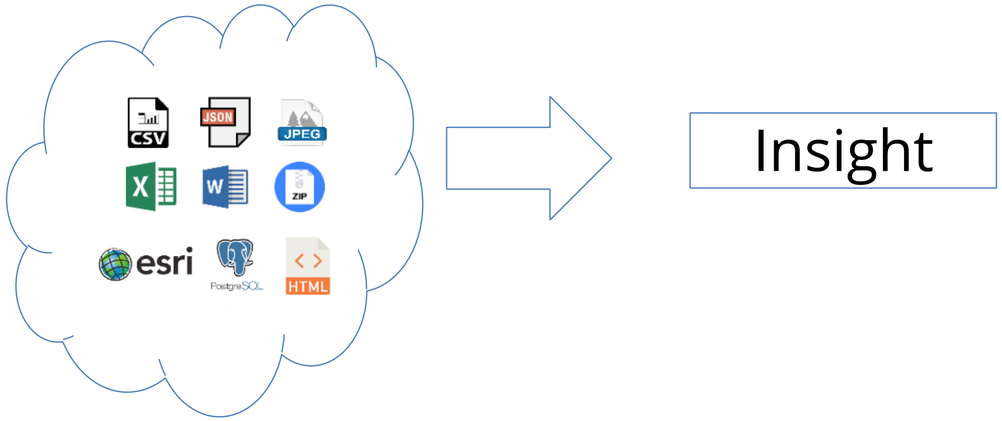
Working with large numbers of non-trivial ETL pipelines.

OR I'VE ALREADY STUFFED IT UP SO YOU DON'T HAVE TO

JESSICA FLANAGAN

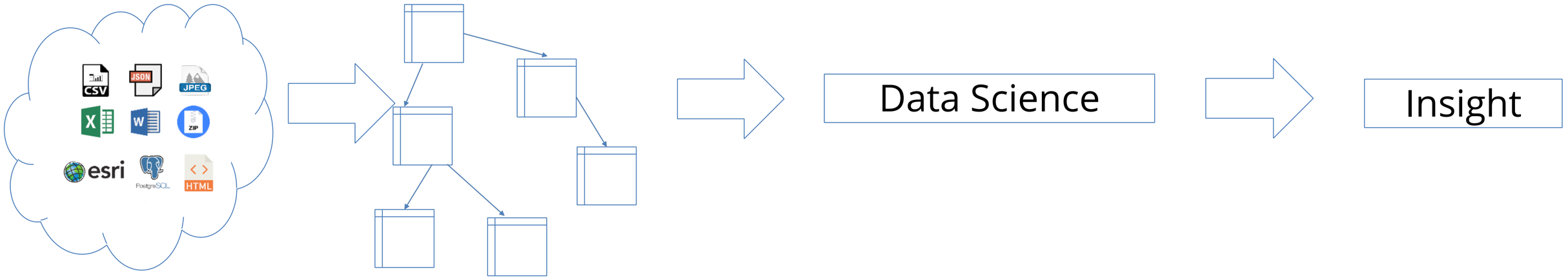
Our Business Model



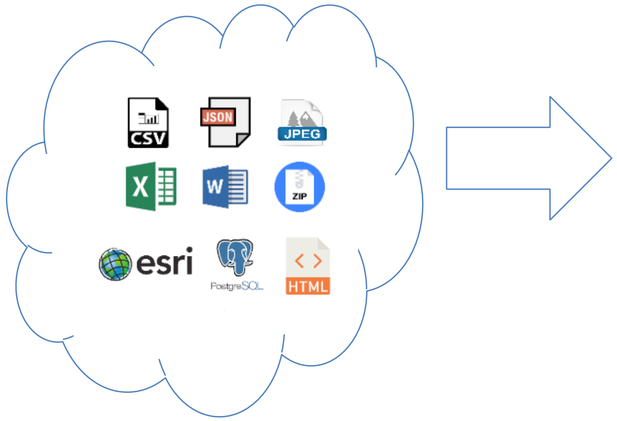
Insight

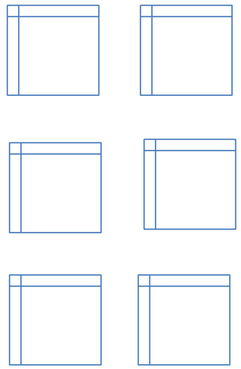
Data Science

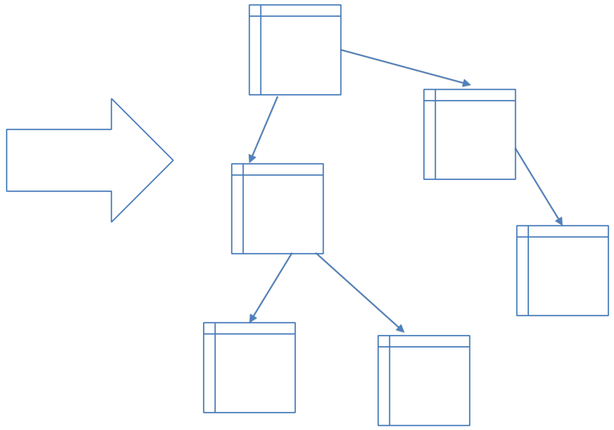
Insight


CSV
JSON
JPEG
ZIP
esri
PostgreSQL
HTML
Data Science
Insight

Our Basic Workflow







Problem Space

What is a non trivial ETL pipeline?

- If you just need to trigger a lambda when you put something in S3 you don't need a pipeline.
- This talk is about batch pipelines not streaming although some concepts may apply.

What is a large number of pipelines?

- Without proper monitoring you have to treat pipelines like children deciding how many you can/want to look after is an exercise for the reader.

Architecture Goals

Create Pipelines that are easy to debug. When they fail we need to see where the failure occurred an any related logging.

Architecture Goals

Create Pipelines that are easy to debug. When they fail we need to see where the failure occurred an any related logging.

We need a whole of system view so we can can see that our pipelines are running and which succeed or fail.

Architecture Goals

Create Pipelines that are easy to debug. When they fail we need to see where the failure occurred an any related logging.

Monitoring so we can see if our pipelines are degrading or producing less results over time.

We need a whole of system view so we can can see that our pipelines are running and which succeed or fail.

Architecture Goals

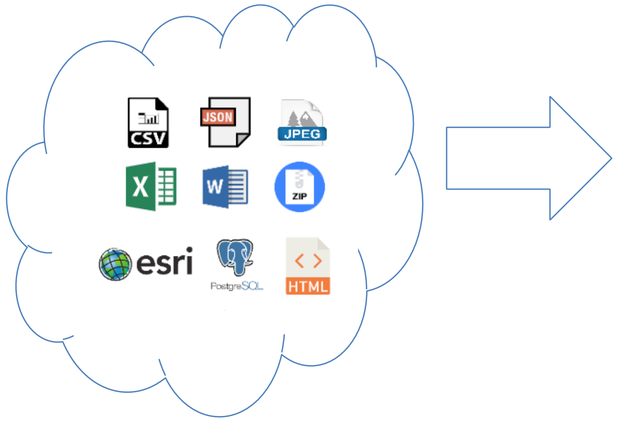
Create Pipelines that are easy to debug. When they fail we need to see where the failure occurred an any related logging.

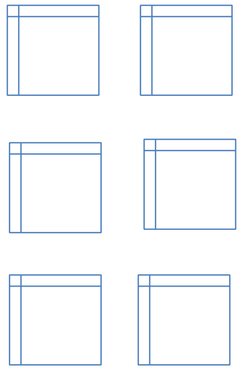
Monitoring so we can see if our pipelines are degrading or producing less results over time.

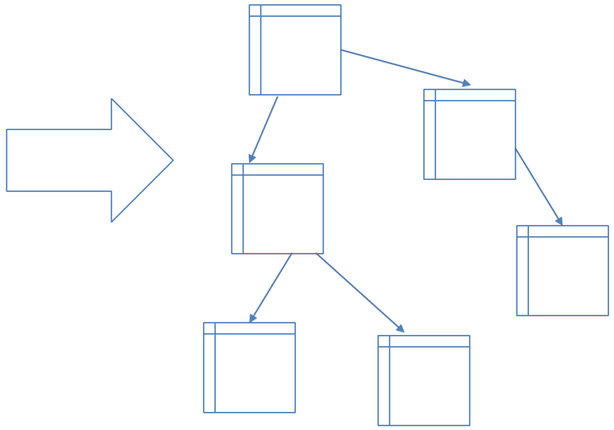
We need a whole of system view so we can can see that our pipelines are running and which succeed or fail.

A way of making sure that if our pipelines fail that someone is tasked with fixing the problem.

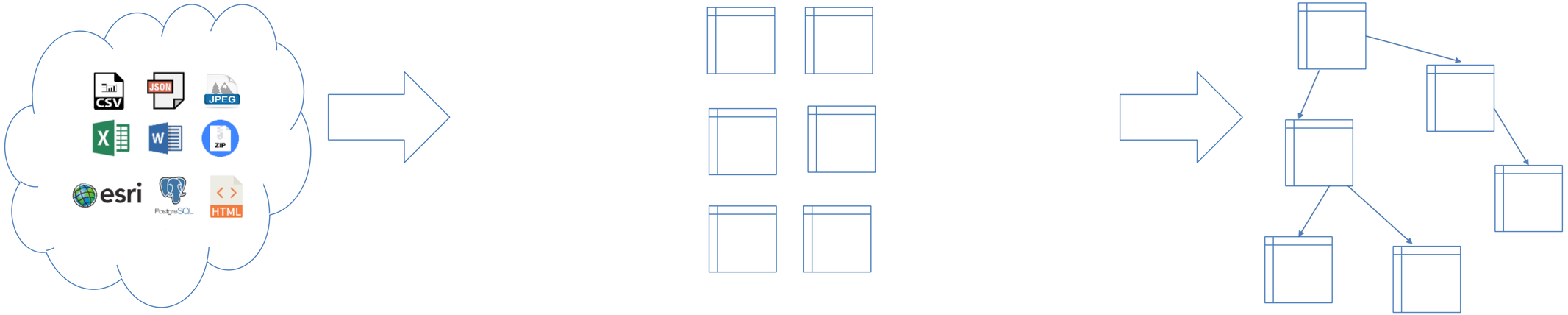
Our Basic Workflow



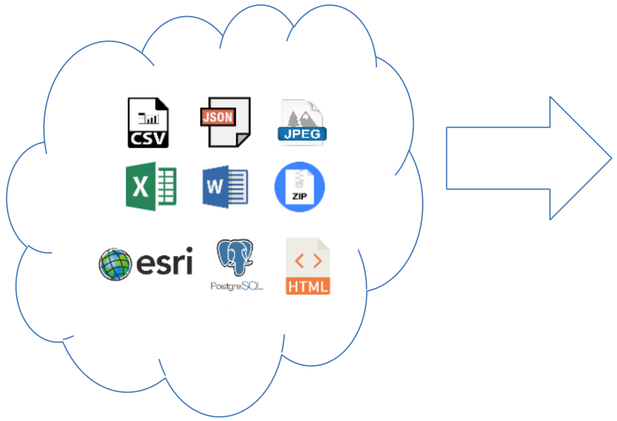




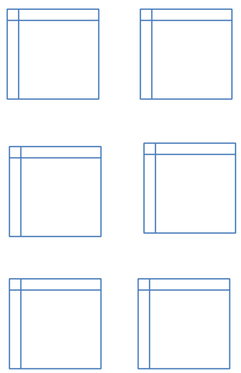
Extract

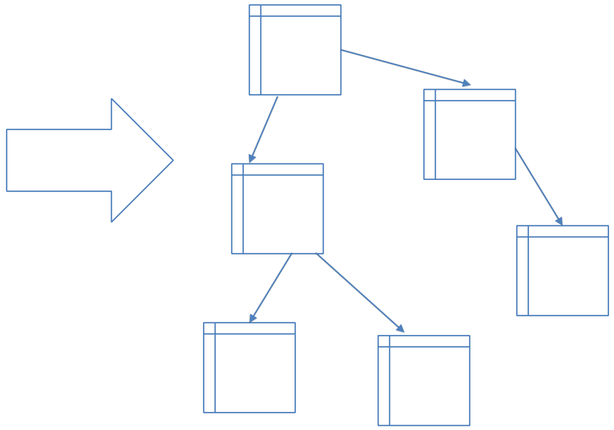


Extract

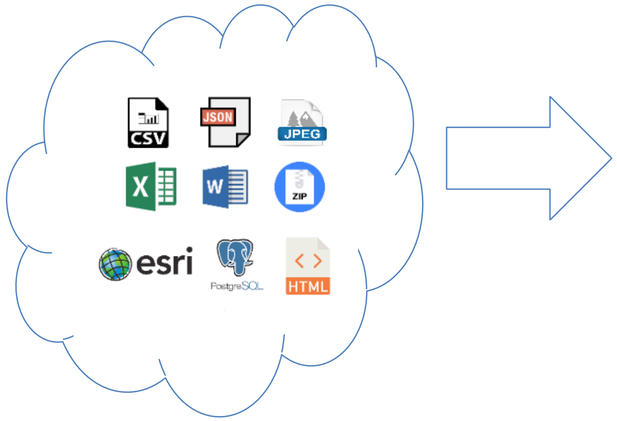


Transform

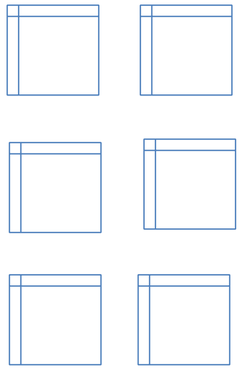




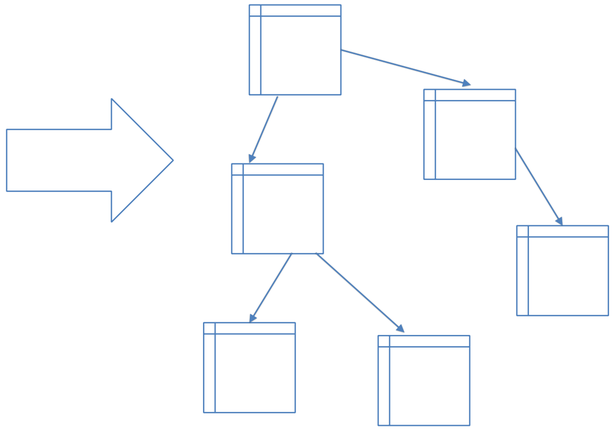
Extract



Transform

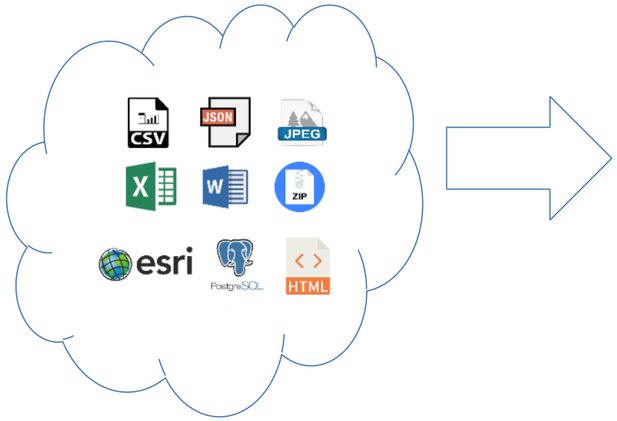


Load

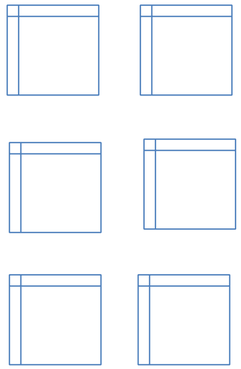


Wherever possible show your working. Allow data scientists to access the data before and after transformation.

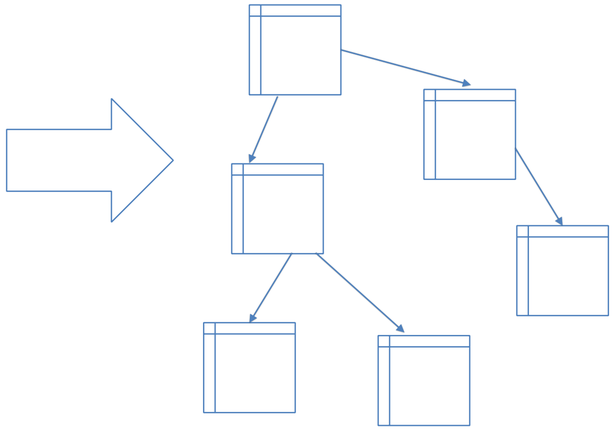
Extract



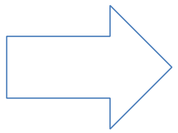
Transform



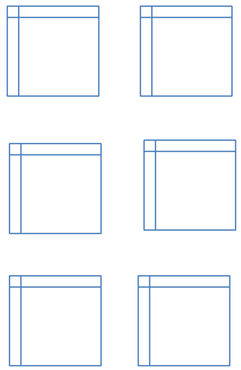
Load

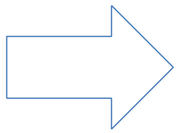


Extract

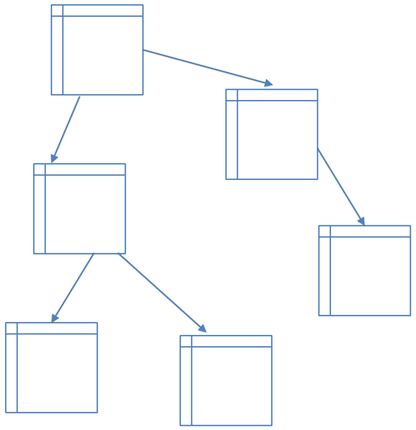


Transform





Load



S3 - Extract

Extract

Transform

Load

docker

OR

AWS Lambda

S3 - Extract

docker

OR

AWS Lambda

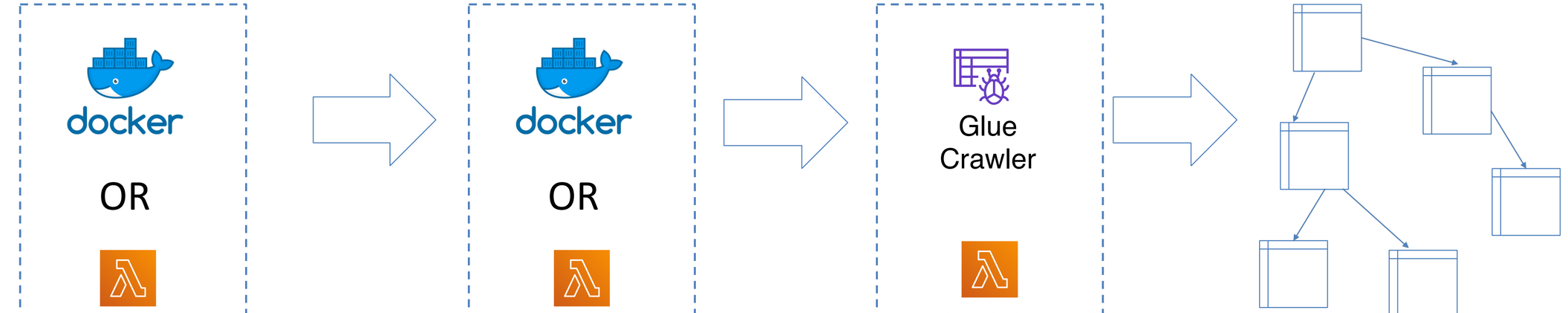
S3 - Structured

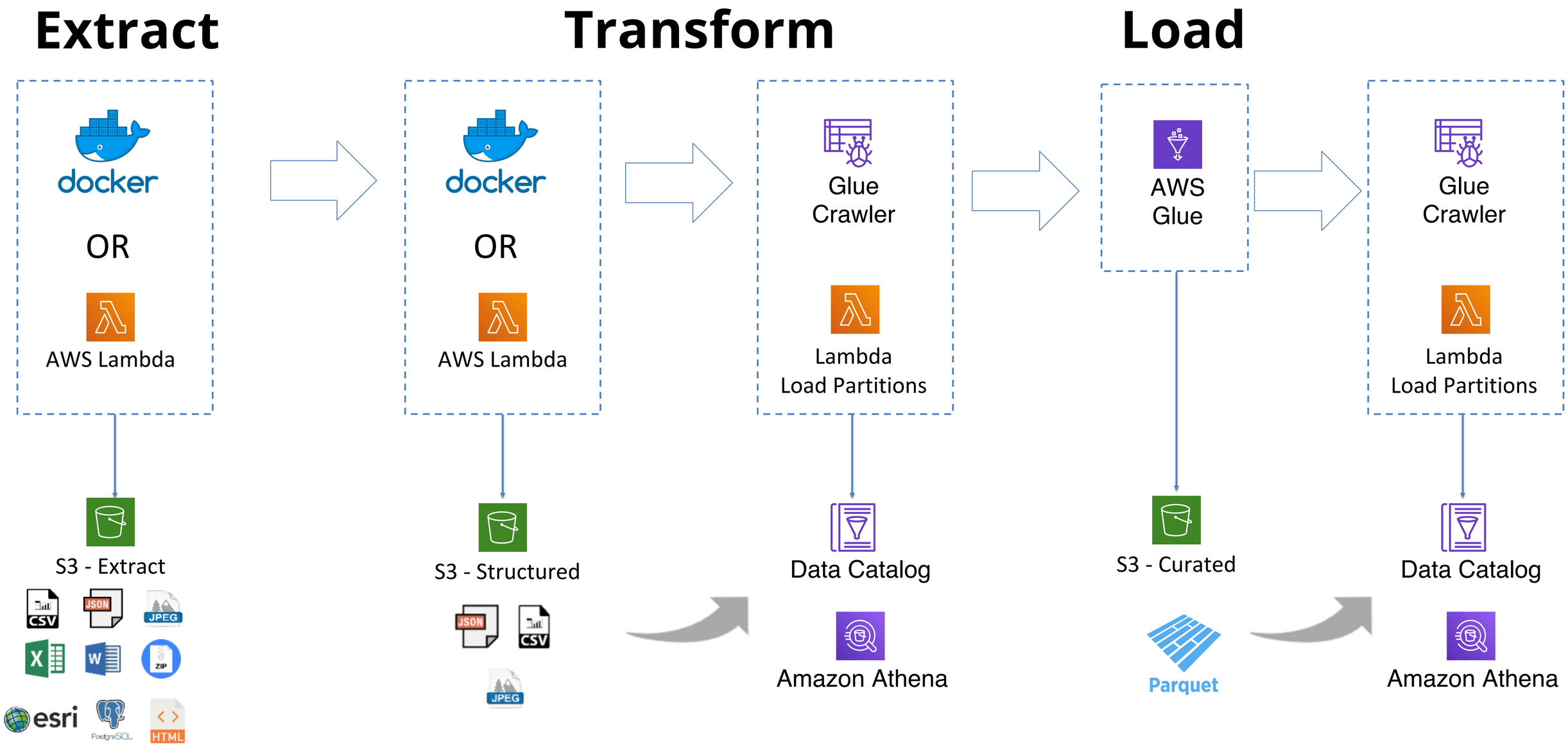
Glue
Crawler

Lambda
Load Partitions

Data Catalog

Amazon Athena




Extract
Transform
Load
docker
OR
AWS Lambda
docker
OR
AWS Lambda
Glue
Crawler
Lambda
Load Partitions
AWS
Glue
Glue
Crawler
Lambda
Load Partitions
S3 - Extract
S3 - Structured
Data Catalog
S3 - Curated
Data Catalog
Amazon Athena
Parquet
Amazon Athena

