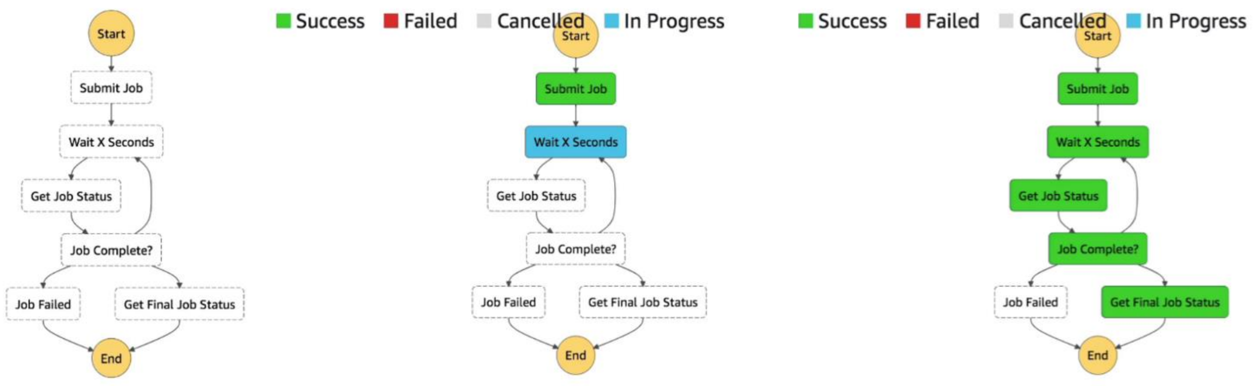
AWS Step Function

AWS Step Functions is a fully managed service that allows you to coordinate multiple AWS services into serverless workflows. It simplifies the process of building complex, distributed applications by defining workflows that coordinate between services and handle retries, parallelization, error handling, and state transitions.

Why Use AWS Step Functions?

- **Workflow Orchestration:** AWS Step Functions provide a visual way to design and run workflows that stitch together services like AWS Lambda, Amazon S3, Amazon DynamoDB, and more.
- **Error Handling and Retries:** Automatically handle errors, retries, and catch failures to ensure that workflows are robust and reliable.
- **State Management:** Maintain the state of your application throughout the execution of your workflows, allowing for complex logic and conditional branching.
- **Decoupling:** Decouple the logic of your application from the infrastructure, making it easier to manage and scale.
- **Parallel Execution:** Execute tasks in parallel to improve performance and reduce the overall execution time of your workflows.
- **Integration with AWS Services:** Seamlessly integrate with a wide range of AWS services, making it easier to build serverless applications.

If you look at this diagram , we can understand – what typical Step Function is used for



Real-Time Use Cases

Here are some real-time use cases where AWS Step Functions can be highly beneficial:

Order Processing System:

Scenario: An e-commerce platform processes customer orders.

Workflow:

- Validate the order.
- Process payment using AWS Lambda or an external payment gateway.
- Update the inventory in Amazon DynamoDB.
- Send an order confirmation email using Amazon SNS or SES.
- Initiate shipping using a third-party shipping service.

Data Processing Pipeline:

Scenario: A data analytics company needs to process large datasets.

Workflow:

- Extract data from an Amazon S3 bucket.
- Transform the data using AWS Lambda.
- Load the transformed data into Amazon Redshift or Amazon RDS.
- Run data validation and quality checks.
- Notify the stakeholders about the completion of the data processing job.

ETL (Extract, Transform, Load) Jobs:

Scenario: A business performs regular ETL tasks to update their data warehouse.

Workflow:

- Extract data from various sources like Amazon S3, RDS, or external APIs.
- Transform the data using AWS Glue or AWS Lambda.
- Load the transformed data into Amazon Redshift or an external database.
- Run validation and cleanup tasks.
- Send a notification on completion.

Microservices Coordination:

Scenario: A microservices-based application needs to manage interactions between different services.

Workflow:

- Invoke various microservices using AWS Lambda or API Gateway.
- Handle dependencies and conditional logic between microservices.
- Manage retries and error handling for each service call.
- Aggregate responses and send a final response back to the user.

Batch Processing:

Scenario: A company processes batches of user data for analytics.

Workflow:

- Trigger batch processing at scheduled intervals using Amazon CloudWatch Events.
- Read the batch data from Amazon S3.
- Process the data using AWS Lambda or Amazon EMR.
- Store the processed data in a database or another S3 bucket.
- Send notifications upon completion.

Machine Learning Workflows:

Scenario: A data science team runs machine learning training and inference jobs.

Workflow:

- Preprocess training data using AWS Lambda or AWS Glue.
- Train a model using Amazon SageMaker.
- Evaluate the model and perform hyperparameter tuning.
- Deploy the model for inference.
- Monitor the model performance and send alerts.

Few basic definitions we need to know, before we jump into how it works and what it does

State Machine:

A state machine is a workflow comprised of states, each representing a single step in the workflow.

Structure: The state machine definition is written in JSON using Amazon States Language (ASL).

Types: There are two types of state machines:

- Standard Workflows: Suited for long-running, durable workflows.
- Express Workflows: Suited for high-volume, short-duration workflows.

