To understand the basics of Step Functions, we will start with some basic understanding on what the State Machine Model does.

The State Machines model does a simple job – **it uses given states and transitions to complete the tasks at hand**. State Machines are also known as a behavioral model. It's an abstract machine (system) that can be in one state at a time, but it can also switch between a finite number of states. This means **it doesn't allow infinity loops**, which removes one, often very costly, source of errors entirely. The two keywords we need to remember are **States** and **Transitions**.

In short, Finite State Machines are a way of modeling workflows in software systems.

What is Step Function:

Step Functions are a Finite State Machine implementation offered as a serverless service by AWS. Step Functions are made of state machines (workflows) and tasks. Tasks are individual states or single units of work.

With Step Functions, State Machines can be used in our serverless architecture to coordinate different AWS services to form processes that solve our use-cases in a well-defined way.

there are two types of workflows available

1. **Standard (**is a long-running workflow that has to be durable and auditable**)**
2. **Express workflow (**allows more scalability. Is needed for a much higher frequency and event processing volume**)**

These workflows determine how Step Functions performs tasks, integrates with AWS services, and manages pricing. After we create a state machine, we **can't change** its workflow type.

Note on pricing: Express workflow pricing is constructed with more details since users will have to pay for the number of executions, including the duration and memory used for those executions. Standard workflow pricing requires users to pay only for each state transition that occurs.

How does the Step Function help us?

By coordinating multiple AWS services into different serverless workflows, we can quickly build and update the apps. Additionally, with Step Functions, we'll be able to both design and run workflows that'll bring together various services, including Amazon ECS and AWS Lambda, into feature-rich applications.

For example, we can call a Lambda function on each step, but we can also wait for human interactions or external API input. This makes Step Functions a mighty service. And best of all, Step Functions itself is serverless too.

Most importantly, State machines orchestrate the work of AWS services, like Lambda functions. When one function ends, it triggers another function to begin.

Note: To trigger the workflow to start the execution against Step Function API, we can use CloudWatch events as a time trigger or use API Gateway as a proxy.

State Types

There are numerous State types, and all of them have a role to play in the overall workflow:

- **Pass**: Pushes input to output, without performing work. Pass states are useful when constructing and debugging state machines.
- **Task**: Takes input and produces output.
- **Choice**: A Choice state adds branching logic to a state machine based on the input. Choice rules can implement 16 different comparison operators, and can be combined using And, Or, and Not
- **Wait**: A Wait state delays the state machine from continuing for a specified time.
- **Success**: Has an expected dead-end that stops execution successfully.
- **Fail**: Has an expected dead-end that stops execution with a failure.
- **Parallel**: Allows a user to implement parallel branches in execution, meaning the user can start multiple states at once.
- (Dynamic) **Mapping**: Runs a set of steps for every input item.

Tasks

Tasks are the leading States in which all the work is done. **Tasks can call Activities** (remote executions):

- Call an execution on either ECS, EC2 machines, or mobile devices.
- Sending SMS notifications and wait for the input.

Error Handling

Error handling includes retries and catches. Error handling is critical because if Parallel tasks execute successfully, but one fails, the entire execution will fail. However, even if the entire execution fails, the state changes will remain intact.

Error handling allows us to track everything that's happened in the log, and by doing so, we'll have a better insight on why some errors happened so we could handle the core problem.

If our Lambda function throws an error, the task it belongs to will fail. Error catch and error handling are essential for Step Functions since it allows for a successful, and error-free function execution.

Use Cases

Step Functions Standard workflow is excellent for business-critical workflows and brings along numerous business benefits. It provides much better error handling logic than Lambda Functions, while it's relatively easy to orchestrate them. On the other hand, it's meant more for business-critical ones since it is pretty expensive compared to Express workflow.

Complex workflow allows us to handle a tremendous amount of states. Complex workflow is excellent for orchestrating microservices since we won't need to build a connection between them, and we can call out different languages from different services.

Step Functions are also beneficial for long-running or delayed workflows. It allows us to have a workflow for up to a year while also implementing the waiting state.

One of the best use practices of Step Functions is for large payloads. By putting payloads in S3 and importing them to Step Functions, we'll be good to go. If we don't, our workflow might fail.

Using timeouts will help us avoid stuck executions since there are no default timeouts in Step Function tasks. Moreover, Step Functions rely on the activity worker's response.

Note: Lambda can have very short-lived service errors. This is why it's good to add Lambda service exceptions since it's excellent at handling these exceptions proactively.