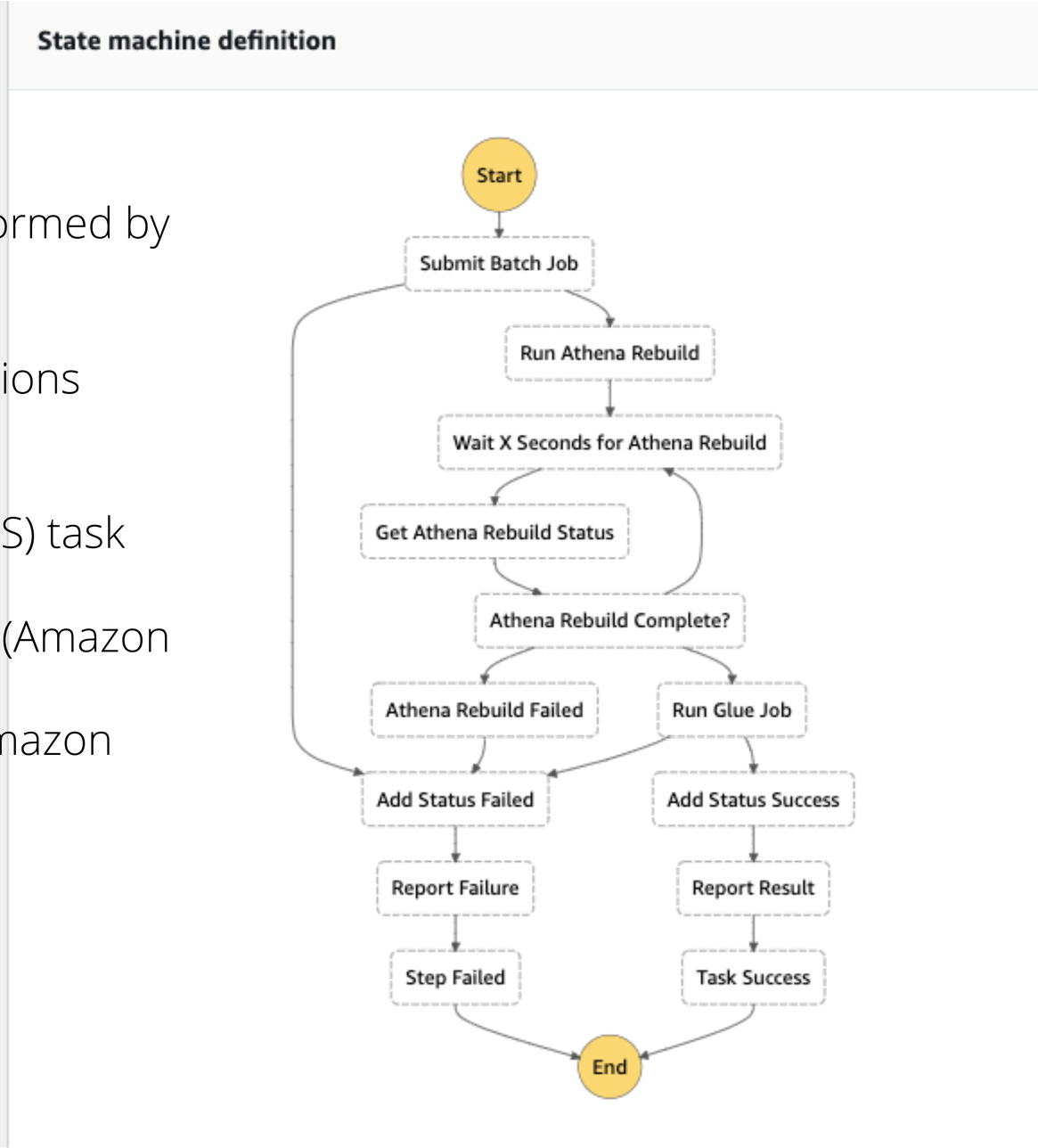
Don’t put extract in the same s3 bucket as the other stages it makes it hard to set different storage and backup options.

Data Pipeline Tools Comparison

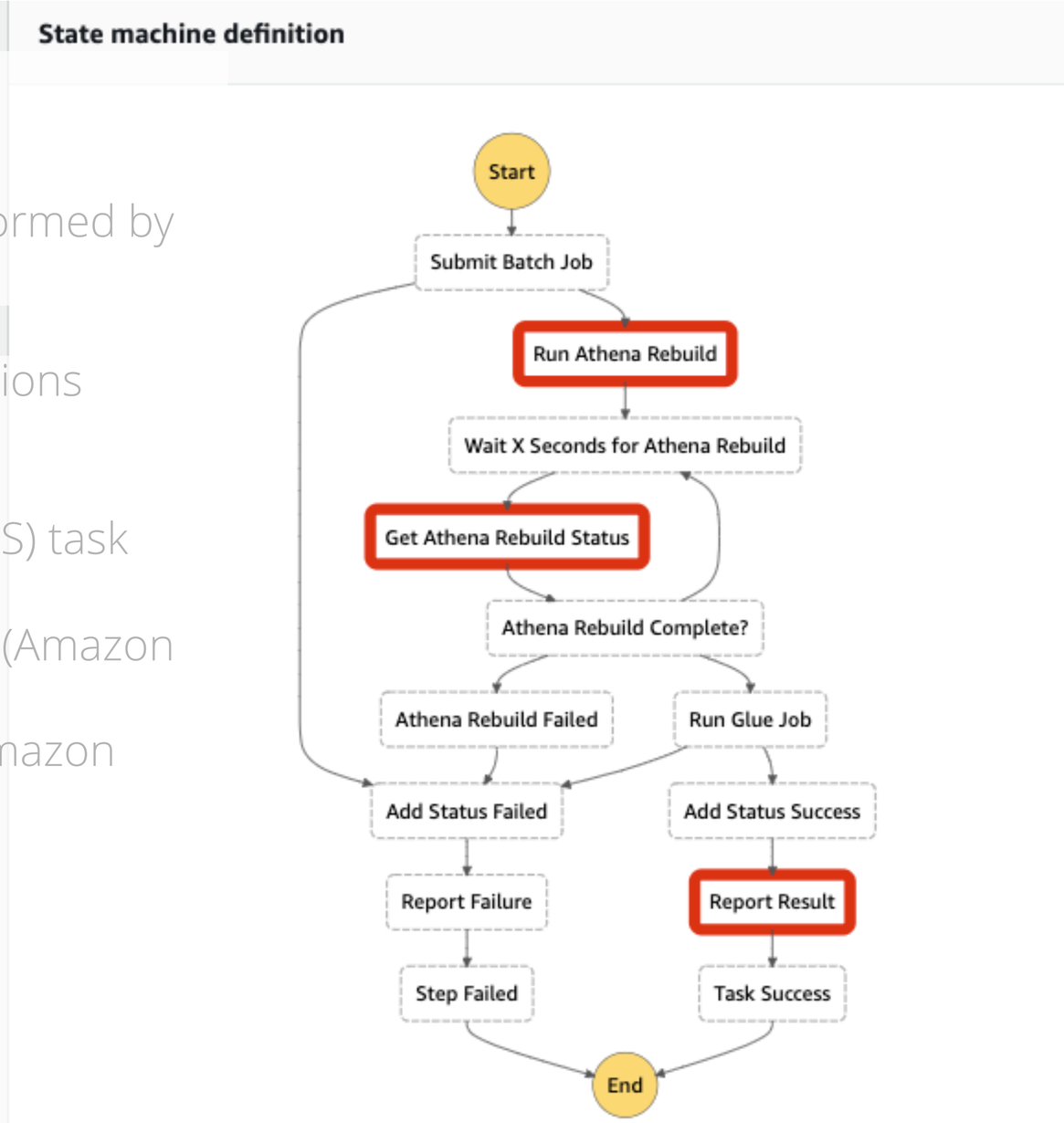
	Data Specific	Flexible
AWS	AWS Data pipelines	AWS Step Functions
Non AWS	Apache Nifi Azkaban Oozie	Apache Airflow Luigi

AWS Step Functions

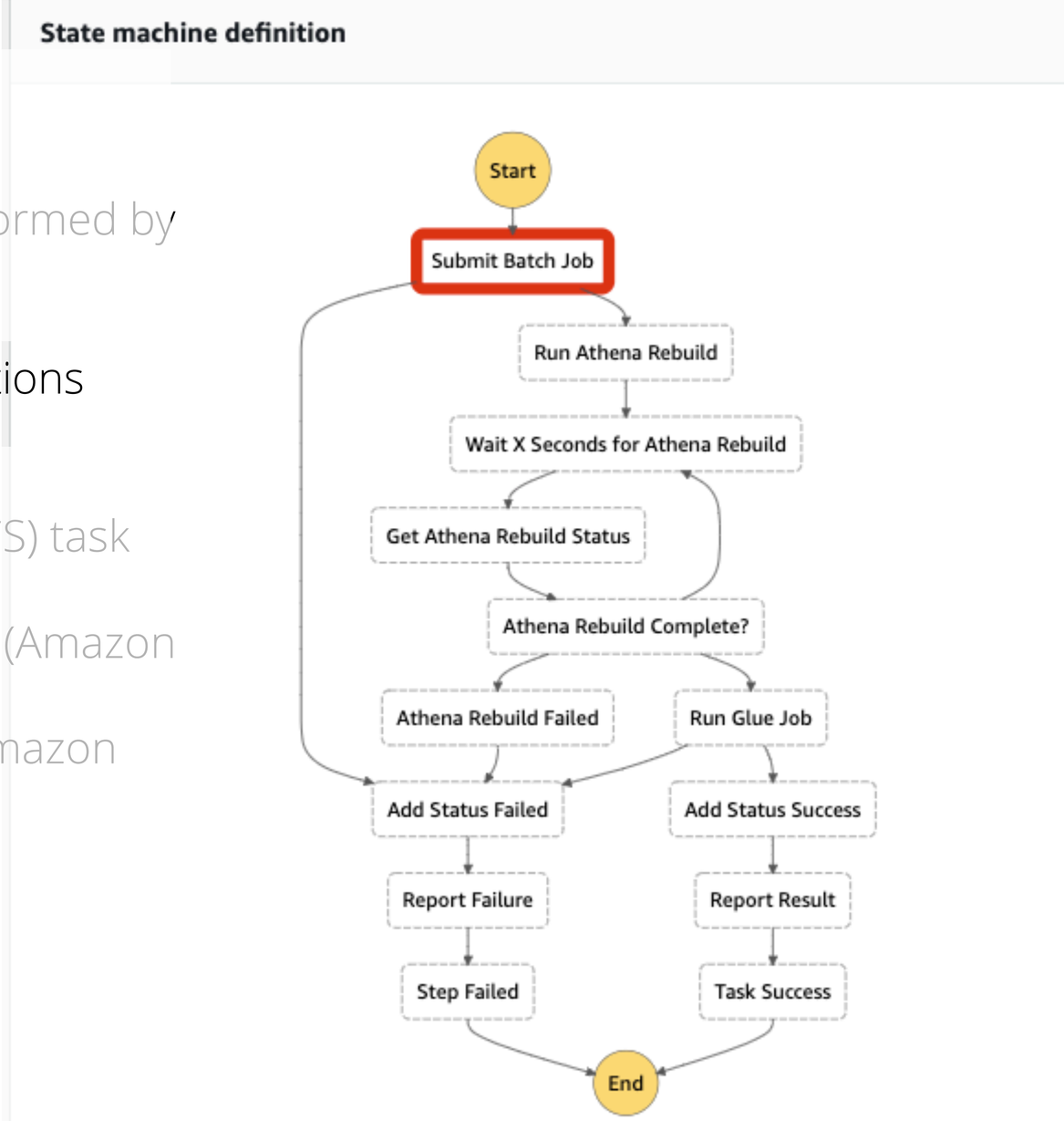
- Tasks - A Task state represents a single unit of work performed by a state machine.
  - Invoke an AWS Lambda function.
  - Run an AWS Batch job and then perform different actions based on the results.
  - Insert or get an item from Amazon DynamoDB.
  - Run an Amazon Elastic Container Service (Amazon ECS) task and wait for it to complete.
  - Publish a topic in Amazon Simple Notification Service (Amazon SNS).
  - Send a message in Amazon Simple Queue Service (Amazon SQS).
  - Manage a job for AWS Glue or Amazon SageMaker.
- Pass - Passes input to output without doing any work.
- Wait - Waits a x seconds before continuing
- Succeed - Terminates successfully
- Fail - Terminates state machine unsuccessfully
- Choice - Applies branching logic
- Parallel - Performs tasks in parallel



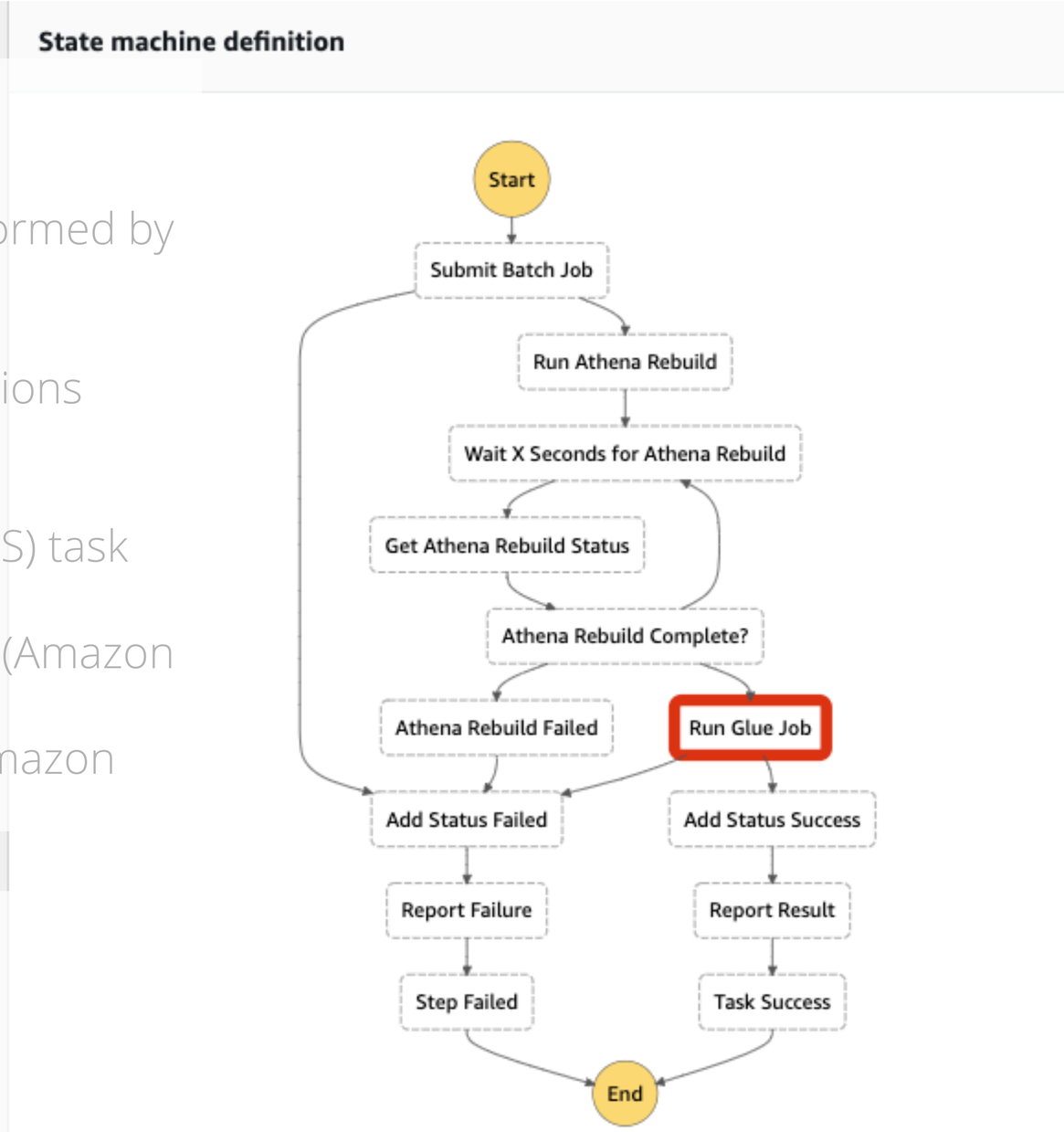
- Tasks - A Task state represents a single unit of work performed by a state machine.
  - Invoke an AWS Lambda function.
  - Run an AWS Batch job and then perform different actions based on the results.
  - Insert or get an item from Amazon DynamoDB.
  - Run an Amazon Elastic Container Service (Amazon ECS) task and wait for it to complete.
  - Publish a topic in Amazon Simple Notification Service (Amazon SNS).
  - Send a message in Amazon Simple Queue Service (Amazon SQS).
  - Manage a job for AWS Glue or Amazon SageMaker.
- Pass - Passes input to output without doing any work.
- Wait - Waits a x seconds before continuing
- Succeed - Terminates successfully
- Fail - Terminates state machine unsuccessfully
- Choice - Applies branching logic
- Parallel - Performs tasks in parallel