States:

Types of States: There are several types of states:

- Task: Executes a unit of work by invoking an AWS service or a Lambda function.
- Choice: Adds branching logic to your workflow.
- Parallel: Executes multiple branches of work in parallel.
- Map: Iterates over a collection of items and executes steps for each item.
- Wait: Delays the state machine for a specified time.
- Pass: Passes input to output without performing work.
- Succeed: Terminates the state machine execution successfully.
- Fail: Terminates the state machine execution with a failure.

Tasks:

a Task is a state in a state machine that performs a single unit of work. Tasks are fundamental components of state machines, allowing you to integrate with other AWS services or run your own custom logic. These are 2 types (Task Token & Activity Token)

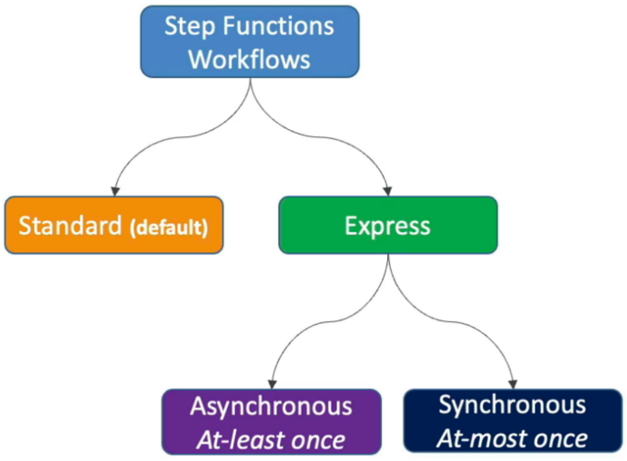
Amazon States Language (ASL)

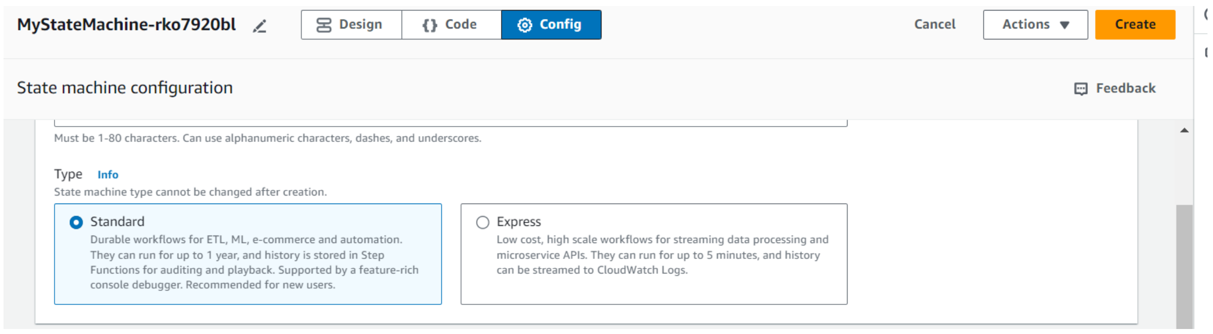
JSON-Based: ASL is a JSON-based language used to define your state machine.

Structure:

- Comment: Optional description of the state machine.
- StartAt: Specifies the name of the state to start with.
- States: Defines all the states in the workflow.
- End: Indicates that a state is a terminal state.

Coming to the AWS Step Function Workflow Types





State machine configuration

Must be 1-80 characters. Can use alphanumeric characters, dashes, and underscores.

Type Info

State machine type cannot be changed after creation.

[✓ Standard]
Durable workflows for ETL, ML, e-commerce and automation. They can run for up to 1 year, and history is stored in Step Functions for auditing and playback. Supported by a feature-rich console debugger. Recommended for new users.

Express
Low cost, high scale workflows for streaming data processing and microservice APIs. They can run for up to 5 minutes, and history can be streamed to CloudWatch Logs.

Standard Workflows (default)

Cost-effective for lower volume, longer-running, and complex workflows.

AWS Step Functions Standard Workflows are designed for managing long-running, complex workflows with robust error handling and detailed execution history. They are ideal for scenarios where precise control over each step and its transitions is crucial.

- Max duration: 1 year
- Execution model: **Exactly-once Execution**
- Execution rate: Over **2,000 workflow executions** per sec
- Execution history: **90 days in the console** (send to CloudWatch to retain for longer)
- Pricing: based on the number of state transitions
- Use cases: **Non-idempotent actions** (eg. payment processing)