Possible Integrations

We can integrate them from the Tasks:

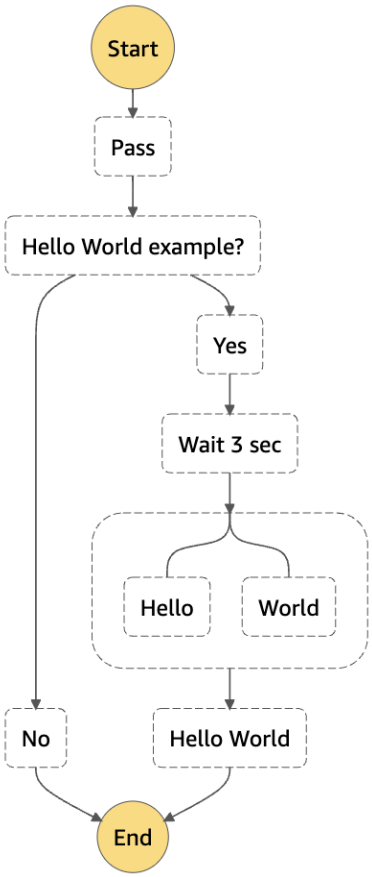
Dev Tools

The AWS CDK has a Step Functions module that allows us to define our workflows directly in our CDK stack, with static type checks and everything.

AWS provides a Step Functions plugin that's used in the Serverless framework. It allows us to do everything Step Functions can do, while it helps devs take care of the rows and many other things we need to define.

It's possible to download Step Functions as a .jar file or a Docker image so we can run it on our machine.

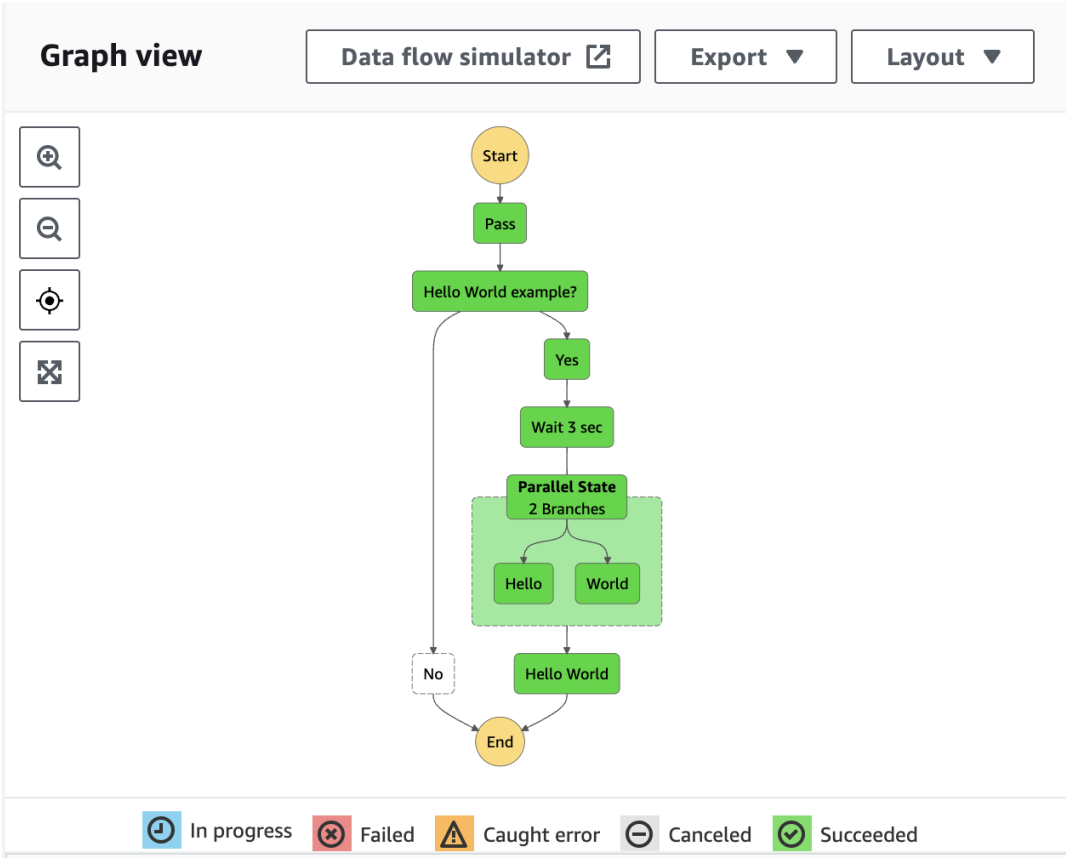
Example:

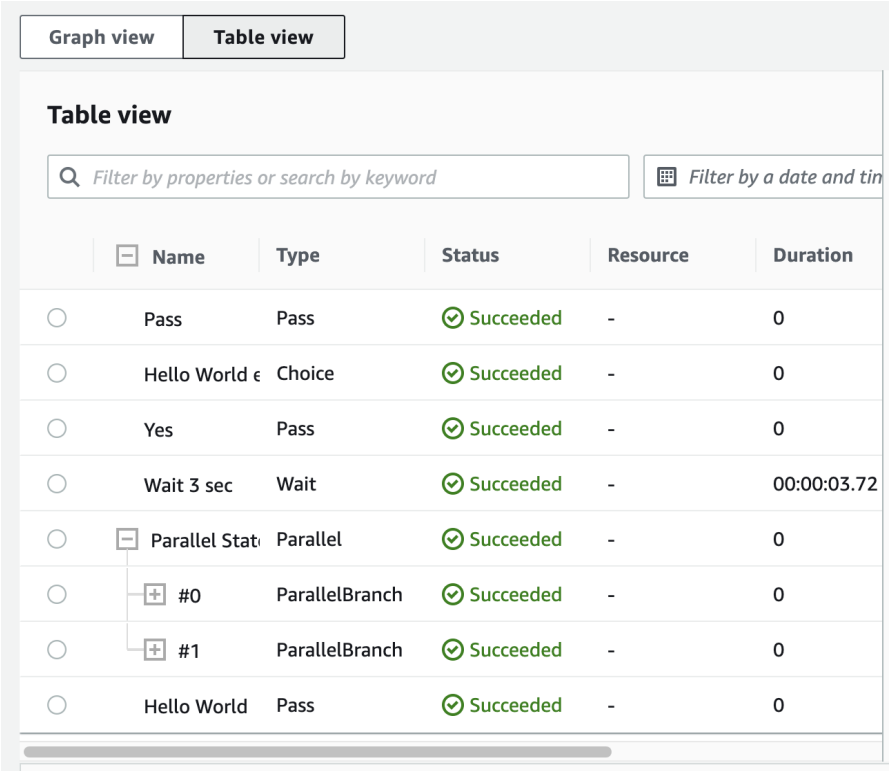



```
{
  "Comment": "A Hello World example demonstrating various state types of the
Amazon States Language",
  "StartAt": "Pass",
  "States": {
    "Pass": {
      "Comment": "A Pass state passes its input to its output, without
performing work. Pass states are useful when constructing and debugging state
machines.",
      "Type": "Pass",
      "Next": "Hello World example?"
    },
    "Hello World example?": {
      "Comment": "A Choice state adds branching logic to a state machine.
Choice rules can implement 16 different comparison operators, and can be
combined using And, Or, and Not",
```

```
      "Type": "Choice",
      "Choices": [
        {
          "Variable": "$.IsHelloWorldExample",
          "BooleanEquals": true,
          "Next": "Yes"
        },
        {
          "Variable": "$.IsHelloWorldExample",
          "BooleanEquals": false,
          "Next": "No"
        }
      ],
      "Default": "Yes"
    },
    "Yes": {
      "Type": "Pass",
      "Next": "Wait 3 sec"
    },
    "No": {
      "Type": "Fail",
      "Cause": "Not Hello World"
    },
    "Wait 3 sec": {
      "Comment": "A Wait state delays the state machine from continuing for a
specified time.",
      "Type": "Wait",
      "Seconds": 3,
      "Next": "Parallel State"
    },
    "Parallel State": {
      "Comment": "A Parallel state can be used to create parallel branches of
execution in your state machine.",
      "Type": "Parallel",
      "Next": "Hello World",
      "Branches": [
        {
          "StartAt": "Hello",
          "States": {
            "Hello": {
              "Type": "Pass",
              "End": true
            }
          }
        },
```

```
        {
          "StartAt": "World",
          "States": {
            "World": {
              "Type": "Pass",
              "End": true
            }
          }
        }
      ]
    },
    "Hello World": {
      "Type": "Pass",
      "End": true
    }
  }
}
```




Graph view
Table view
Table view
Filter by properties or search by keyword
Filter by a date and tin
Name
Type
Status
Resource
Duration
Pass
Pass
Succeeded
-
0
Hello World e
Choice
Succeeded
-
0
Yes
Pass
Succeeded
-
0
Wait 3 sec
Wait
Succeeded
-
00:00:03.72
Parallel Stat
Parallel
Succeeded
-
0
#0
ParallelBranch
Succeeded
-
0
#1
ParallelBranch
Succeeded
-
0
Hello World
Pass
Succeeded
-
0

Ref:
https://dashbird.io/blog/ultimate-guide-aws-step-functions/
https://docs.aws.amazon.com/step-functions/latest/dg/getting-started.html
https://docs.aws.amazon.com/step-functions/latest/dg/concepts-states.html
https://docs.aws.amazon.com/step-functions/latest/dg/sfn-prerequisites.html