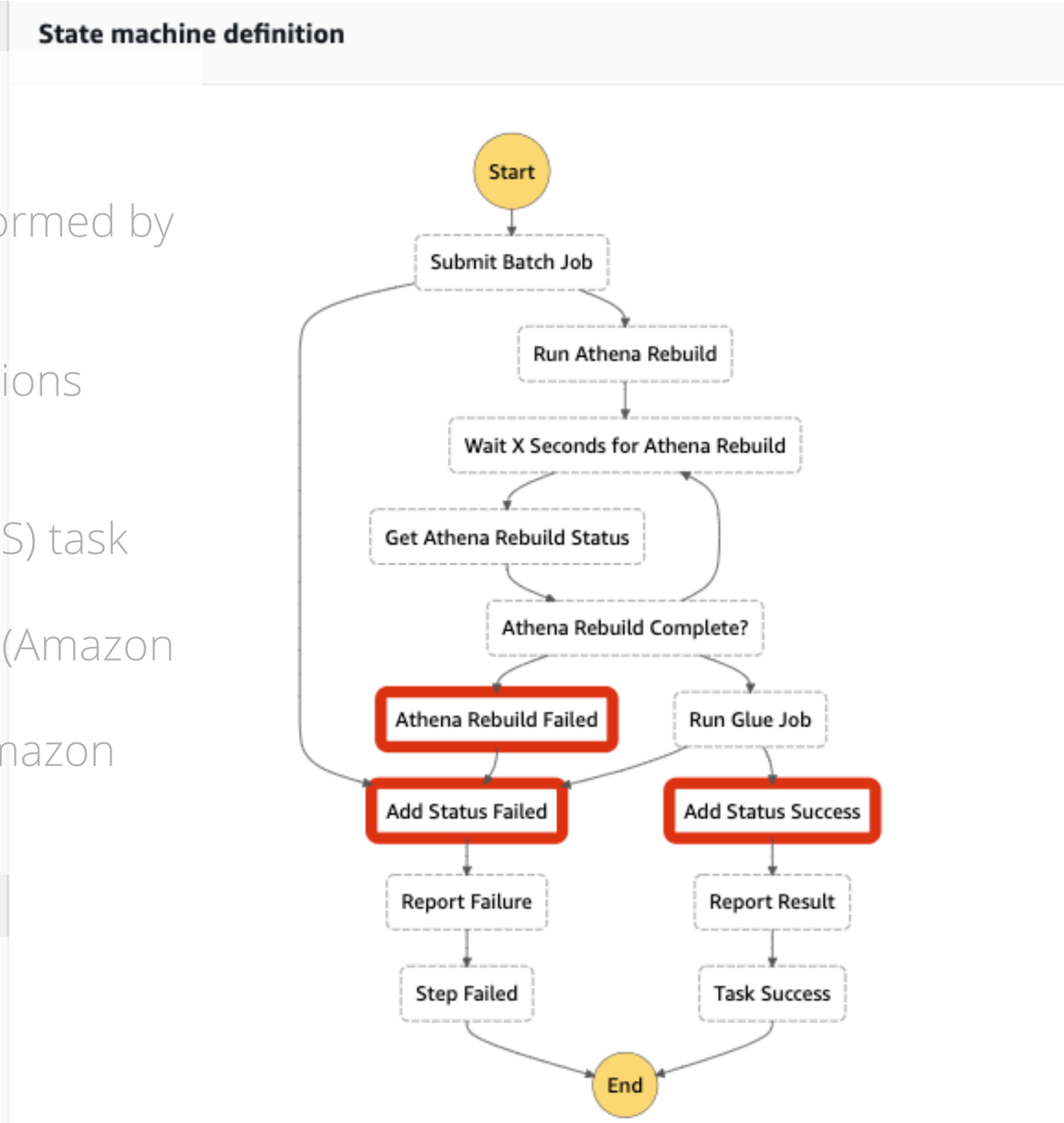
- Tasks - A Task state represents a single unit of work performed by a state machine.
  - Invoke an AWS Lambda function.
  - Run an AWS Batch job and then perform different actions based on the results.
  - Insert or get an item from Amazon DynamoDB.
  - Run an Amazon Elastic Container Service (Amazon ECS) task and wait for it to complete.
  - Publish a topic in Amazon Simple Notification Service (Amazon SNS).
  - Send a message in Amazon Simple Queue Service (Amazon SQS).
  - Manage a job for AWS Glue or Amazon SageMaker.
- Pass - Passes input to output without doing any work.
- Wait - Waits a x seconds before continuing
- Succeed - Terminates successfully
- Fail - Terminates state machine unsuccessfully
- Choice - Applies branching logic
- Parallel - Performs tasks in parallel



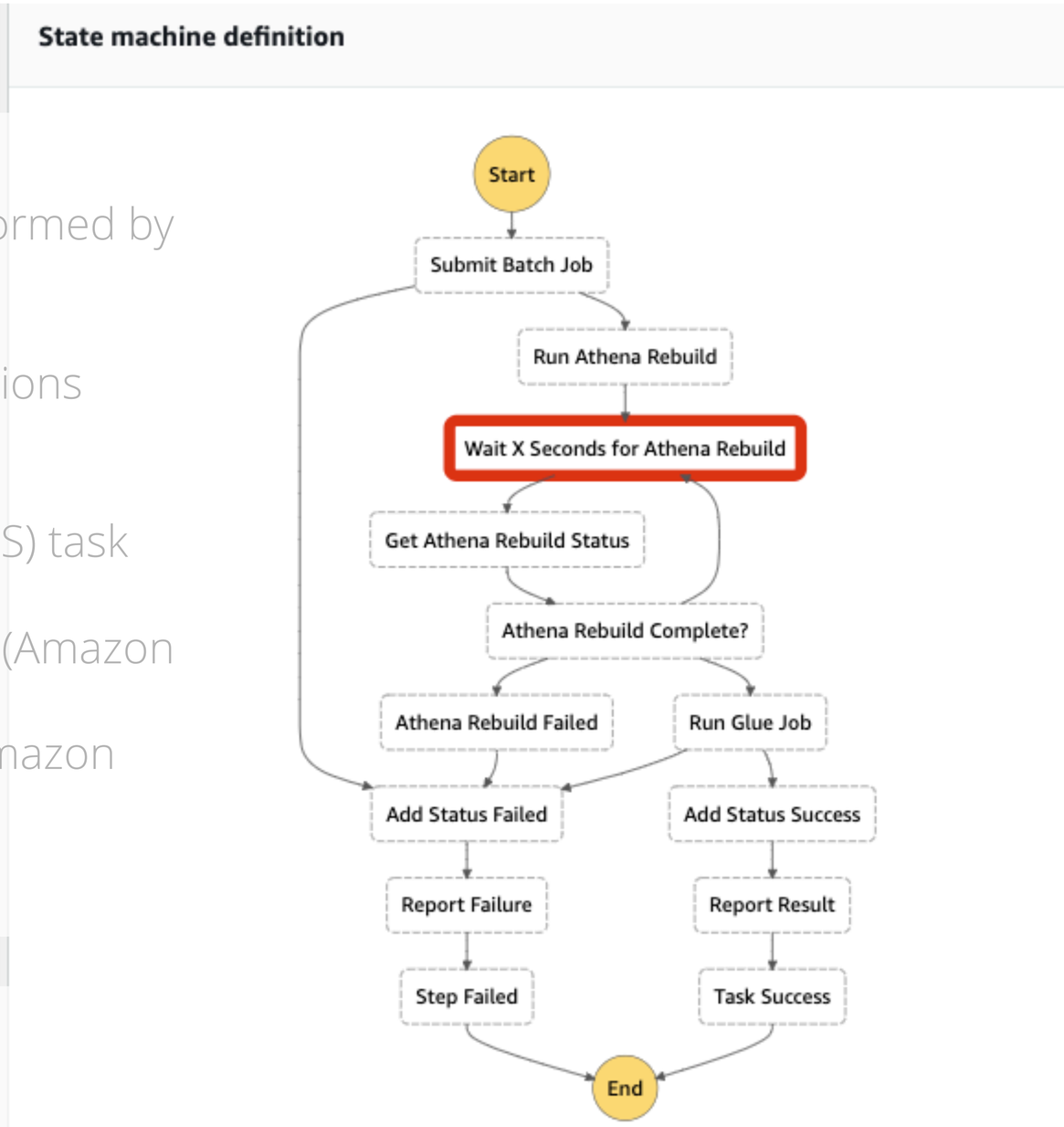
- Tasks - A Task state represents a single unit of work performed by a state machine.
  - Invoke an AWS Lambda function.
  - Run an AWS Batch job and then perform different actions based on the results.
  - Insert or get an item from Amazon DynamoDB.
  - Run an Amazon Elastic Container Service (Amazon ECS) task and wait for it to complete.
  - Publish a topic in Amazon Simple Notification Service (Amazon SNS).
  - Send a message in Amazon Simple Queue Service (Amazon SQS).
  - Manage a job for AWS Glue or Amazon SageMaker.
- Pass - Passes input to output without doing any work.
- Wait - Waits a x seconds before continuing
- Succeed - Terminates successfully
- Fail - Terminates state machine unsuccessfully
- Choice - Applies branching logic
- Parallel - Performs tasks in parallel



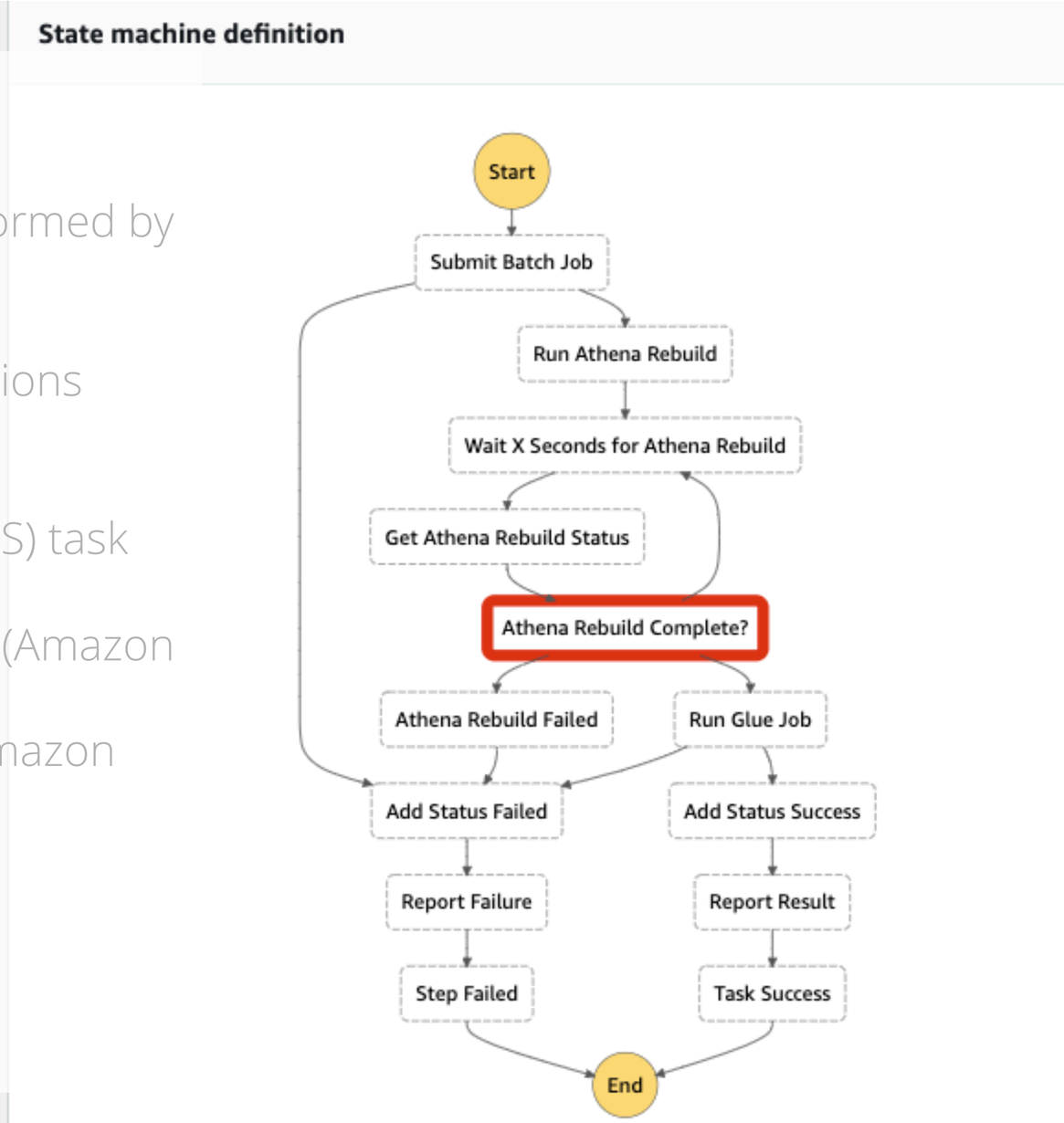
- Tasks - A Task state represents a single unit of work performed by a state machine.
  - Invoke an AWS Lambda function.
  - Run an AWS Batch job and then perform different actions based on the results.
  - Insert or get an item from Amazon DynamoDB.
  - Run an Amazon Elastic Container Service (Amazon ECS) task and wait for it to complete.
  - Publish a topic in Amazon Simple Notification Service (Amazon SNS).
  - Send a message in Amazon Simple Queue Service (Amazon SQS).
  - Manage a job for AWS Glue or Amazon SageMaker.
- Pass - Passes input to output without doing any work.
- Wait - Waits a x seconds before continuing
- Succeed - Terminates successfully
- Fail - Terminates state machine unsuccessfully
- Choice - Applies branching logic
- Parallel - Performs tasks in parallel



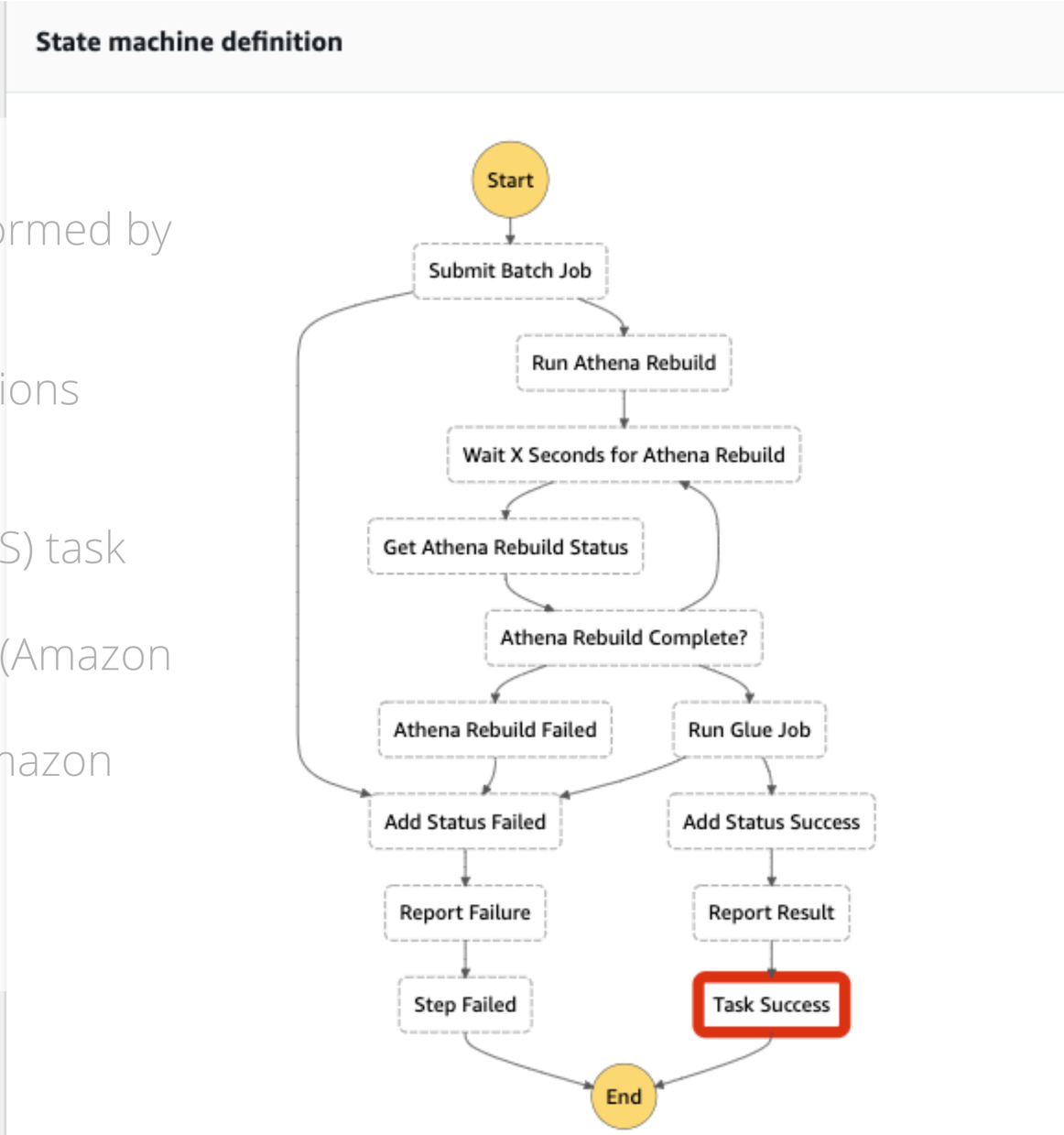
- Tasks - A Task state represents a single unit of work performed by a state machine.
  - Invoke an AWS Lambda function.
  - Run an AWS Batch job and then perform different actions based on the results.
  - Insert or get an item from Amazon DynamoDB.
  - Run an Amazon Elastic Container Service (Amazon ECS) task and wait for it to complete.
  - Publish a topic in Amazon Simple Notification Service (Amazon SNS).
  - Send a message in Amazon Simple Queue Service (Amazon SQS).
  - Manage a job for AWS Glue or Amazon SageMaker.
- Pass - Passes input to output without doing any work.
- Wait - Waits a x seconds before continuing
- Succeed - Terminates successfully
- Fail - Terminates state machine unsuccessfully
- Choice - Applies branching logic
- Parallel - Performs tasks in parallel



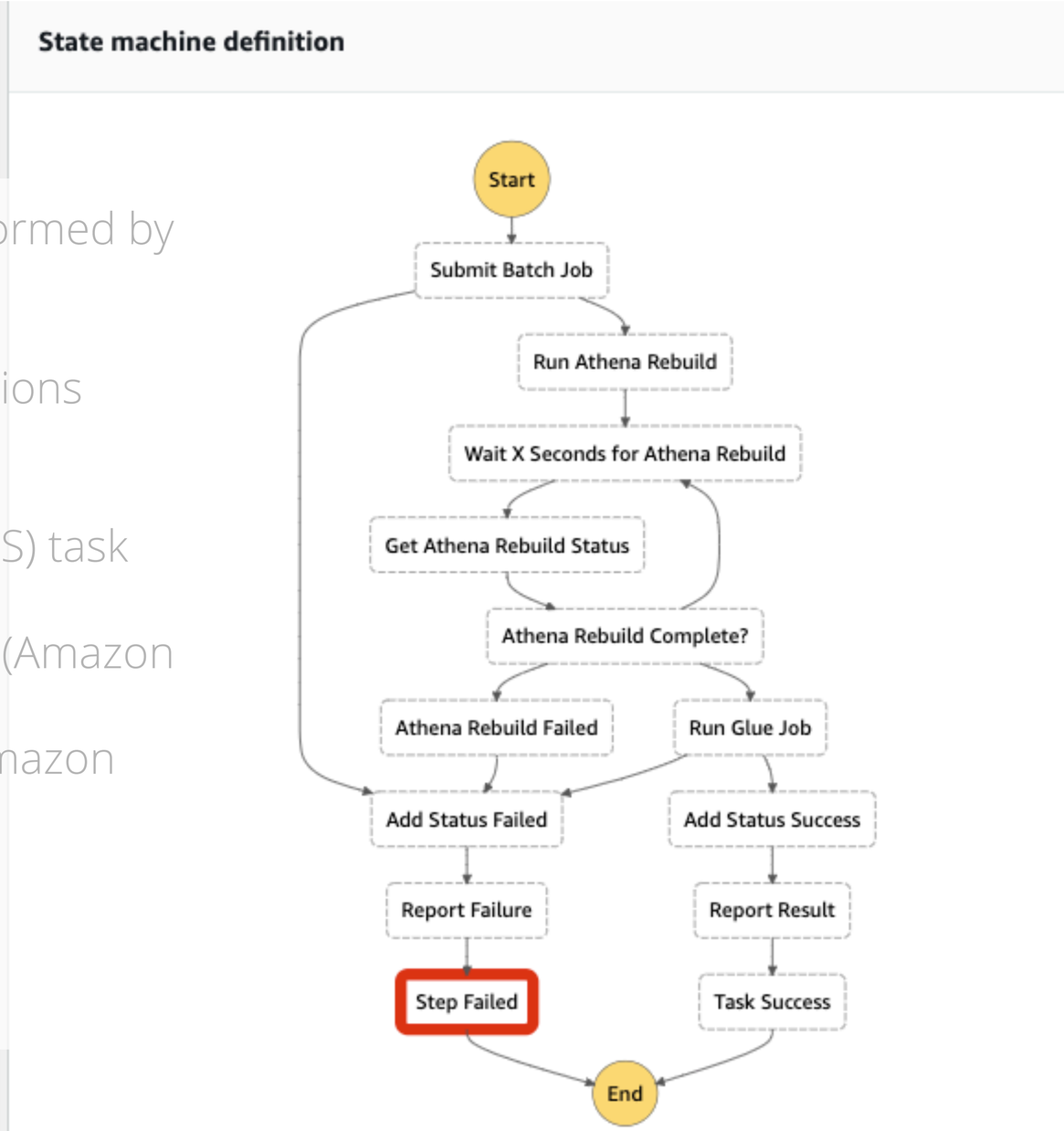
- Tasks - A Task state represents a single unit of work performed by a state machine.
  - Invoke an AWS Lambda function.
  - Run an AWS Batch job and then perform different actions based on the results.
  - Insert or get an item from Amazon DynamoDB.
  - Run an Amazon Elastic Container Service (Amazon ECS) task and wait for it to complete.
  - Publish a topic in Amazon Simple Notification Service (Amazon SNS).
  - Send a message in Amazon Simple Queue Service (Amazon SQS).
  - Manage a job for AWS Glue or Amazon SageMaker.
- Pass - Passes input to output without doing any work.
- Wait - Waits a x seconds before continuing
- Succeed - Terminates successfully
- Fail - Terminates state machine unsuccessfully
- Choice - Applies branching logic
- Parallel - Performs tasks in parallel