Express Workflows

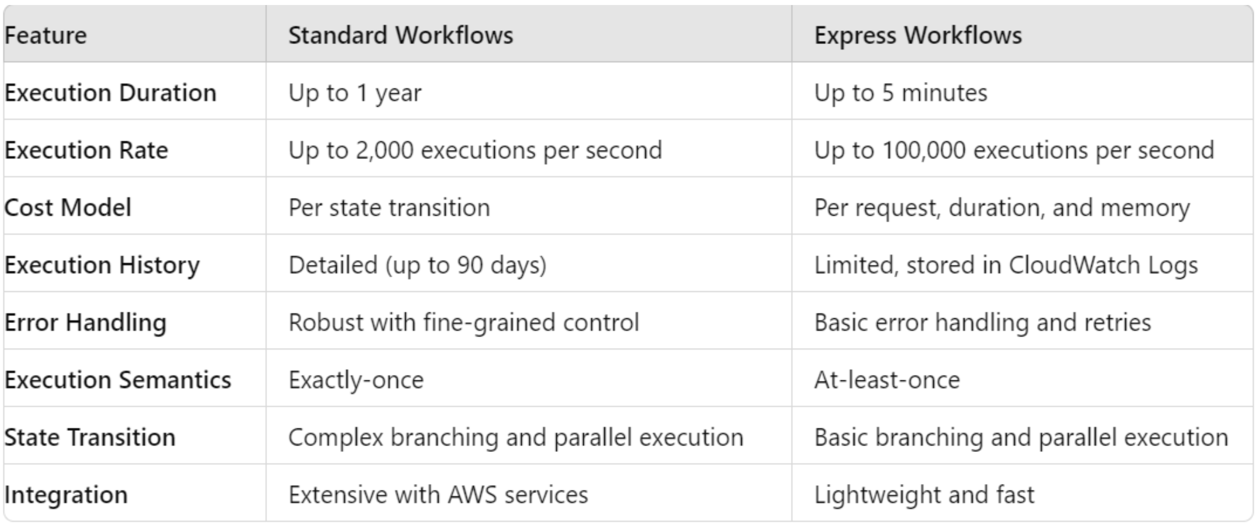
Cost-effective for high-volume, short-duration workflows.

AWS Step Functions Express Workflows are optimized for high-volume, short-duration workflows. They are designed to handle high execution rates and are suitable for event-driven applications where quick responses are needed.

- Max duration: 5 min
- Execution rate: Over 100,000 workflow executions per sec
- Execution history: Not available in the console (must use CloudWatch)
- Use cases: IoT data ingestion, streaming data, backend for apps, etc.
- Types:

  Asynchronous Express Workflows

  Synchronous Express Workflows



Feature	Standard Workflows	Express Workflows
Execution Duration	Up to 1 year	Up to 5 minutes
Execution Rate	Up to 2,000 executions per second	Up to 100,000 executions per second
Cost Model	Per state transition	Per request, duration, and memory
Execution History	Detailed (up to 90 days)	Limited, stored in CloudWatch Logs
Error Handling	Robust with fine-grained control	Basic error handling and retries
Execution Semantics	Exactly-once	At-least-once
State Transition	Complex branching and parallel execution	Basic branching and parallel execution
Integration	Extensive with AWS services	Lightweight and fast

Choosing Between Standard and Express Workflows

- ***Standard Workflows:*** Choose when you need detailed execution history, complex error handling, long-running workflows, or when exactly-once execution semantics are critical.
- ***Express Workflows:*** Choose when you need to handle high-volume, short-duration workflows, can tolerate at-least-once execution semantics, and require a cost-effective solution for high-throughput scenarios.

Synchronous Express Workflows

- In synchronous execution, the caller waits for the workflow to complete and receives the result directly.
- Suitable for use cases requiring immediate results, such as API responses, where the application needs the output of the workflow to proceed.
- Blocks the caller until the workflow completes.
- Immediate result is returned to the caller.
- Ideal for high-volume, short-duration tasks needing immediate feedback.
- Execution model: **At-most Once Execution**
- When the workflow is invoked, it completes it and returns the response.
- Can be invoked from **API Gateway** or **Lambda function**

At Most Once

Each step in the workflow is guaranteed to be executed no more than once. This means that the workflow will not repeat any step, even if there is an error.

Implications: There is a risk of losing some steps or data if an error occurs, as the steps won't be retried automatically.

Asynchronous Express Workflows

- In asynchronous execution, the caller does not wait for the workflow to complete. The workflow starts and runs independently, allowing the caller to proceed with other tasks.
- Ideal for high-volume, event-driven architectures where immediate results are not necessary.
- Does not block the caller.
- Caller receives an execution ARN to check status or get results later.
- Suitable for high-volume, event-driven tasks not needing immediate feedback.

▪ Execution model: **At-least Once Execution** (the operation must be idempotent because there could be retries)

▪ When the workflow is invoked, it just starts and doesn’t return the result of computation.

▪ Use cases: where we don’t need immediate response (eg. messaging services)

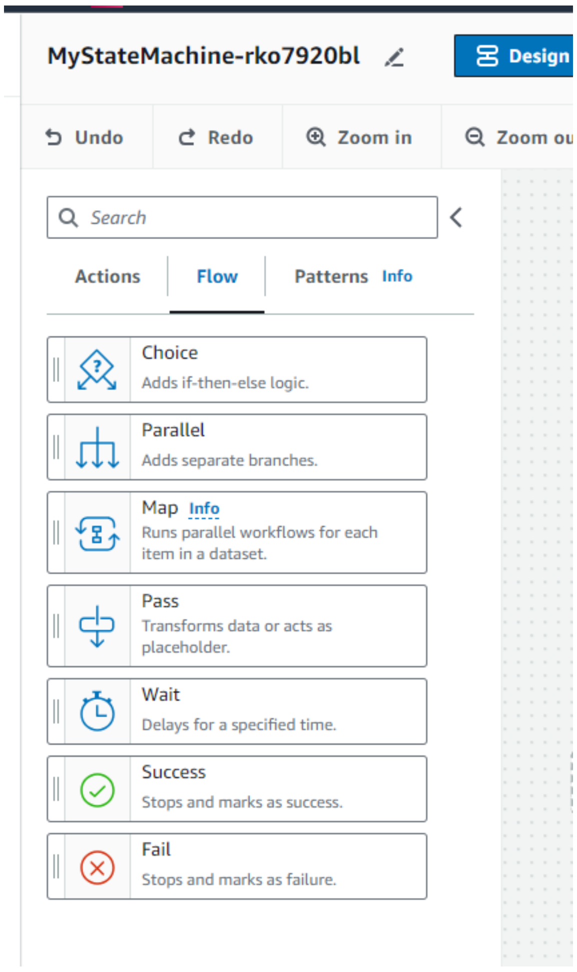
At Least Once

Each step in the workflow is guaranteed to be executed at least once. This means that in certain situations (such as retries due to errors or network issues), a step may be executed more than once.