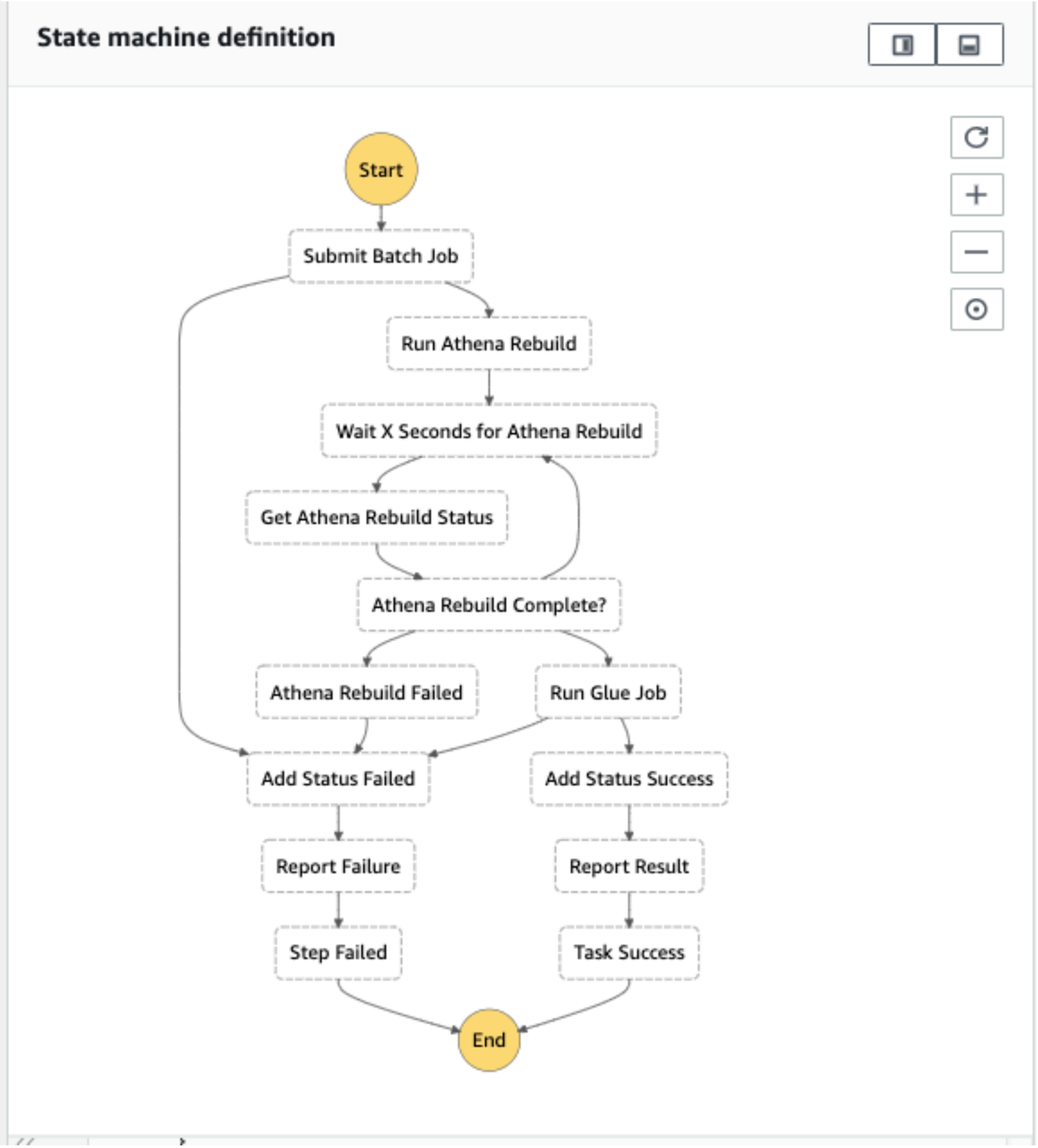
- Tasks - A Task state represents a single unit of work performed by a state machine.
  - Invoke an AWS Lambda function.
  - Run an AWS Batch job and then perform different actions based on the results.
  - Insert or get an item from Amazon DynamoDB.
  - Run an Amazon Elastic Container Service (Amazon ECS) task and wait for it to complete.
  - Publish a topic in Amazon Simple Notification Service (Amazon SNS).
  - Send a message in Amazon Simple Queue Service (Amazon SQS).
  - Manage a job for AWS Glue or Amazon SageMaker.
- Pass - Passes input to output without doing any work.
- Wait - Waits a x seconds before continuing
- Succeed - Terminates successfully
- Fail - Terminates state machine unsuccessfully
- Choice - Applies branching logic
- Parallel - Performs tasks in parallel



- Tasks - A Task state represents a single unit of work performed by a state machine.
  - Invoke an AWS Lambda function.
  - Run an AWS Batch job and then perform different actions based on the results.
  - Insert or get an item from Amazon DynamoDB.
  - Run an Amazon Elastic Container Service (Amazon ECS) task and wait for it to complete.
  - Publish a topic in Amazon Simple Notification Service (Amazon SNS).
  - Send a message in Amazon Simple Queue Service (Amazon SQS).
  - Manage a job for AWS Glue or Amazon SageMaker.
- Pass - Passes input to output without doing any work.
- Wait - Waits a x seconds before continuing
- Succeed - Terminates successfully
- Fail - Terminates state machine unsuccessfully
- Choice - Applies branching logic
- Parallel - Performs tasks in parallel



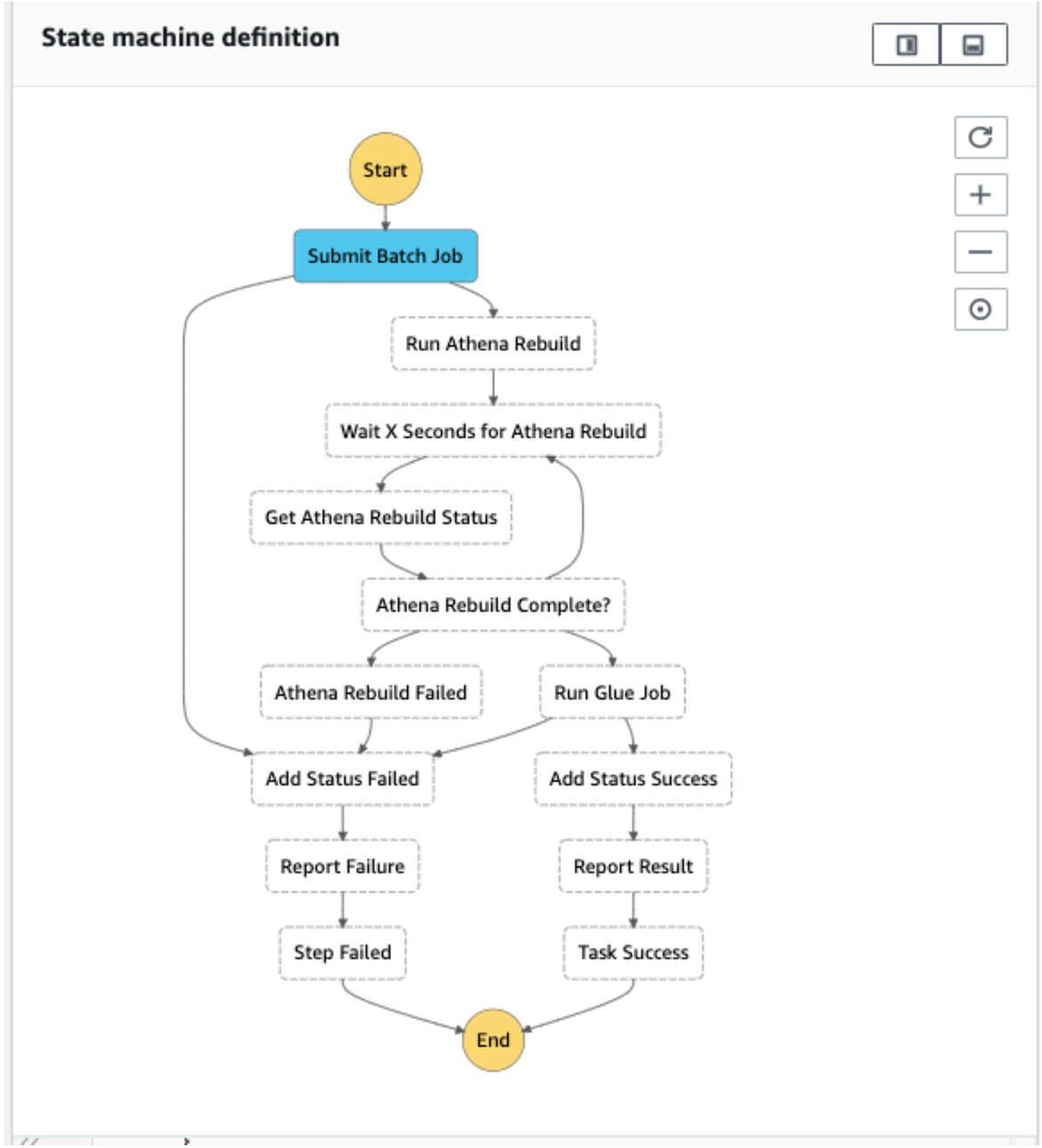
Step Function Transitions



Extract/Transform

Load

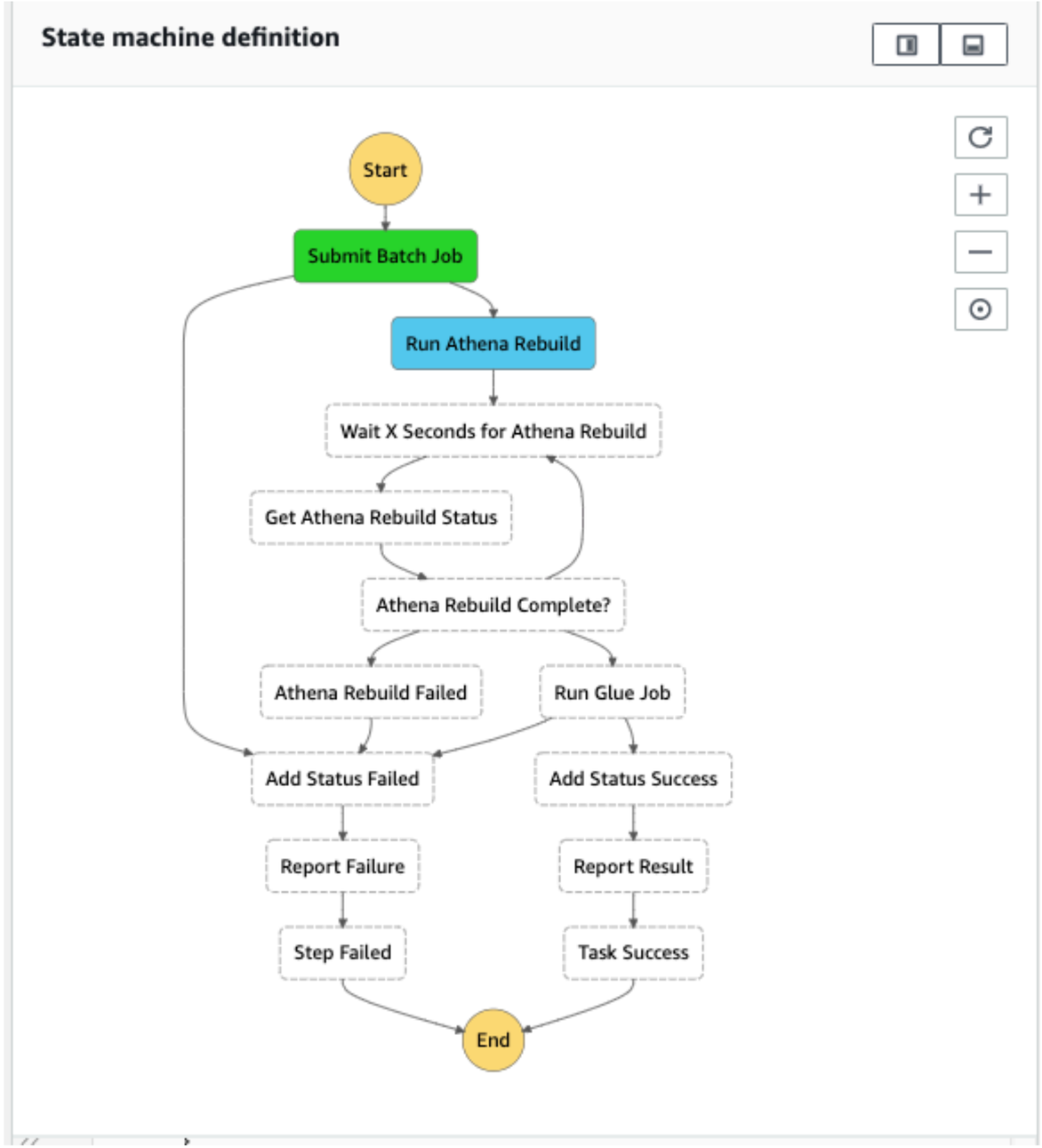
Report



Extract/Transform

Load

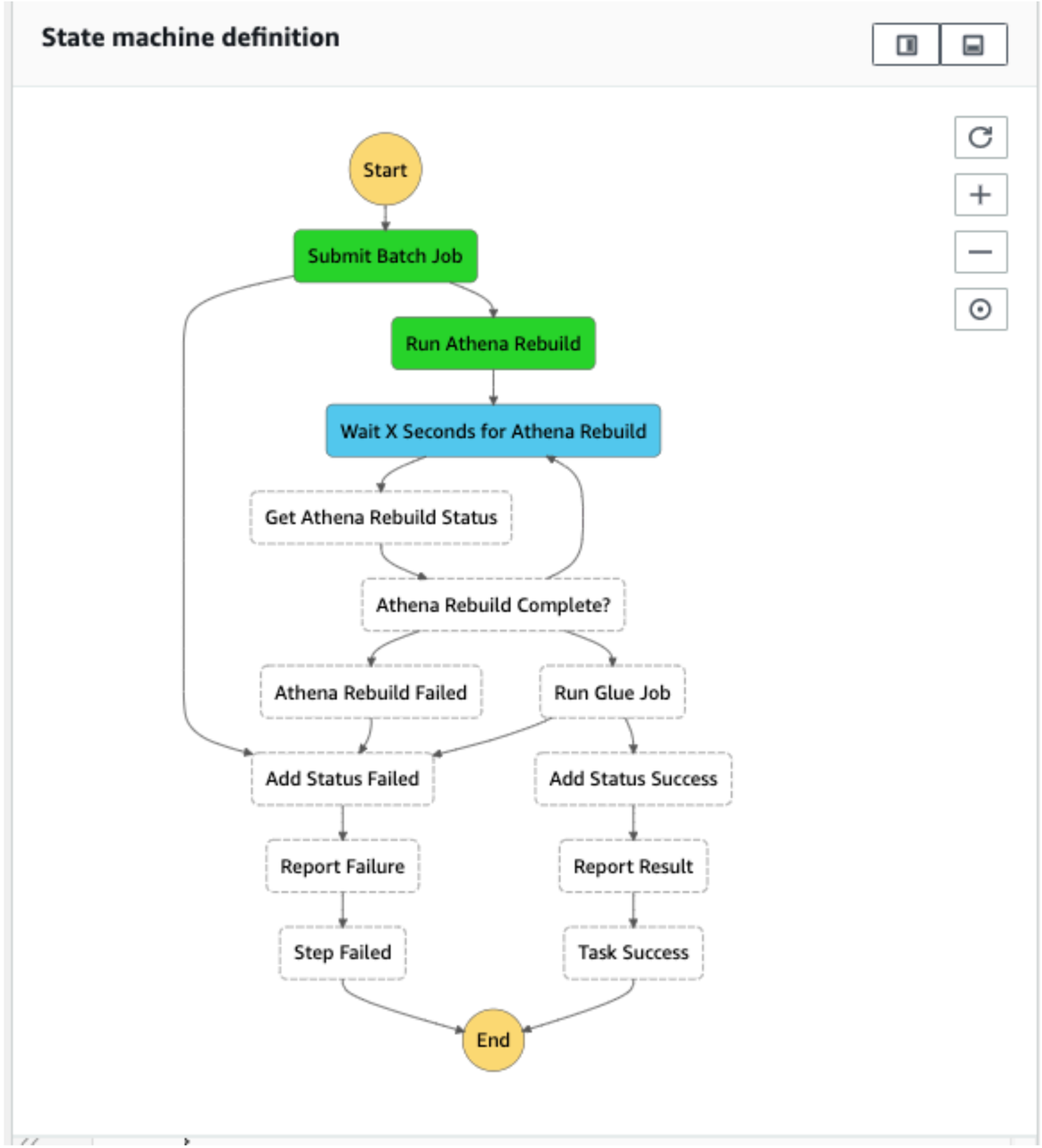
Report



Extract/Transform

Load

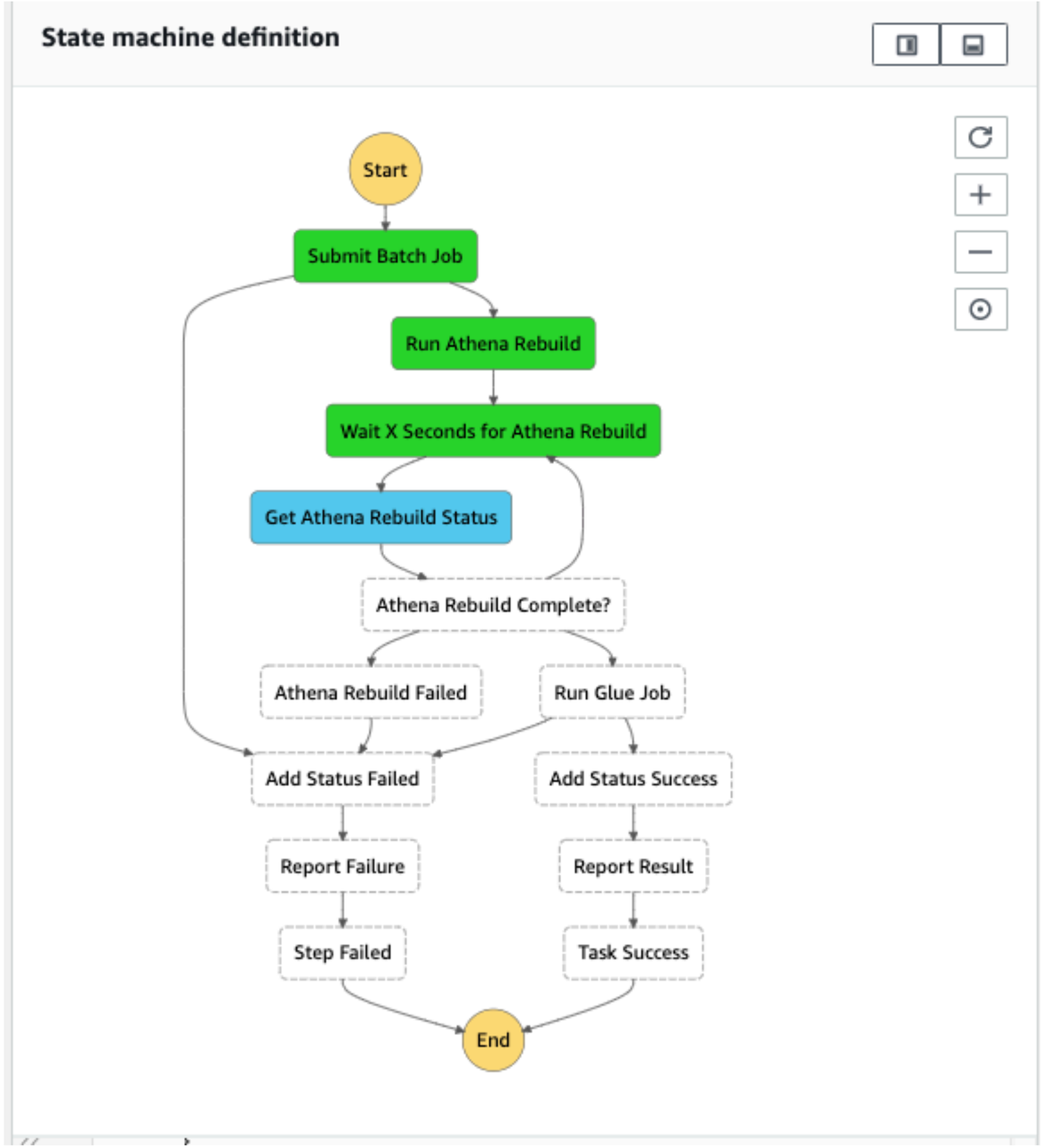
Report



Extract/Transform

Load

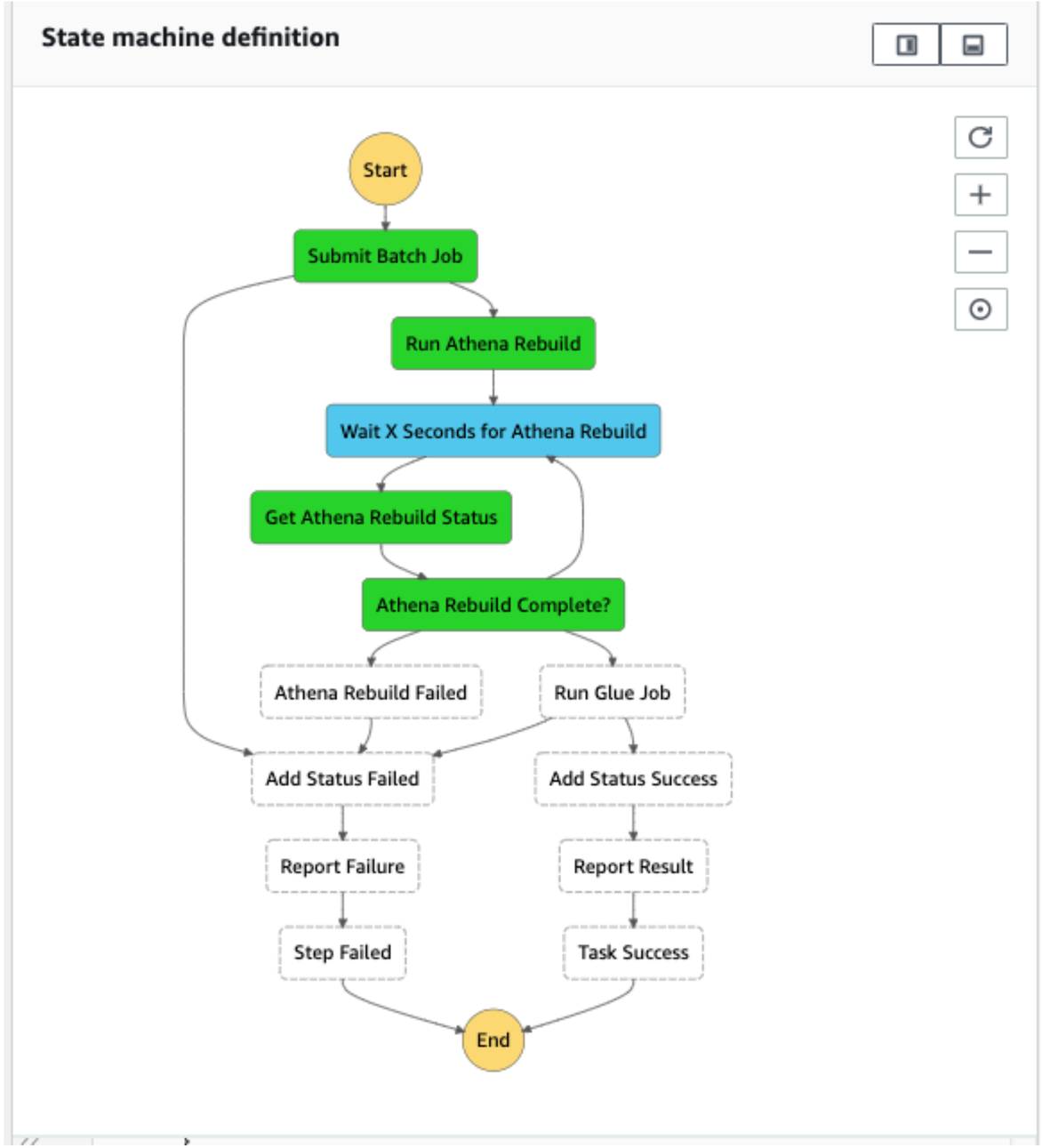
Report



Extract/Transform

Load

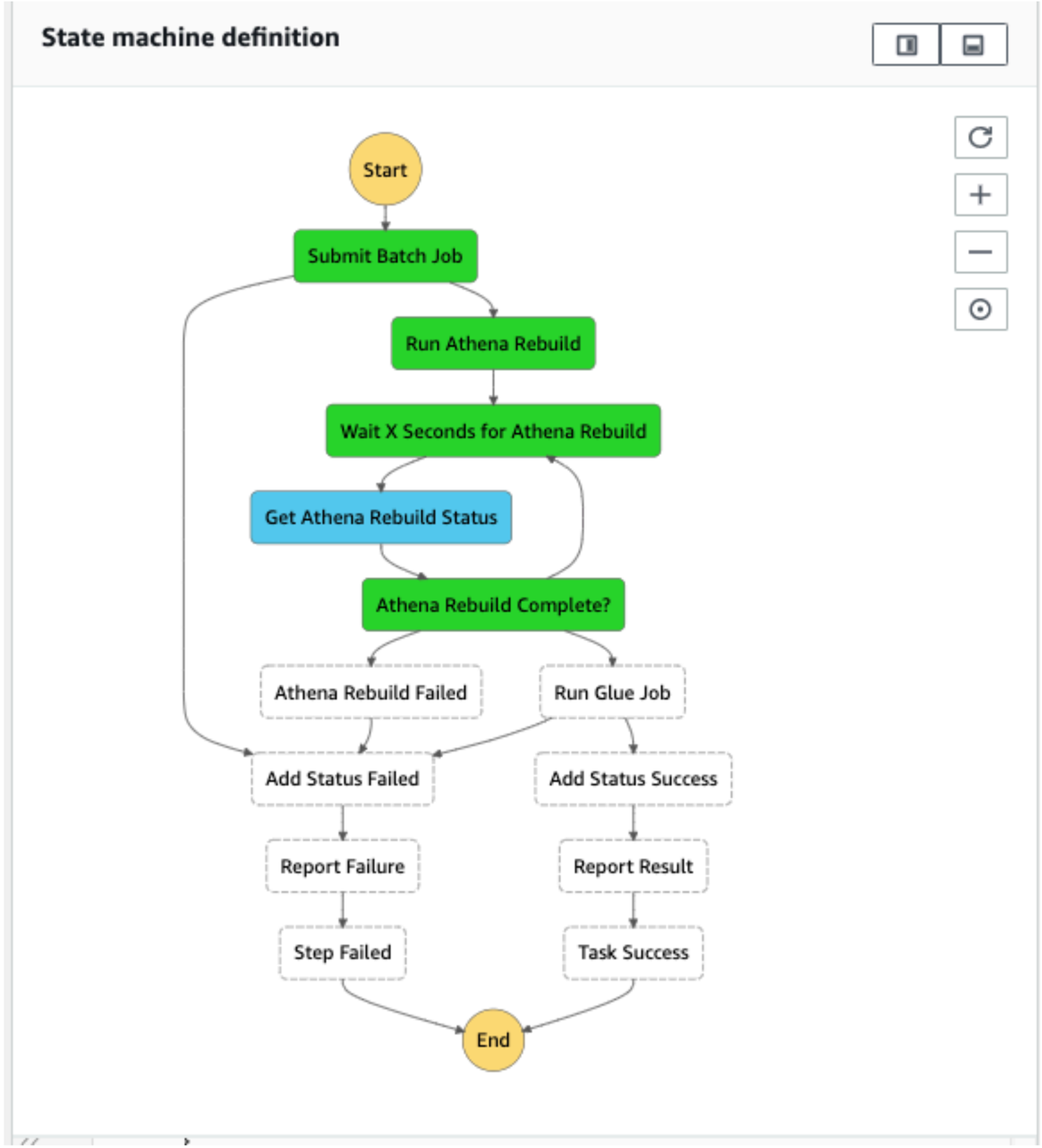
Report



Extract/Transform

Load

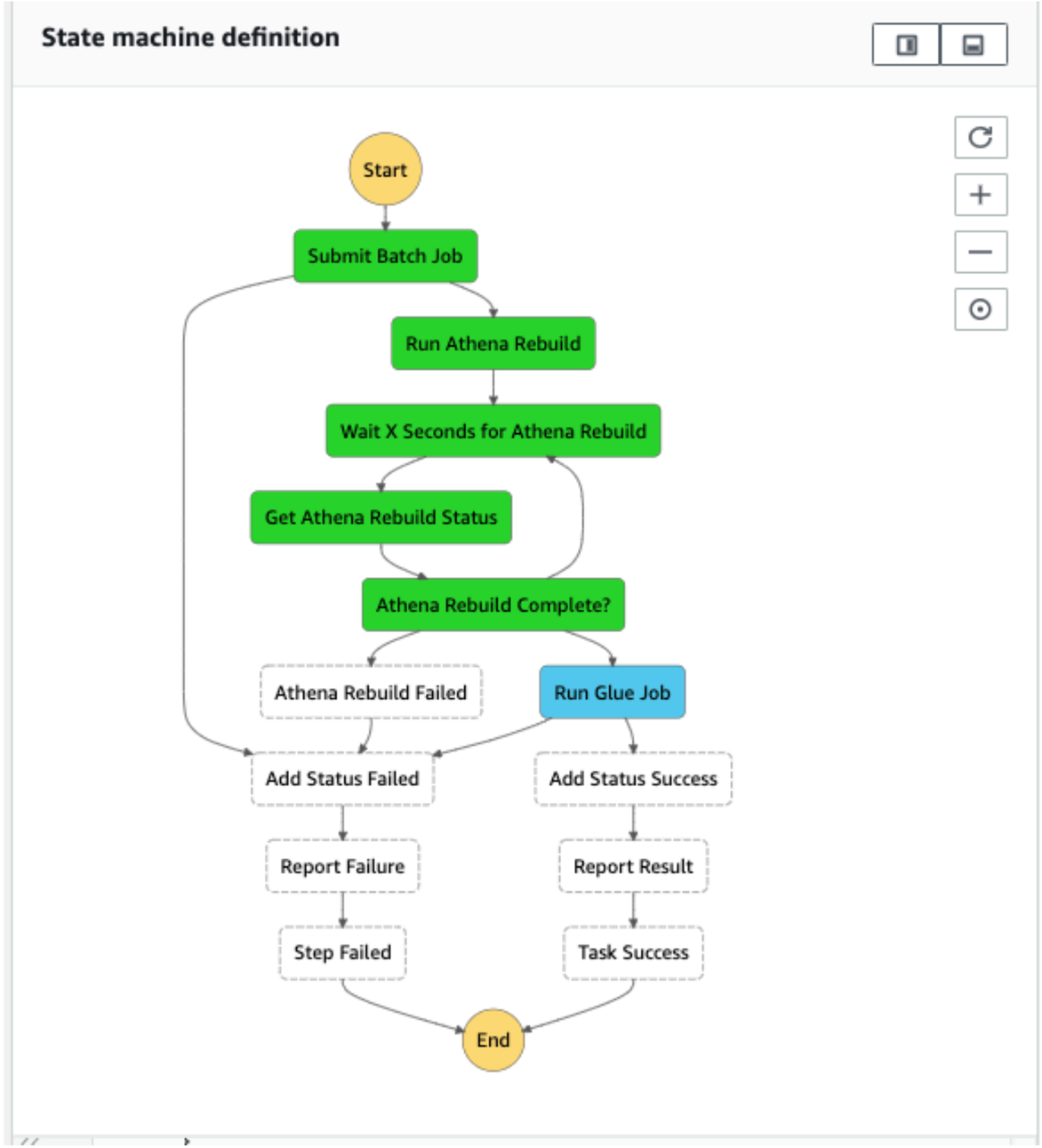
Report



Extract/Transform

Load

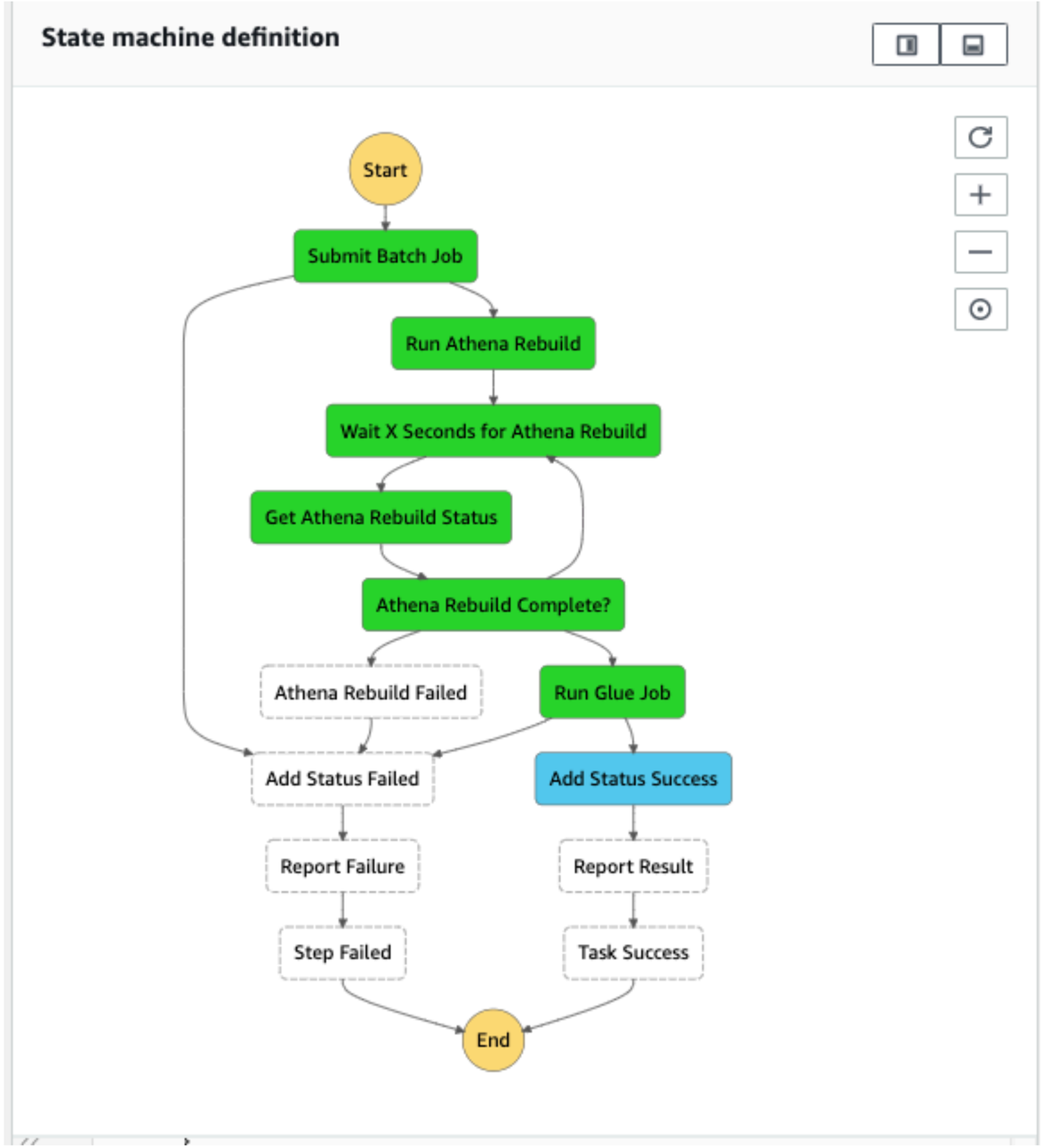
Report



Extract/Transform

Load

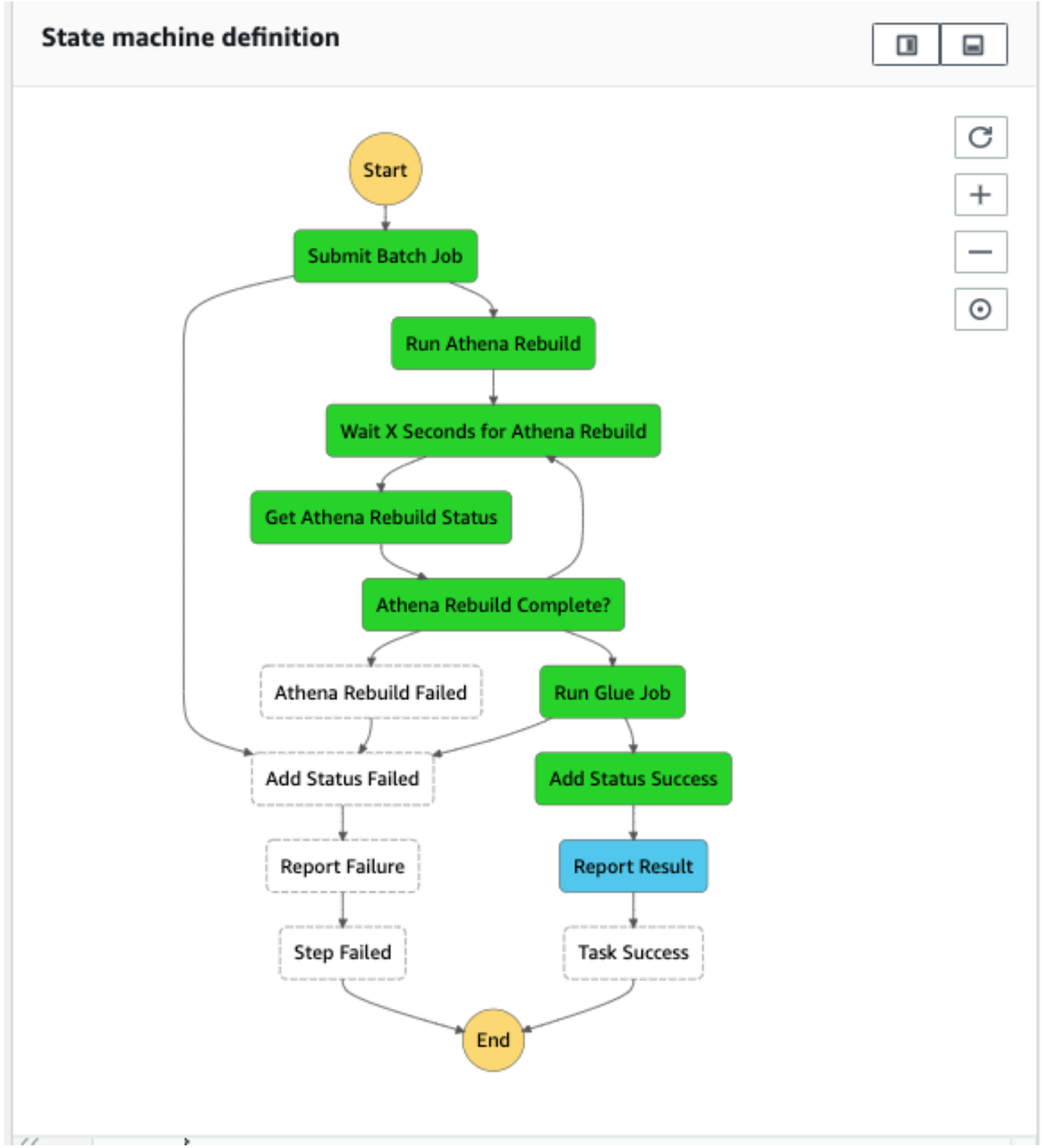
Report



Extract/Transform

Load

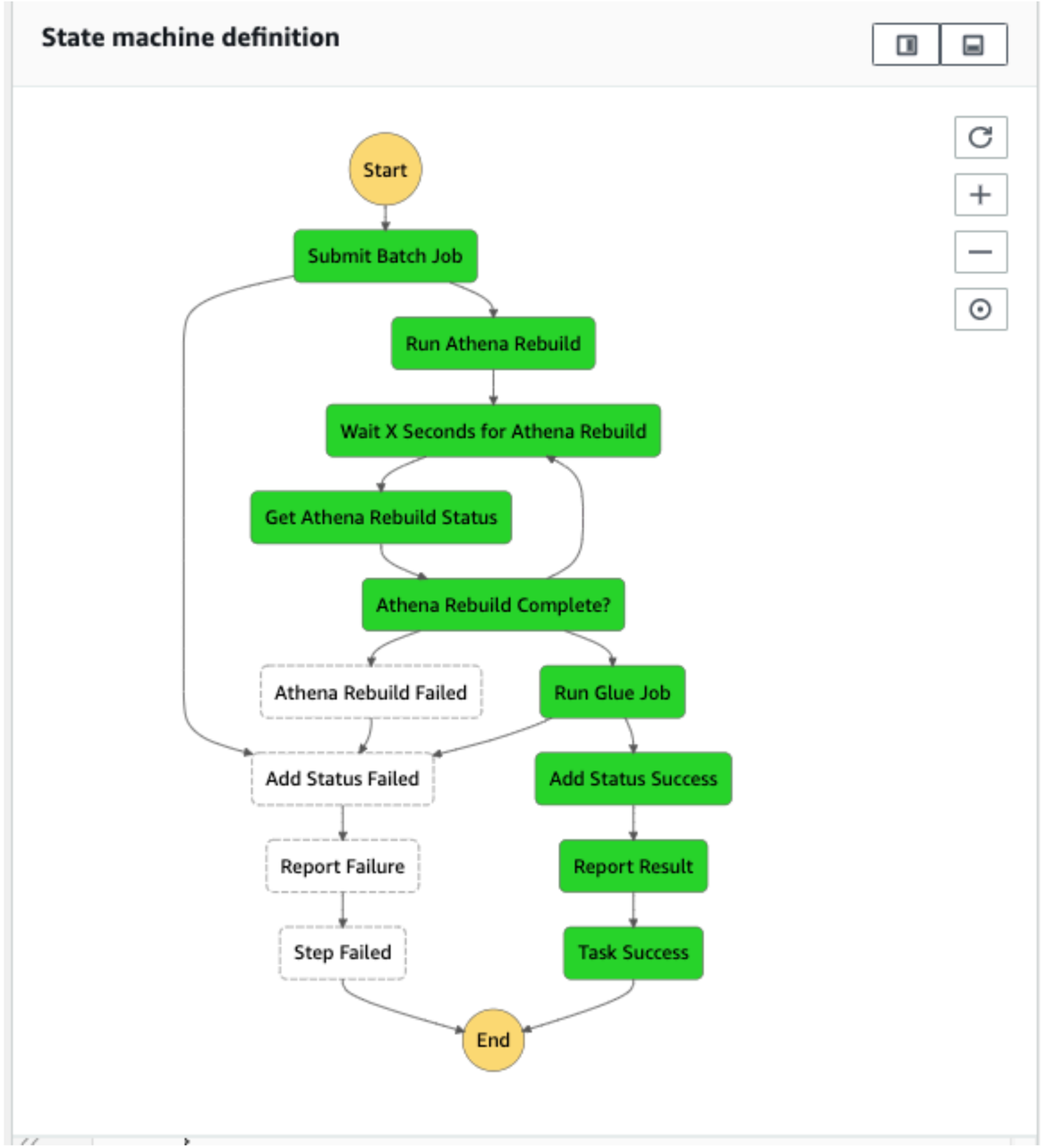
Report



Extract/Transform

Load

Report

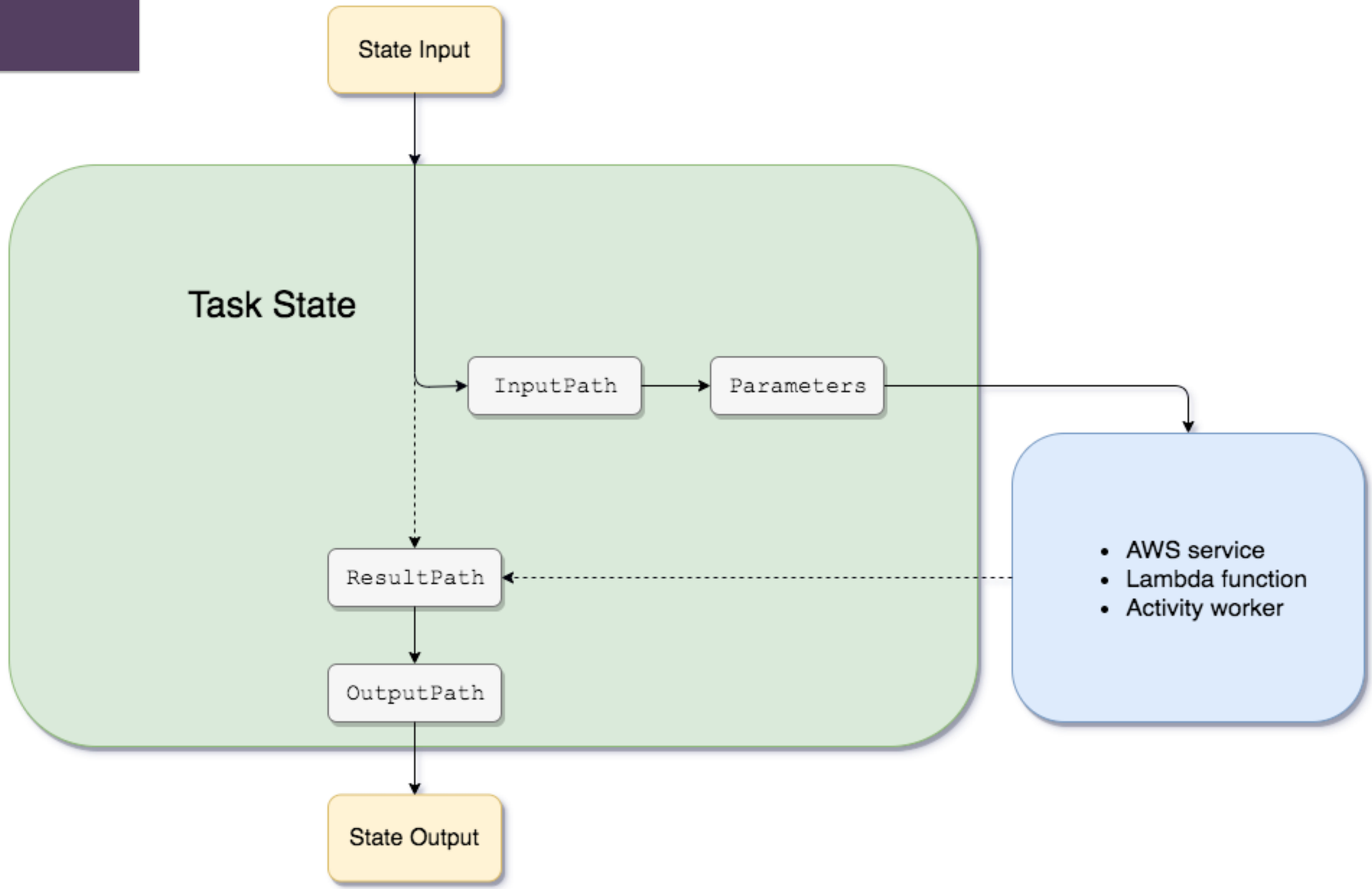


Extract/Transform

Load

Report

State Input


State Input
Task State
InputPath
Parameters
ResultPath
OutputPath
State Output
AWS service
Lambda function
Activity worker

Step Function Definition

```
"Submit Batch Job": {
  "Type": "Task",
  "Resource": "arn:aws:states:::batch:submitJob.sync",
  "Parameters": {
    "JobName.$": "$.jobName",
    "JobQueue.$": "$.batchJobQueue",
    "JobDefinition.$": "$.batchJobDef",
    "ContainerOverrides": {
      "Environment": [
        {
          "Name": "logging",
          "Value.$": "$.logging"
        }
      ],
      "Command.$": "$.batchJobCmd"

    }
  },
  "ResultPath": "$.result",
  "Next": "Run Athena Rebuild",
  "Catch": [
      {
        "ErrorEquals": [ "States.ALL" ],
        "ResultPath": "$.failstep",
        "Next": "Add Status Failed"
      }
  ]
},
```

Task Definition

```
{
  "athenaTable": "parcel_county_example",
  "athenaDatabase": "structured",
  "batchJobQueue": "arn:aws:batch:us-east-2:xxxxx:job-queue
    /ETLJobQueue-xxxxxxxx",
  "batchJobDef": "arn:aws:batch:us-east-2:xxxxx:job-definition
    /Parcel_CA_EXAMPLE_Job:21",
  "glueJobName": "ParcelCaExample_GlueJob",
  "metricTopic": "arn:aws:sns:us-east-2:xxxxx:DataDogMetrics",
  "wait_time": "30",
  "dataType": "parcel",
  "state": "ca",
  "county": "example",
  "jobName": "ParcelCaExample20190715234500",
  "logging": "{\"stateMachineLink\": \"https://us-east-2.console
    .aws.amazon.com/states/home?region=us-east-2#/executions
    /details/ParcelCaExample20190715234500\",
    \"stateMachineExecutionID\":
    \                          0190715234500\", \"stepfunctionName\":
    \                          ", \"countyName\": \"Example\",