Implications: If your workflow steps are not idempotent (i.e., they cannot handle being executed multiple times without causing unintended side effects), you might see issues due to the repeated execution.

We saw the states definitions in previous page - Now if we take a stab at states by considering our language that we use

- **Choice State** - Test for a condition to send to a branch (or default branch)
- **Fail or Succeed State** - Stop execution with failure or success
- **Pass State** - Simply pass its input to its output or inject some fixed data
- **Wait State** - Provide a delay for a certain amount of time or until a specified time or date
- **Map State** - Dynamically iterate steps
- **Parallel State** - Begin parallel branches of execution (**asynchronous** execution)
- **Task State** - Run some code **synchronously**



Choice
Adds if-then-else logic.

Parallel
Adds separate branches.

Map Info
Runs parallel workflows for each item in a dataset.

Pass
Transforms data or acts as placeholder.

Wait
Delays for a specified time.

Success
Stops and marks as success.

Fail
Stops and marks as failure.

Error Handling

- Handle errors in the state machine instead of the application code. This makes the application logic simpler. Also, step functions provide execution history.

- Predefined error codes:

  - `States.ALL` - any error

  - `States.Timeout` - task ran longer than `TimeoutSeconds` or no heartbeat received

  - `States.TaskFailed` - task execution failure

  - `States.Permissions` - insufficient privileges to execute code

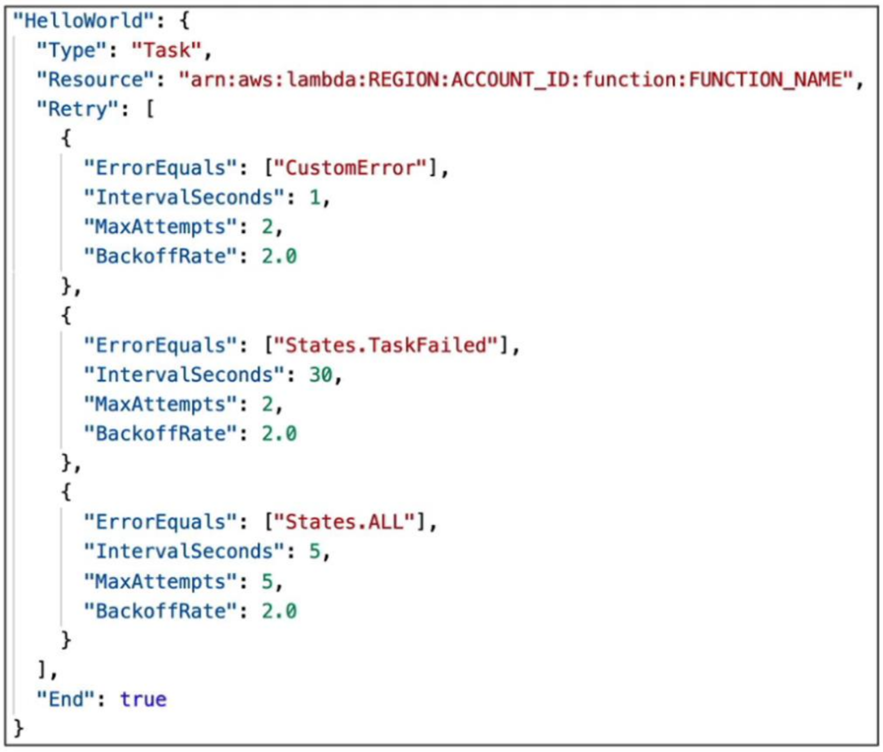
- The state can also throw custom errors that can be caught in the step function (eg. a Lambda function throwing a custom error)

Now we will see Error handling states …!!