    \"state\": \"ca\", \"dataType\": \"parcel\", \"start-time\":
    \"20190715234500\"}",
  "batchJobCmd": []
}
```

Job Input

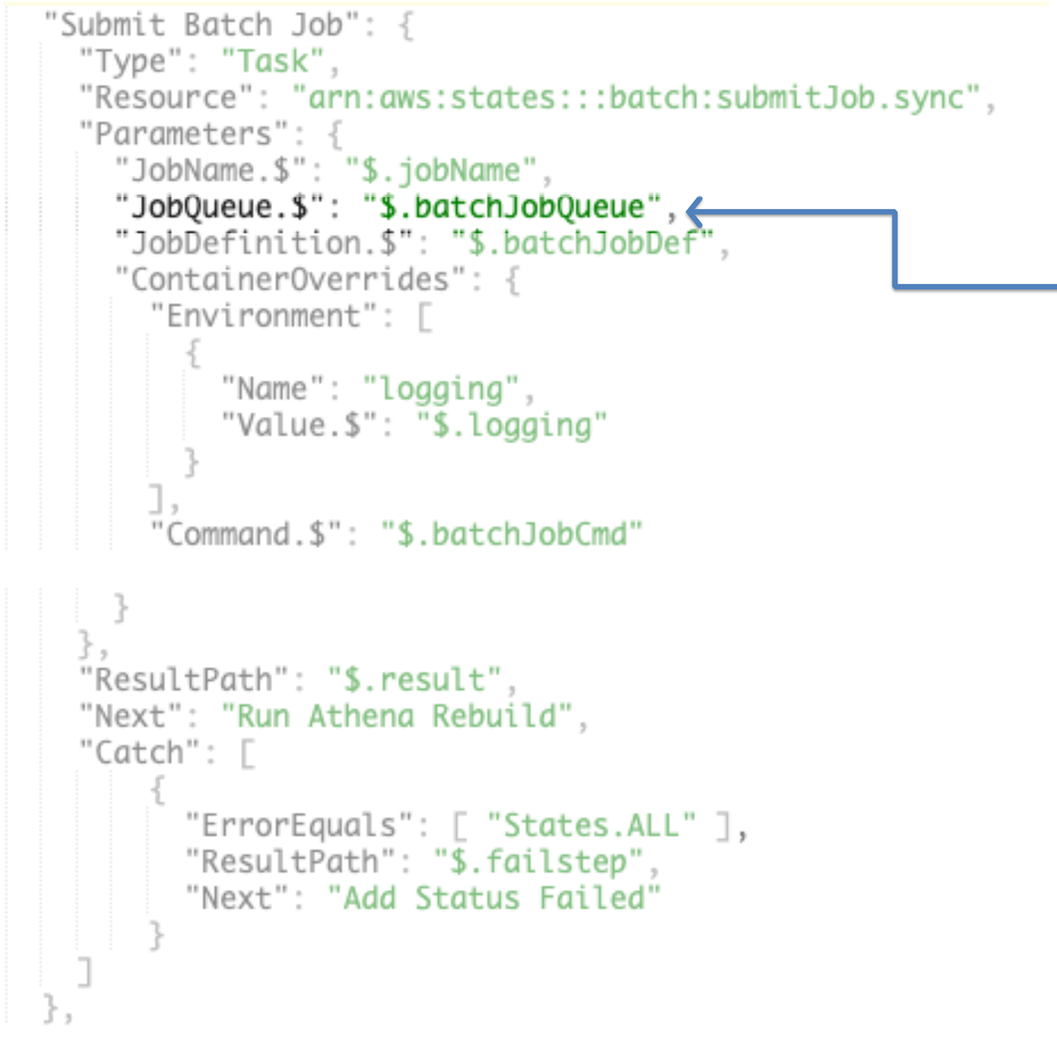
```
"Submit Batch Job": {
  "Type": "Task",
  "Resource": "arn:aws:states:::batch:submitJob.sync",
  "Parameters": {
    "JobName.$": "$.jobName",
    "JobQueue.$": "$.batchJobQueue",
    "JobDefinition.$": "$.batchJobDef",
    "ContainerOverrides": {
      "Environment": [
        {
          "Name": "logging",
          "Value.$": "$.logging"
        }
      ],
      "Command.$": "$.batchJobCmd"

    }
  },
  "ResultPath": "$.result",
  "Next": "Run Athena Rebuild",
  "Catch": [
      {
        "ErrorEquals": [ "States.ALL" ],
        "ResultPath": "$.failstep",
        "Next": "Add Status Failed"
      }
  ]
},
```

```
{
  "athenaTable": "parcel_county_example",
  "athenaDatabase": "structured",
  "batchJobQueue": "arn:aws:batch:us-east-2:xxxxx:job-queue
    /ETLJobQueue-xxxxxxxx",
  "batchJobDef": "arn:aws:batch:us-east-2:xxxxx:job-definition
    /Parcel_CA_EXAMPLE_Job:21",
  "glueJobName": "ParcelCaExample_GlueJob",
  "metricTopic": "arn:aws:sns:us-east-2:xxxxx:DataDogMetrics",
  "wait_time": "30",
  "dataType": "parcel",
  "state": "ca",
  "county": "example",
  "jobName": "ParcelCaExample20190715234500",
  "logging": "{\"stateMachineLink\": \"https://us-east-2.console
    .aws.amazon.com/states/home?region=us-east-2#/executions
    /details/ParcelCaExample20190715234500\",
    \"stateMachineExecutionID\":
    \                          0190715234500\", \"stepfunctionName\":
    \                          ", \"countyName\": \"Example\",
    \"state\": \"ca\", \"dataType\": \"parcel\", \"start-time\":
    \"20190715234500\"}",
  "batchJobCmd": []
}
```

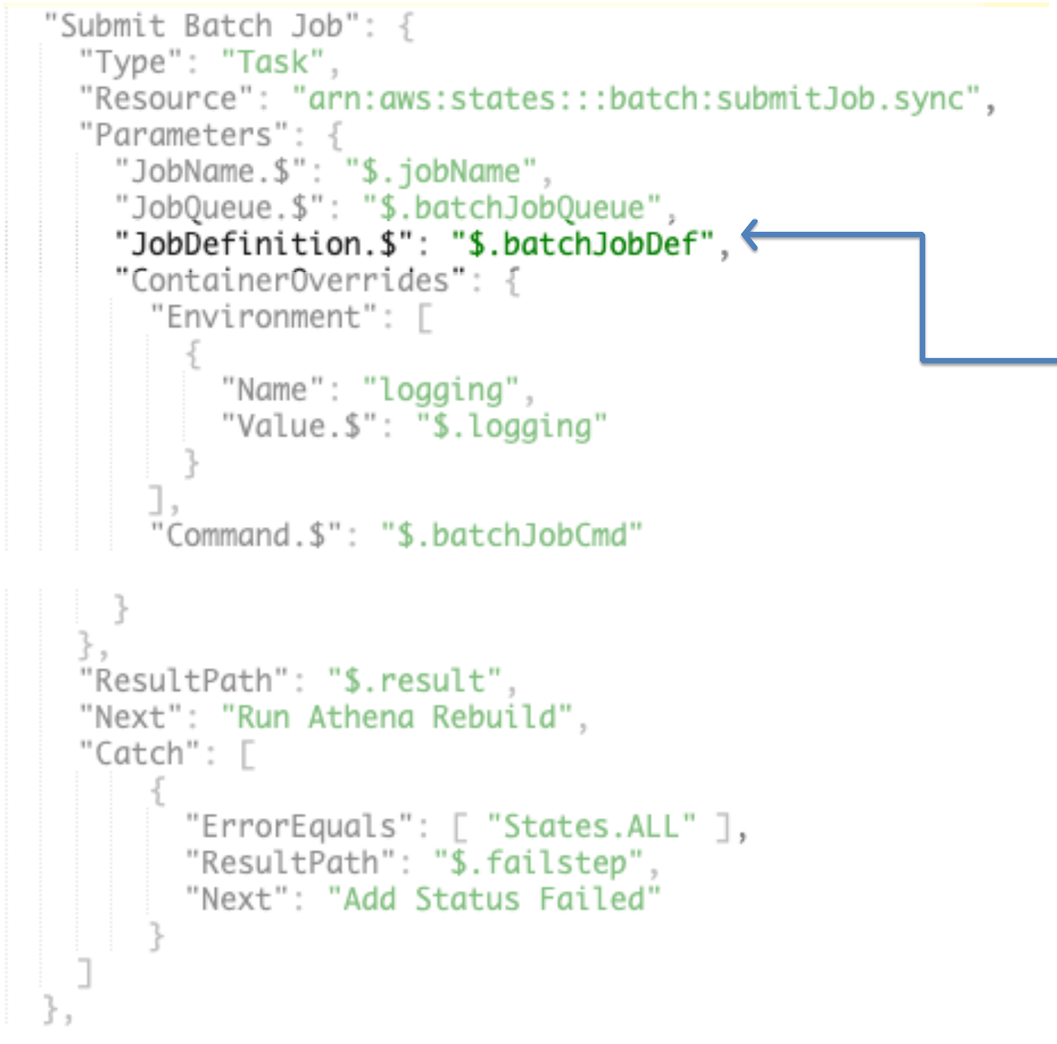
Task Definition

Job Input



Task Definition

Job Input



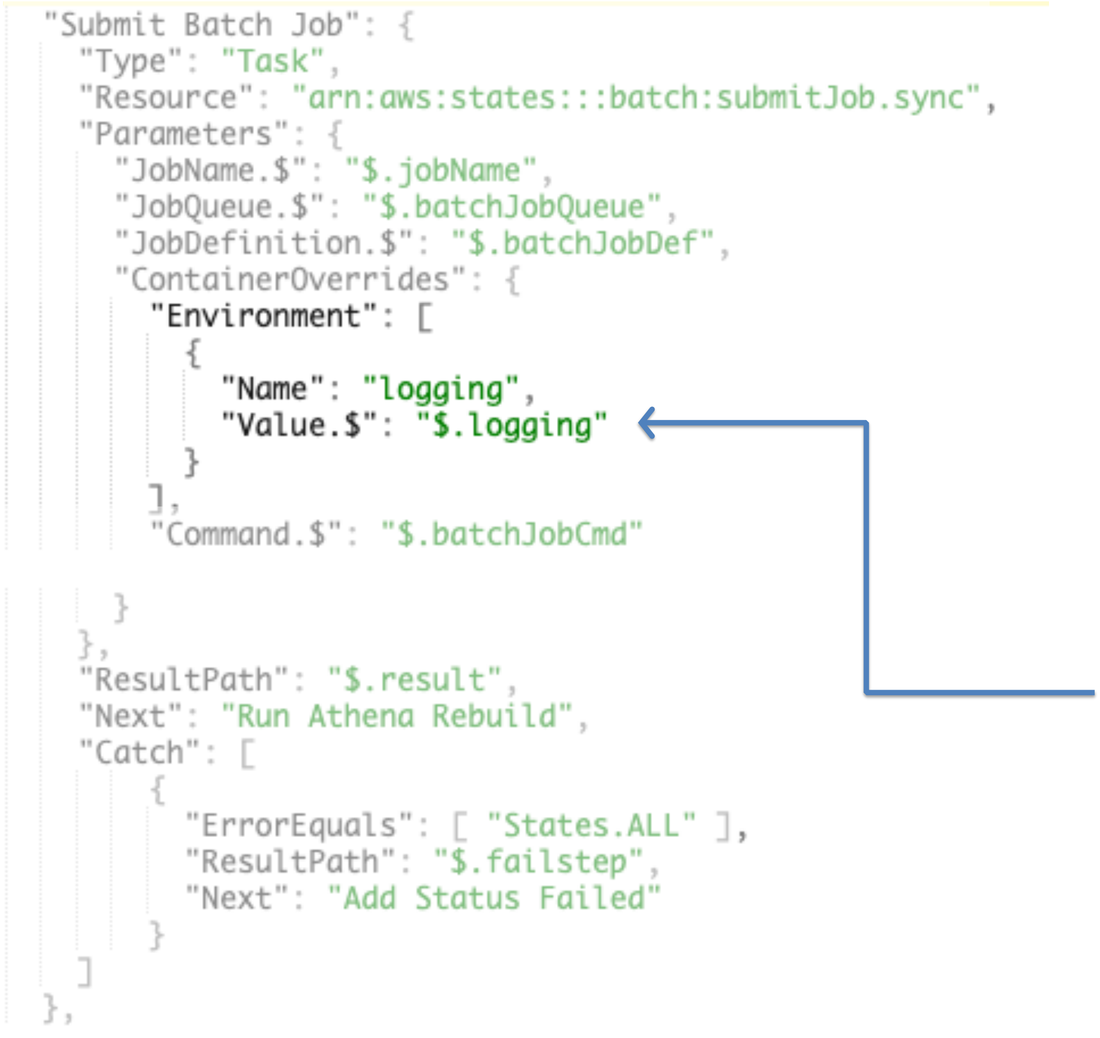
```
"Submit Batch Job": {
  "Type": "Task",
  "Resource": "arn:aws:states:::batch:submitJob.sync",
  "Parameters": {
    "JobName.$": "$.jobName",
    "JobQueue.$": "$.batchJobQueue",
    "JobDefinition.$": "$.batchJobDef",
    "ContainerOverrides": {
      "Environment": [
        {
          "Name": "logging",
          "Value.$": "$.logging"
        }
      ],
      "Command.$": "$.batchJobCmd"

    }
  },
  "ResultPath": "$.result",
  "Next": "Run Athena Rebuild",
  "Catch": [
      {
        "ErrorEquals": [ "States.ALL" ],
        "ResultPath": "$.failstep",
        "Next": "Add Status Failed"
      }
  ]
},
```