Error handling states

Retry

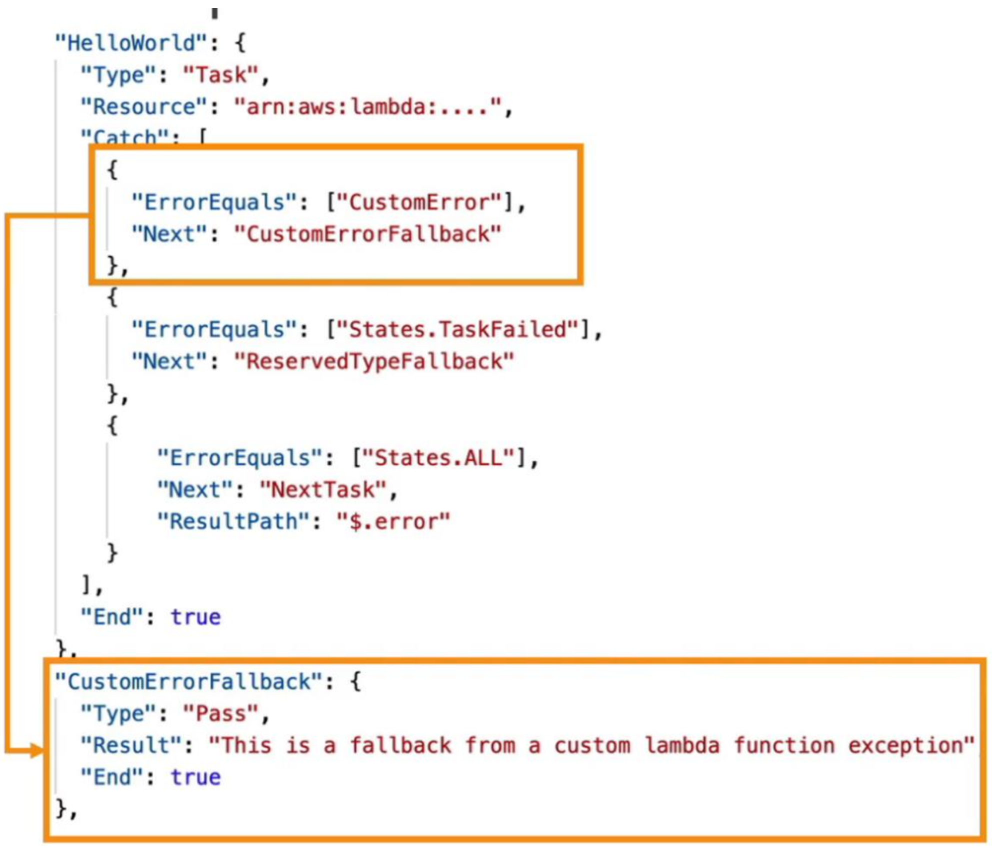
- Retry failed state
- Retry block is evaluated top to bottom
- BackoffRate - what factor the IntervalSeconds should be multiplied with at each retry
- When MaxAttempts are reached, the Catch block kicks in



```
"HelloWorld": {
  "Type": "Task",
  "Resource": "arn:aws:lambda:REGION:ACCOUNT_ID:function:FUNCTION_NAME",
  "Retry": [
    {
      "ErrorEquals": ["CustomError"],
      "IntervalSeconds": 1,
      "MaxAttempts": 2,
      "BackoffRate": 2.0
    },
    {
      "ErrorEquals": ["States.TaskFailed"],
      "IntervalSeconds": 30,
      "MaxAttempts": 2,
      "BackoffRate": 2.0
    },
    {
      "ErrorEquals": ["States.ALL"],
      "IntervalSeconds": 5,
      "MaxAttempts": 5,
      "BackoffRate": 2.0
    }
  ],
  "End": true
}
```

Catch

- Transition to failed path
- Catch block is evaluated top to bottom
- After all the retries have been exhausted, the state function goes into Catch
- If the error is of X type, go to the next state Y.
- ***ResultPath - a path that determines what input is sent to the state specified in the Next field.***

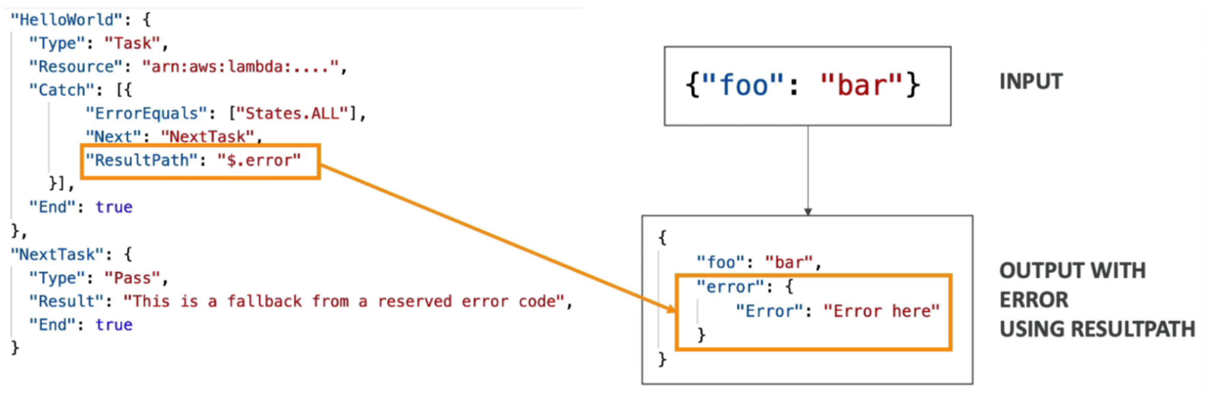


```
"HelloWorld": {
  "Type": "Task",
  "Resource": "arn:aws:lambda:....",
  "Catch": [
    {
      "ErrorEquals": ["CustomError"],
      "Next": "CustomErrorFallback"
    },
    {
      "ErrorEquals": ["States.TaskFailed"],
      "Next": "ReservedTypeFallback"
    },
    {
        "ErrorEquals": ["States.ALL"],
        "Next": "NextTask",
        "ResultPath": "$.error"
    }
  ],
  "End": true
},
"CustomErrorFallback": {
  "Type": "Pass",
  "Result": "This is a fallback from a custom lambda function exception"
  "End": true
},
```

ResultPath is used to control how the result of a Task (or other state types that produce results) is incorporated into the state machine's JSON data. It allows you to specify where in the input JSON the result should be inserted.