Task Definition

```
{
  "athenaTable": "parcel_county_example",
  "athenaDatabase": "structured",
  "batchJobQueue": "arn:aws:batch:us-east-2:xxxxx:job-queue
    /ETLJobQueue-xxxxxxxx",
  "batchJobDef": "arn:aws:batch:us-east-2:xxxxx:job-definition
    /Parcel_CA_EXAMPLE_Job:21",
  "glueJobName": "ParcelCaExample_GlueJob",
  "metricTopic": "arn:aws:sns:us-east-2:xxxxx:DataDogMetrics",
  "wait_time": "30",
  "dataType": "parcel",
  "state": "ca",
  "county": "example",
  "jobName": "ParcelCaExample20190715234500",
  "logging": "{\"stateMachineLink\": \"https://us-east-2.console
    .aws.amazon.com/states/home?region=us-east-2#/executions
    /details/ParcelCaExample20190715234500\",
    \"stateMachineExecutionID\":
    \                          0190715234500\", \"stepfunctionName\":
    \                          ", \"countyName\": \"Example\",
    \"state\": \"ca\", \"dataType\": \"parcel\", \"start-time\":
    \"20190715234500\"}",
  "batchJobCmd": []
}
```

Job Input



```
"Submit Batch Job": {
  "Type": "Task",
  "Resource": "arn:aws:states:::batch:submitJob.sync",
  "Parameters": {
    "JobName.$": "$.jobName",
    "JobQueue.$": "$.batchJobQueue",
    "JobDefinition.$": "$.batchJobDef",
    "ContainerOverrides": {
      "Environment": [
        {
          "Name": "logging",
          "Value.$": "$.logging"
        }
      ],
      "Command.$": "$.batchJobCmd"

    }
  },
  "ResultPath": "$.result",
  "Next": "Run Athena Rebuild",
  "Catch": [
      {
        "ErrorEquals": [ "States.ALL" ],
        "ResultPath": "$.failstep",
        "Next": "Add Status Failed"
      }
  ]
},
```

```
{
  "athenaTable": "parcel_county_example",
  "athenaDatabase": "structured",
  "batchJobQueue": "arn:aws:batch:us-east-2:xxxxx:job-queue
    /ETLJobQueue-xxxxxxxx",
  "batchJobDef": "arn:aws:batch:us-east-2:xxxxx:job-definition
    /Parcel_CA_EXAMPLE_Job:21",
  "glueJobName": "ParcelCaExample_GlueJob",
  "metricTopic": "arn:aws:sns:us-east-2:xxxxx:DataDogMetrics",
  "wait_time": "30",
  "dataType": "parcel",
  "state": "ca",
  "county": "example",
  "jobName": "ParcelCaExample20190715234500",
  "logging": "{\"stateMachineLink\": \"https://us-east-2.console
    .aws.amazon.com/states/home?region=us-east-2#/executions
    /details/ParcelCaExample20190715234500\",
    \"stateMachineExecutionID\":
    \                          0190715234500\", \"stepfunctionName\":
    \                          ", \"countyName\": \"Example\",
    \"state\": \"ca\", \"dataType\": \"parcel\", \"start-time\":
    \"20190715234500\"}",
  "batchJobCmd": []
}
```

Task Definition

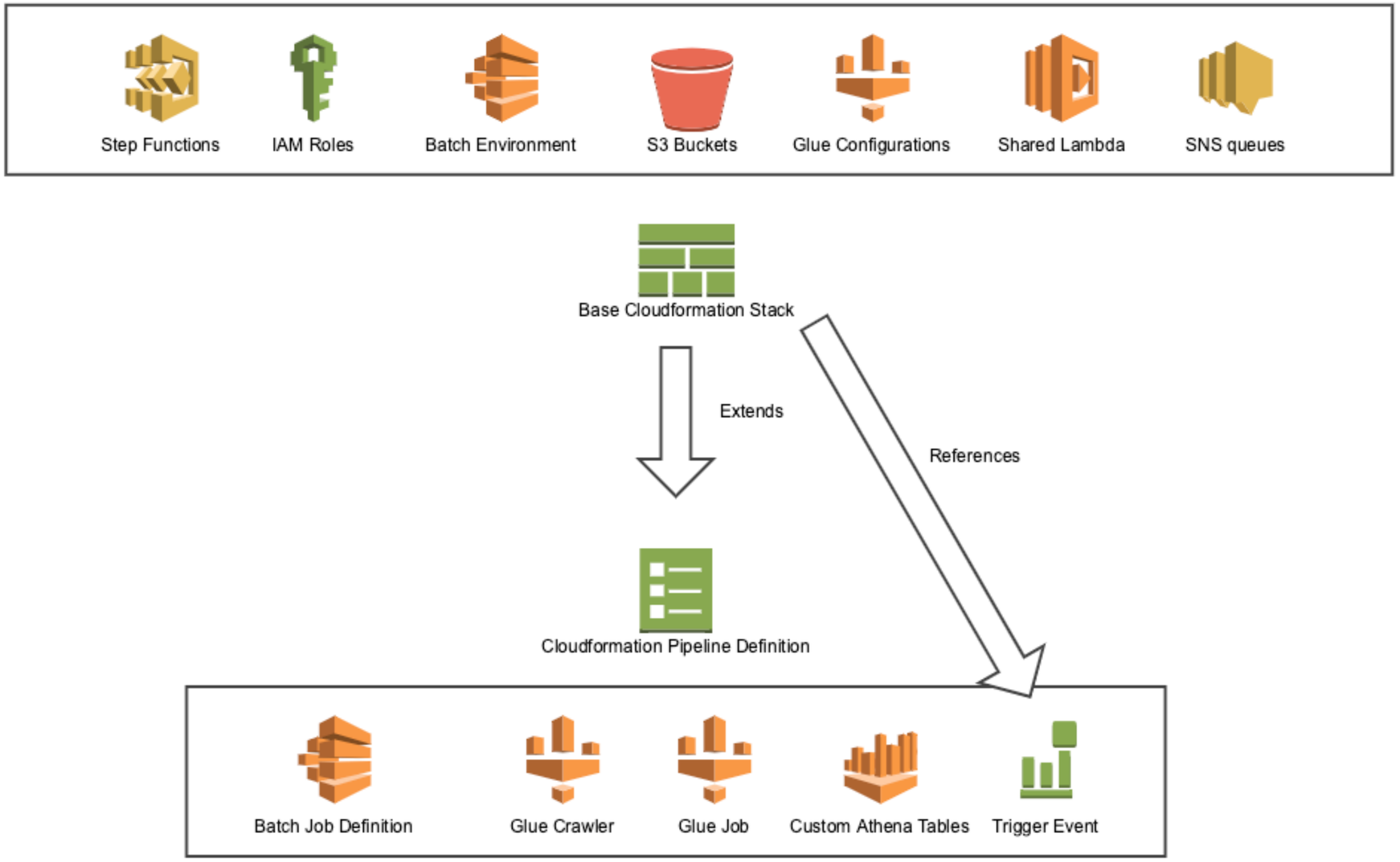
Job Input

Task Definition

Job Input

Creating a custom step function for each pipeline, ended up with a lot of duplicated code and as we added more validation we had to go and update it everywhere.

Basic Pipeline Definition



Triggering a Data Pipeline

Where are the Logs?

- Execution event history summary in the Step Function console
- CloudWatch Logs - **LogGroup**/LogStream
  - Lambda
    - **/aws/lambda/functionName1/**LambdaExecutionID1
    - **/aws/lambda/functionName2/**LambdaExecutionID1
  - Batch
    - **/aws/batch/job/JobName_Batch_Job**/BatchExecutionID
  - Glue
    - **/aws-glue/jobs/output/**GlueExecutionID
    - **/aws-glue/jobs/error**/GlueExecutionID
    - **/aws-glue/crawlers/**JobName_Crawler

Tried redirecting all the logs for each pipeline to its own cloud watch stream.

This is possible but made a mess, all the shortcuts AWS gives you with errors to look at logs no longer worked.

JSON Logging

Everything is logged as JSON. python, pyspark, even bash.

For python we use JSON logging with: python-json-logger

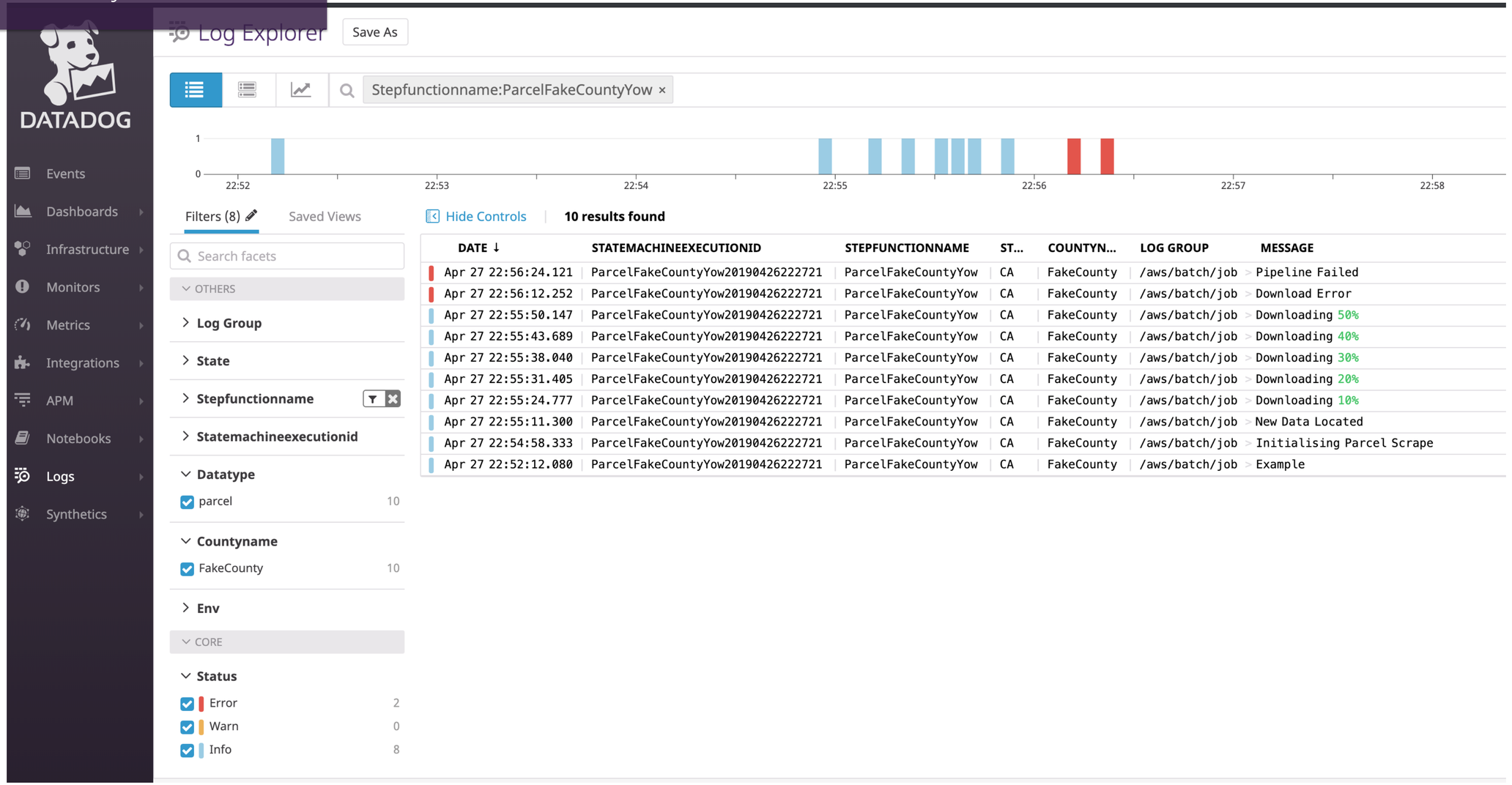
For shell scripts eg entrypoint.sh we use a forked version of bash-banyan that lets us set additional fields: https://github.com/JessFlan/bash-bunyan

Additional Logging Fields

- stateMachineLink
- stateMachineExecutionID
- stepfunctionName
- countyName
- city
- state
- dataType
- start-time

Data Dog

CloudWatch makes it hard to read JSON logs without expanding every line.



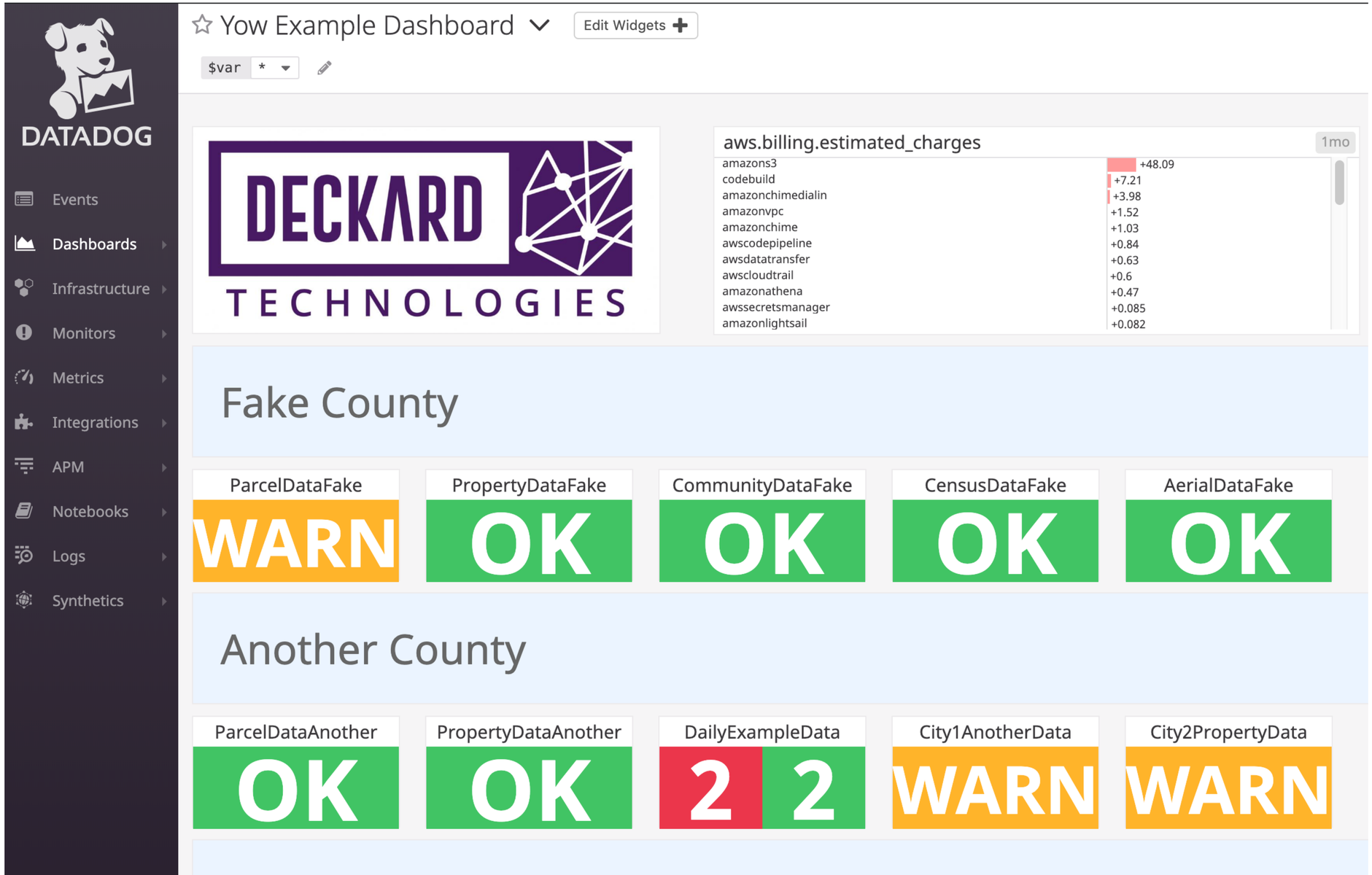
Data Dog Checks

Custom checks are an easy way to report the status of a pipeline or service.

At the end of a job run our custom lambda submits a check with a status of:

- Ok (Job ran produced data)
- Warning (Job ran no new data)
- Critical (Job Failed)

Dashboard Example



Checks are considered ‘Inactive’ if Data Dog doesn’t receive an update for 24 hours

Datadog Events

Events allow us to track pipeline results for months rather than hours.

Datadog Events

At the end of a job run our custom lambda uses the Datadog events API to:

- Submit an event of type error (Job Failed)
  - **tagged with “unresolved”**

Datadog Events

At the end of a job run our custom lambda uses the Datadog events API to:

- Submit an event of type error (Job Failed)
  - **tagged with “unresolved”**
- Submit an event of type success if the job ran produced data
  - **Tag all previous error events as resolved**

Datadog Events

At the end of a job run our custom lambda uses the Datadog events API to:

- Submit an event of type error (Job Failed)
  - **tagged with “unresolved”**
- Submit an event of type success if the job ran produced data
  - **Tag all previous error events as resolved**
- Submit an event of type warning if the job ran no new data

Datadog Events

At the end of a job run our custom lambda uses the Datadog events API to:

- Submit an event of type error (Job Failed)
  - **tagged with “unresolved”**
- Submit an event of type success if the job ran produced data
  - **Tag all previous error events as resolved**
- Submit an event of type warning if the job ran no new data

These are all tagged with the same fields as our JSON logs.