How ResultPath Works

- Default Behaviour: If ResultPath is not specified, the output of the Task replaces the input JSON.
- Custom Behaviour: By using ResultPath, you can insert the Task's output into a specific part of the input JSON, or you can manipulate the input and output data to produce a desired state output.



```
"HelloWorld": {
  "Type": "Task",
  "Resource": "arn:aws:lambda:....",
  "Catch": [{
        "ErrorEquals": ["States.ALL"],
        "Next": "NextTask",
        "ResultPath": "$.error"
    }],
  "End": true
},
"NextTask": {
  "Type": "Pass",
  "Result": "This is a fallback from a reserved error code",
  "End": true
}
```

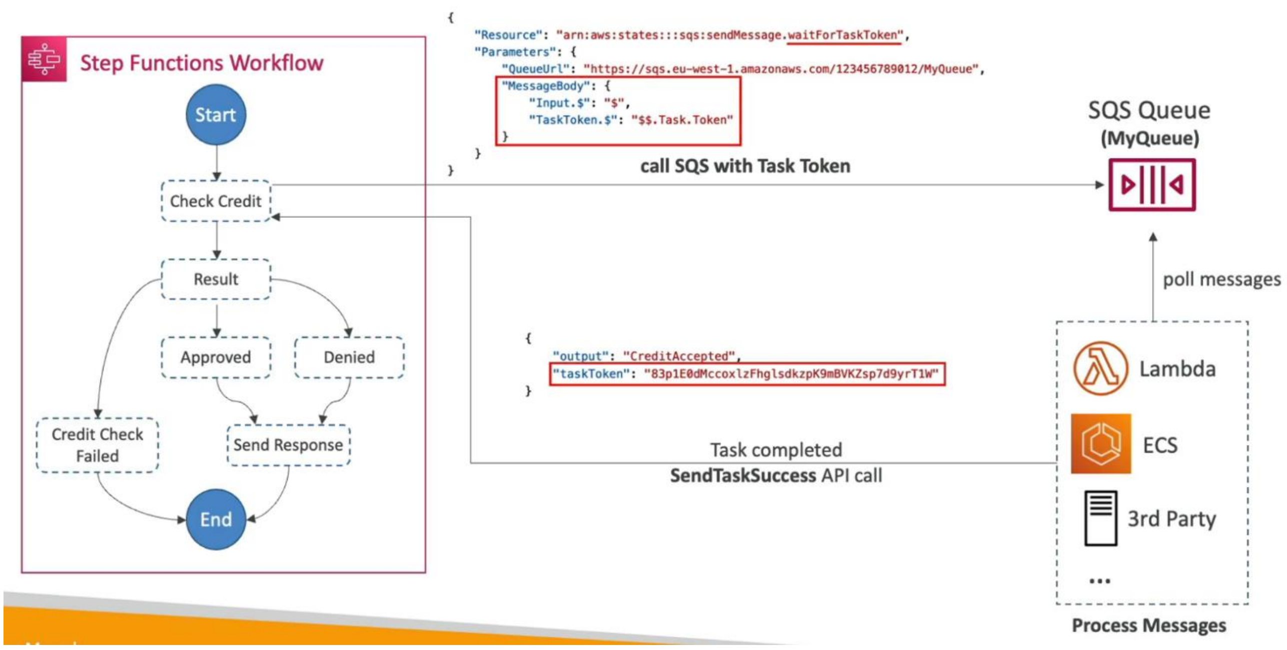
```
{"foo": "bar"}
```

```
{
    "foo": "bar",
    "error": {
        "Error": "Error here"
    }
}
```

Till now we saw the workflow types, state types and now we will see the Task Types ..!!

Token Task or Task Token (Push based)

- Used to integrate the workflow with an external task where the external task must finish the job before the workflow proceeds further.
- **Push-based** (the task pushes the work to the external application or worker which after completion invokes a callback API)
- The task is paused until it receives the callback (API call) with the TaskToken
- Append .waitForTaskToken to the Resource field to tell Step Functions to wait for the Task Token to be returned. Example: "Resource": "arn:aws:states:::sqs:sendMessage.waitForTaskToken"
- **Working**: The step function is paused during the Check Credit task execution, where we pass the TaskToken to the external application (push to SQS). After the external application is done processing, it makes a SendTaskSuccess API call with the result of processing and the passed TaskToken. This means the external task was executed successfully and the step function can continue execution. If the external application fails to process, it makes a SendTaskFailure API call which is treated as task failed.

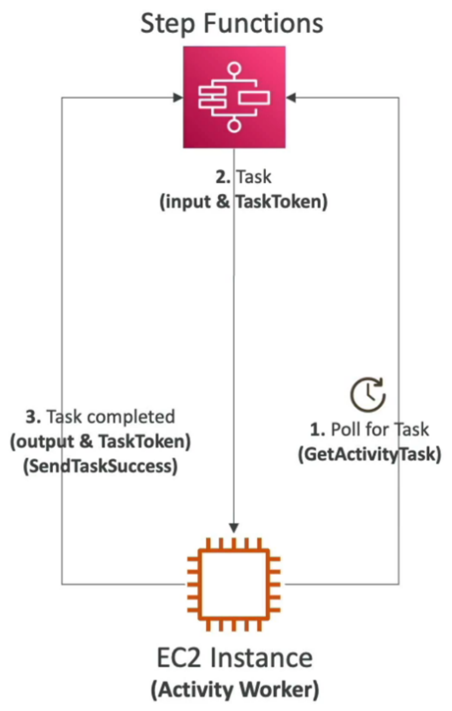


```
{
    "Resource": "arn:aws:states:::sqs:sendMessage.waitForTaskToken",
    "Parameters": {
        "QueueUrl": "https://sqs.eu-west-1.amazonaws.com/123456789012/MyQueue",
        "MessageBody": {
            "Input.$": "$",
            "TaskToken.$": "$$.Task.Token"
        }
    }
}
```

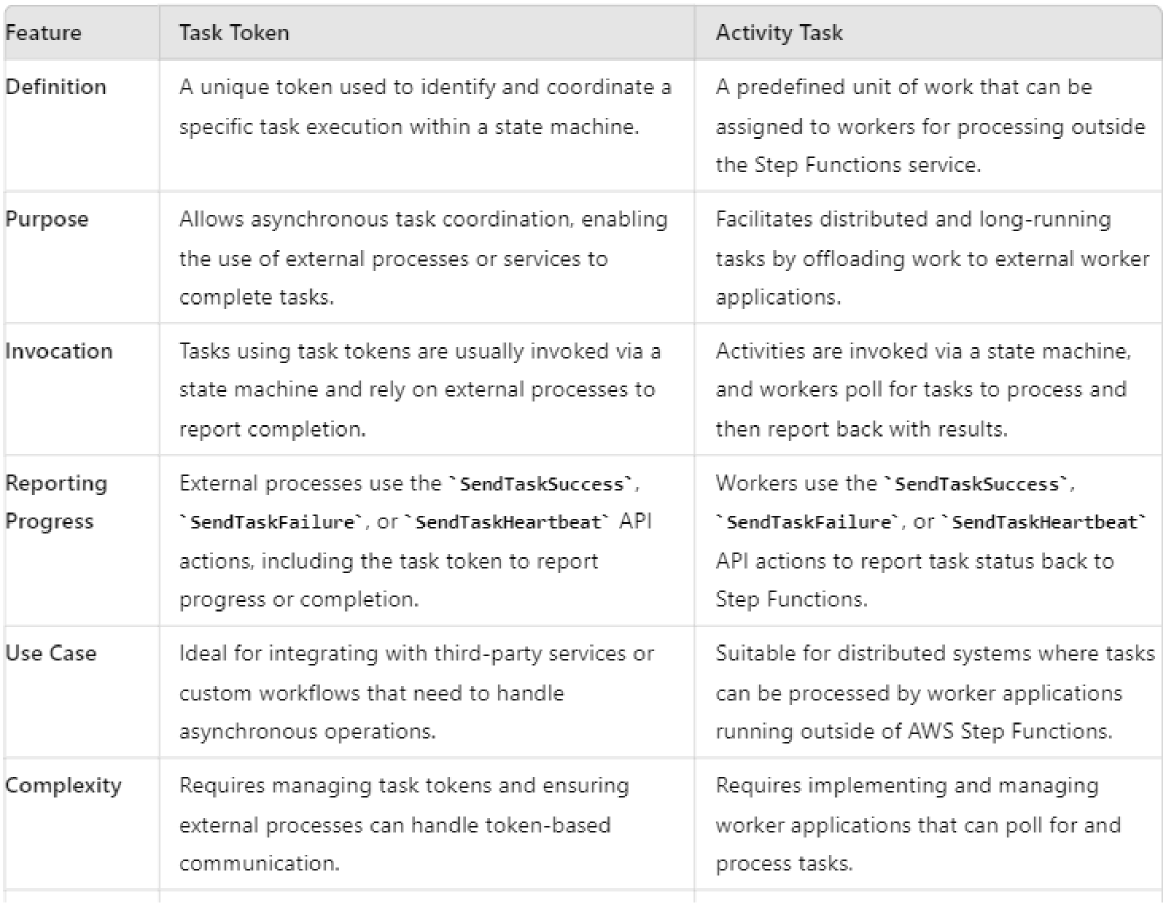
```
{
    "output": "CreditAccepted",
    "taskToken": "83p1E0dMccoxlzFhglsdkzpK9mBVKZsp7d9yrT1W"
}
```

Activity Task (Pull based Task)

- **Activity Workers** (running on any compute resource) poll the workflow for tasks using GetActivityTask API.
- If an activity worker gets a task, it will complete it and send the response of success or failure as SendTaskSuccess or SendTaskFailure API call.
- **Pull-based** (tasks are pulled by the Activity Workers)
- TaskToken is used to identify which task got completed (same way as in Wait for Task Token)
- To keep the Task active:
    - Configure TimeoutSeconds on the step function - how long the task will wait for the activity worker to complete (max 1 year)
    - Send heartbeats periodically from the activity worker to the task using SendTaskHeartBeat at an interval less than HeartBeatSeconds parameter (set in the step function).