Dashboard Example

☆ Presentation

Add Template Variables

County 1

County 1

OK
8

STATUS	NAME
OK	[redacted]parcel has new data within 32 days
OK	[redacted]permit(permitlist): Run within 8 days
OK	[redacted](permitlist): New data within 16 days
OK	[redacted] permit(permit): Job succeeded

County 2

County 2

Alert
3

OK
8

STATUS	NAME
ALERT	[redacted] permit: New data within 16 days
ALERT	[redacted] permit: Run within 8 days
ALERT	[redacted]permit(permit): New data within 16 days
OK	[redacted]permit(permit): Job succeeded

County 3

County 3

Alert
3

OK
5

STATUS	NAME
ALERT	[redacted]ermit(permitlist): Run within 8 days
ALERT	[redacted]ermit(permitlist): New data within 16 days
ALERT	[redacted]ermit(permit): New data within 16 days

Assessor Website

Web Replicant

OK
2

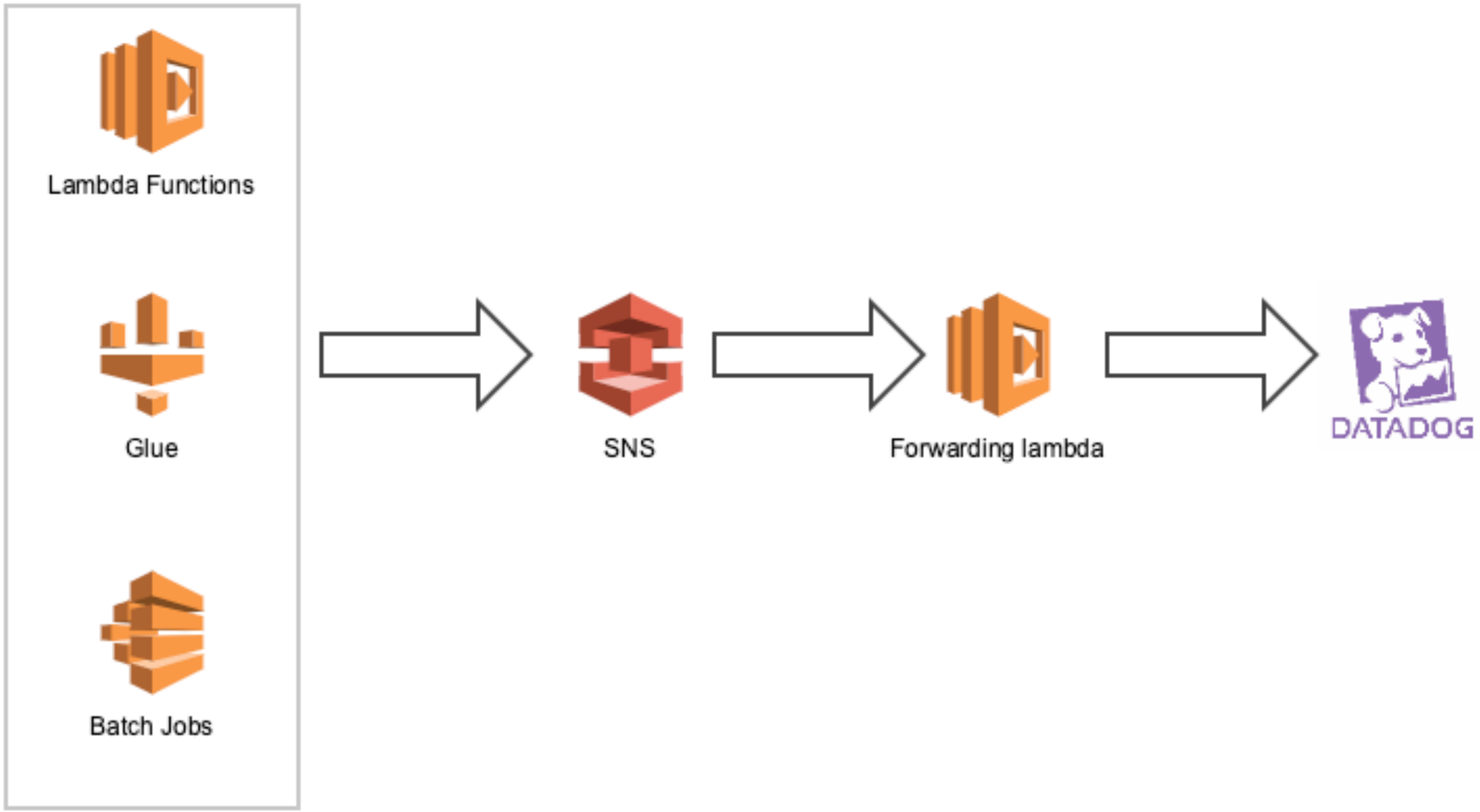
STATUS	NAME
OK	[Synthetics] Test or[redacted]
OK	[Synthetics] Test or[redacted]

aws.billing.estimated_charges

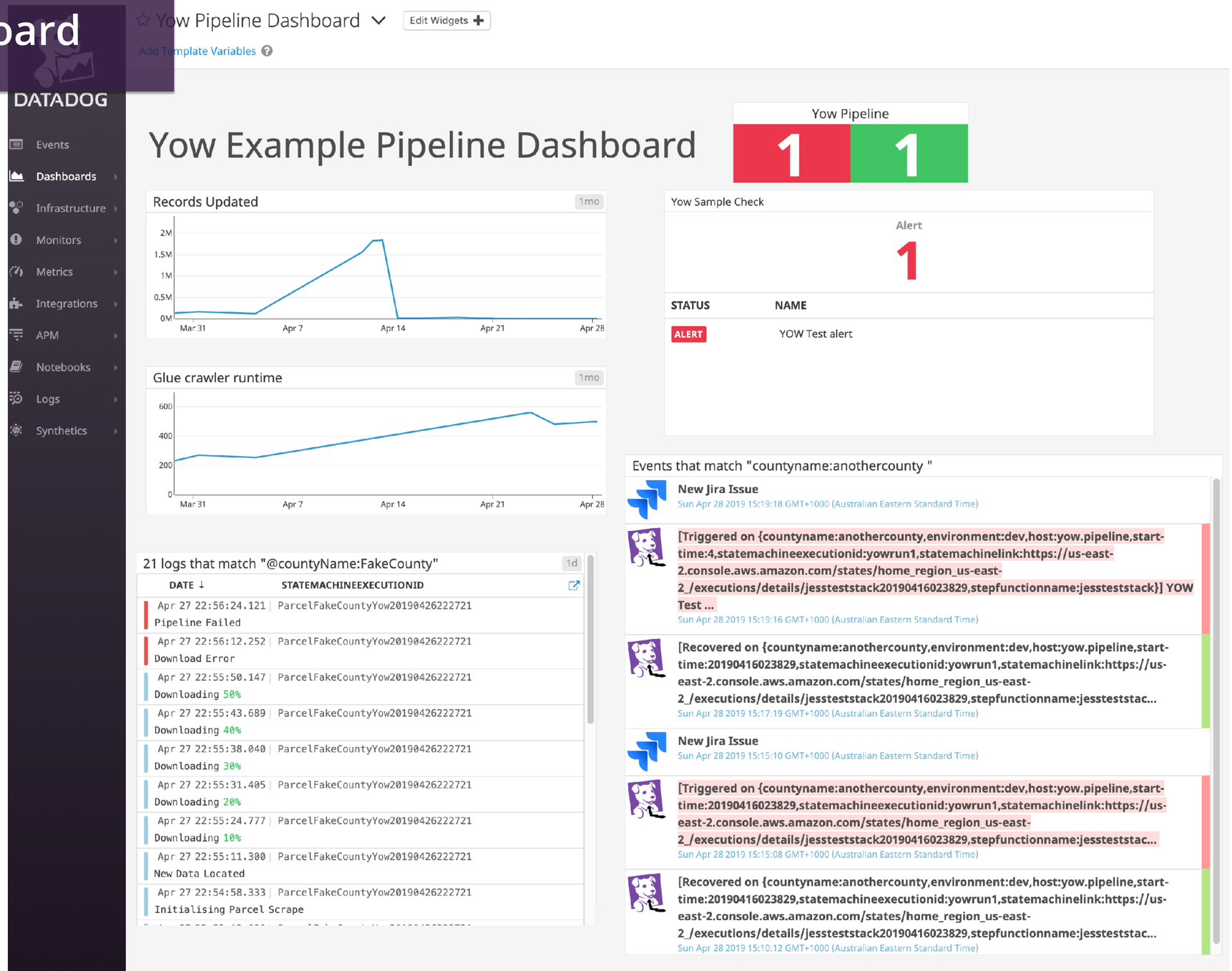
1mo

awsglue	+125
amazons3	+42.7
amazonec2	+32.81
amazonathena	+19
awssupportbusiness	+16.34
awscodepipeline	+9.02
amazonvpc	+6.24
amazonecr	+2.82
codebuild	+1.64
amazonroute53	+0.86
amazonrds	+0.68
amazonchimedialin	+0.47
amazonsagemaker	+0.43
amazonlightsail	+0.27
awslambda	+0.12

Monitoring Pipeline Metrics



Pipeline Dashboard



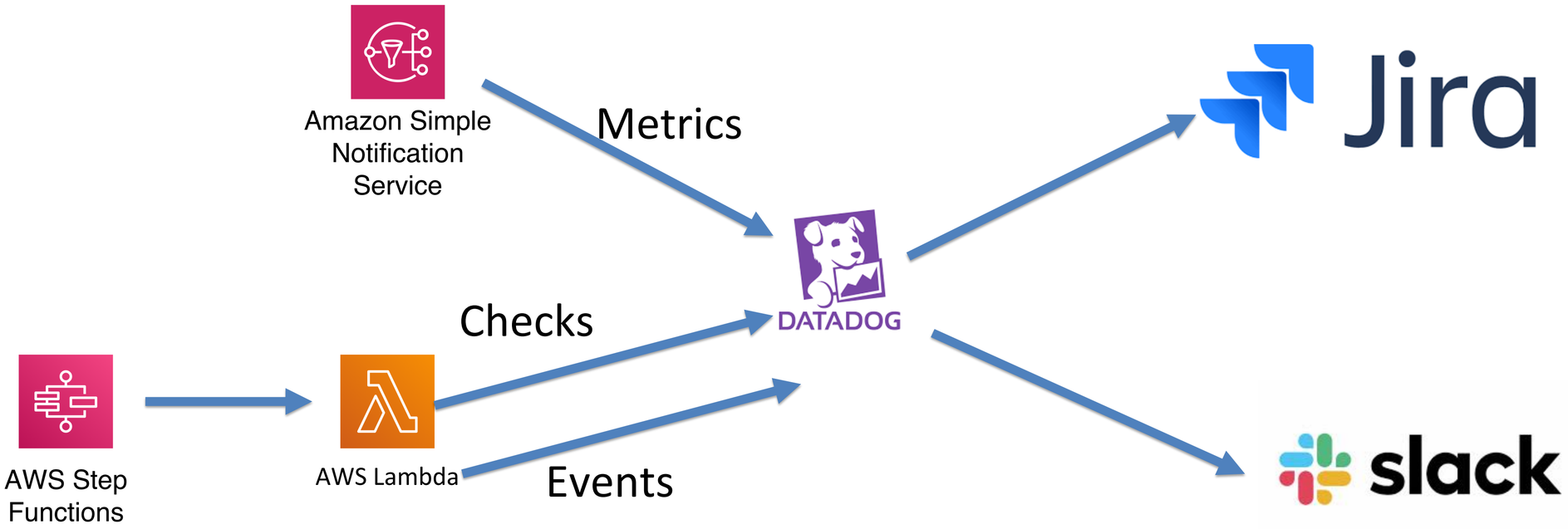
Alerting

We then tried using SNS task within AWS step functions to generate alerts and send metrics. However, you were only able to provide a JSON path we needed to construct a string and format the output.

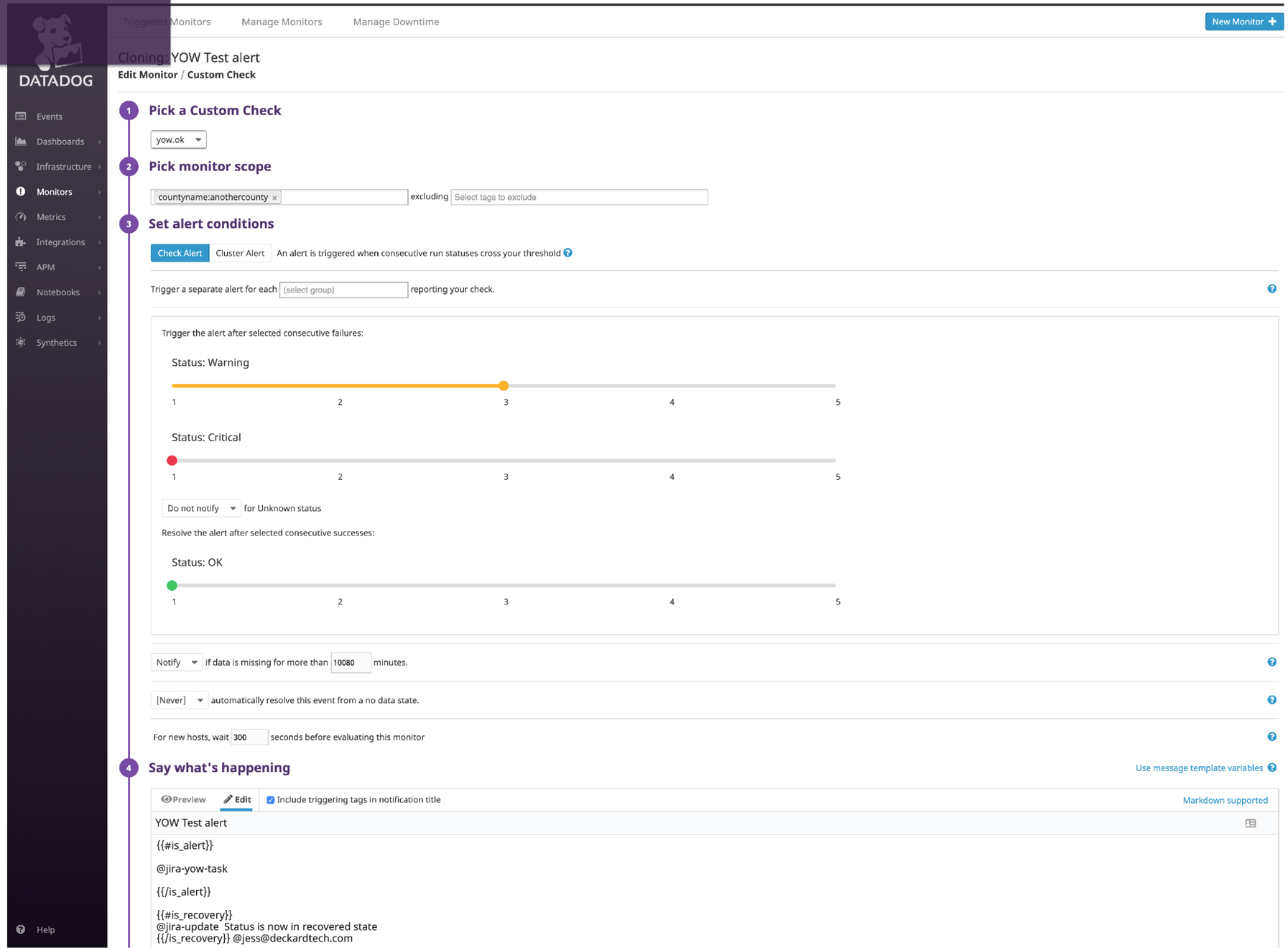
Alerting

Then we wrote a lambda to format notifications and pushed them to SNS which went to Slack. This was much too noisy; some pipelines run frequently and kept failing. Other failures got lost in the noise everyone thought someone else was dealing with it.

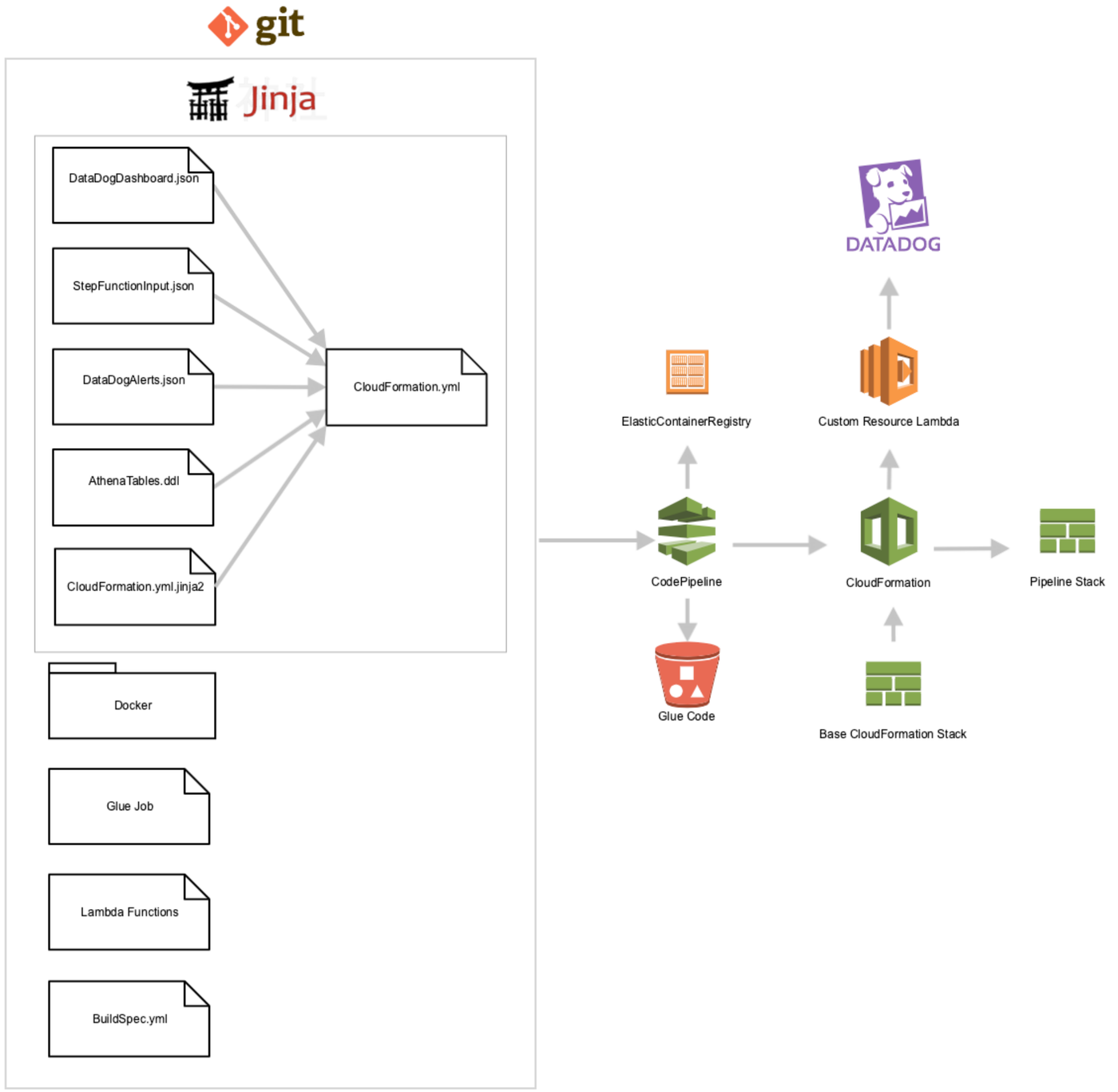
Alerting Solution



Example Alert



Deployment


git
Jinja
DataDogDashboard.json
StepFunctionInput.json
DataDogAlerts.json
AthenaTables.ddl
CloudFormation.yml.jinja2
CloudFormation.yml
Docker
Glue Job
Lambda Functions
BuildSpec.yml
DATADOG
ElasticContainerRegistry
Custom Resource Lambda
CodePipeline
CloudFormation
Pipeline Stack
Glue Code
Base CloudFormation Stack

Architecture Goals

Create Pipelines that are easy to debug. When they fail we need to see where the failure occurred an any related logging.

Monitoring so we can see if our pipelines are degrading or producing less results over time.

We need a whole of system view so we can can see that our pipelines are running and which succeed or fail.

A way of making sure that if our pipelines fail that someone is tasked with fixing the problem.

Architecture Goals

Create Pipelines that are easy to debug. When they fail we need to see where the failure occurred an any related logging.
Step Function + JSON logging to DataDog

Monitoring so we can see if our pipelines are degrading or producing less results over time.

We need a whole of system view so we can can see that our pipelines are running and which succeed or fail.

A way of making sure that if our pipelines fail that someone is tasked with fixing the problem.

Architecture Goals

Create Pipelines that are easy to debug. When they fail we need to see where the failure occurred an any related logging.

Step Function + JSON logging to DataDog

Monitoring so we can see if our pipelines are degrading or producing less results over time.

We need a whole of system view so we can can see that our pipelines are running and which succeed or fail.

DataDog Dashboards

A way of making sure that if our pipelines fail that someone is tasked with fixing the problem.

Architecture Goals

Create Pipelines that are easy to debug. When they fail we need to see where the failure occurred an any related logging.
Step Function + JSON logging to DataDog

Monitoring so we can see if our pipelines are degrading or producing less results over time.
DataDog Metrics

We need a whole of system view so we can can see that our pipelines are running and which succeed or fail.
DataDog Dashboards

A way of making sure that if our pipelines fail that someone is tasked with fixing the problem.

Architecture Goals

Create Pipelines that are easy to debug. When they fail we need to see where the failure occurred an any related logging.
Step Function + JSON logging to DataDog

Monitoring so we can see if our pipelines are degrading or producing less results over time.
DataDog Metrics

We need a whole of system view so we can can see that our pipelines are running and which succeed or fail.
DataDog Dashboards

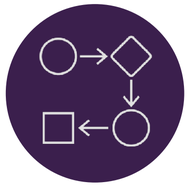
A way of making sure that if our pipelines fail that someone is tasked with fixing the problem.
DataDog Alerts + Jira

Recap

Log everything in JSON - Bash banyan for logging entry points etc.

Recap

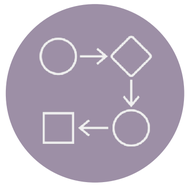
Log everything in JSON - Bash banyan
for logging entry points etc.



Where possible share step functions.

Recap

Log everything in JSON - Bash banyan
for logging entry points etc.



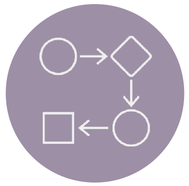
Where possible share step functions.

Use the same tags for checks, events, alerts and logging.

Recap

Log everything in JSON - Bash banyan for logging entry points etc.

Use custom resources in cloud formation so that tear down is sorted and everything is revision controlled.



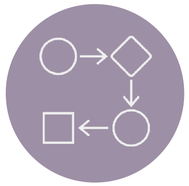
Where possible share step functions.

Use the same tags for checks, events, alerts and logging.

Recap

Log everything in JSON - Bash banyan for logging entry points etc.

Use custom resources in cloud formation so that tear down is sorted and everything is revision controlled.



Where possible share step functions.

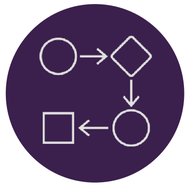
Don’t move where logging goes just push a copy.

Use the same tags for checks, events, alerts and logging.

Recap

Log everything in JSON - Bash banyan for logging entry points etc.

Use custom resources in cloud formation so that tear down is sorted and everything is revision controlled.



Where possible share step functions.

Don’t move where logging goes just push a copy.

Use the same tags for checks, events, alerts and logging.

Questions?

Jessica Flanagan
Chief of Data Architecture
Deckard Technologies