Now we need to understand

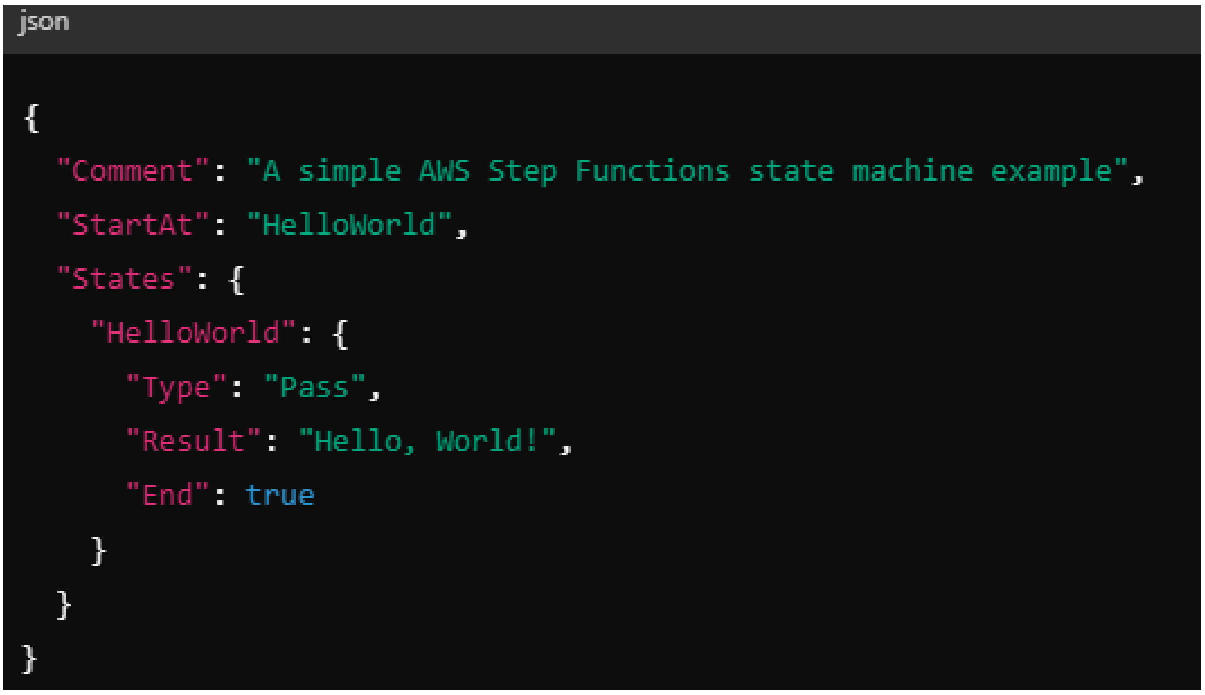


Feature	Task Token	Activity Task
Definition	A unique token used to identify and coordinate a specific task execution within a state machine.	A predefined unit of work that can be assigned to workers for processing outside the Step Functions service.
Purpose	Allows asynchronous task coordination, enabling the use of external processes or services to complete tasks.	Facilitates distributed and long-running tasks by offloading work to external worker applications.
Invocation	Tasks using task tokens are usually invoked via a state machine and rely on external processes to report completion.	Activities are invoked via a state machine, and workers poll for tasks to process and then report back with results.
Reporting Progress	External processes use the `` `SendTaskSuccess` ``, `` `SendTaskFailure` ``, or `` `SendTaskHeartbeat` `` API actions, including the task token to report progress or completion.	Workers use the `` `SendTaskSuccess` ``, `` `SendTaskFailure` ``, or `` `SendTaskHeartbeat` `` API actions to report task status back to Step Functions.
Use Case	Ideal for integrating with third-party services or custom workflows that need to handle asynchronous operations.	Suitable for distributed systems where tasks can be processed by worker applications running outside of AWS Step Functions.
Complexity	Requires managing task tokens and ensuring external processes can handle token-based communication.	Requires implementing and managing worker applications that can poll for and process tasks.

Both Task Tokens and Activity Tasks provide mechanisms to handle long-running or asynchronous operations within AWS Step Functions, but the choice between them depends on your specific use case and the architecture of your application.

Sample Step function code & how do we define it

AWS Step Functions are defined using JSON-based state machine language. Each state machine consists of states that define tasks or actions to be performed sequentially or in parallel.



```json
{
  "Comment": "A simple AWS Step Functions state machine example",
  "StartAt": "HelloWorld",
  "States": {
    "HelloWorld": {
      "Type": "Pass",
      "Result": "Hello, World!",
      "End": true
    }
  }
}
```

Comment: Optional description of the state machine.

StartAt: The initial state of the state machine (HelloWorld in this case).

States: Contains definitions for all states in the state machine.

- **HelloWorld**: Represents a state named HelloWorld.
    - **Type**: Specifies the type of state (Pass in this case, which just passes the input to the output without doing anything).
    - **Result:** The output of the state.
    - **End:** Indicates that this state is a terminal state (true means it ends the execution).

AWS Step Questions

AWS Step Function Questions

https://lisireddy.medium.com/aws-step-function-questions-301698babb84

AWS Step Function Use case-based questions

https://lisireddy.medium.com/aws-step-function-use-case-based-questions-6c14fd3bc4d5

Wish you the best …! Happy Learning ..!

Yours' Love (@lisireddy across all the platforms)