Building Serverlesspresso: Creating event-driven architectures

Bianca Mota (she/her)

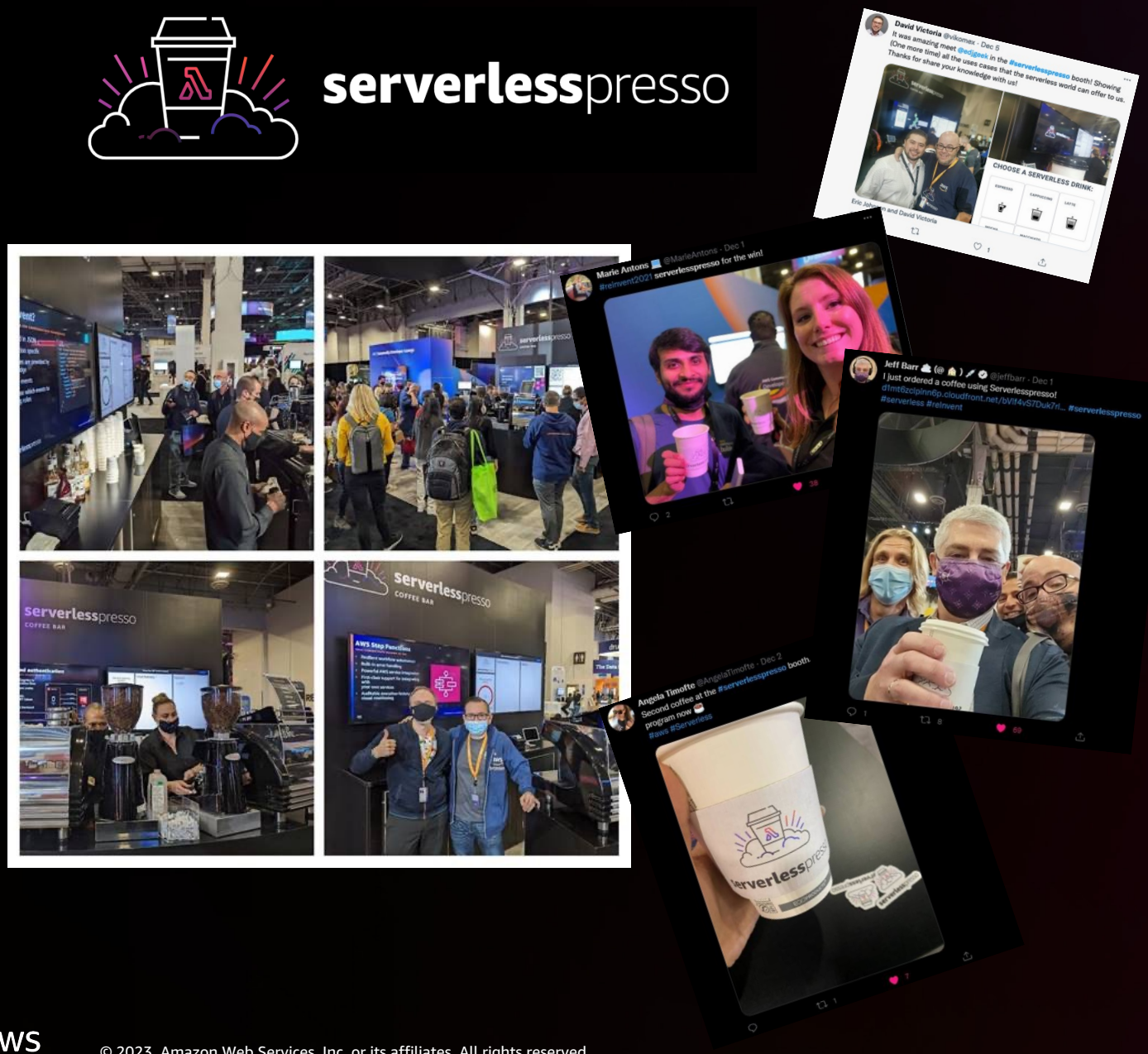
Startup Solutions Architect
AWS

Agenda

1. Intro to Serverlesspresso
2. Design decisions
3. Lessons learned
4. Useful patterns

serverlesspresso

An event-driven coffee ordering app
built with serverless architecture



- 1,920 drinks at re:Invent 2021
- 71 drinks per hour
- Appeared at AWS Summits, EDA Day, GOTO, and others
- Averages 1,000 drinks per day

- 1,806 drinks at Web Summit Rio 2023 🇧🇷

What is Serverlesspresso?

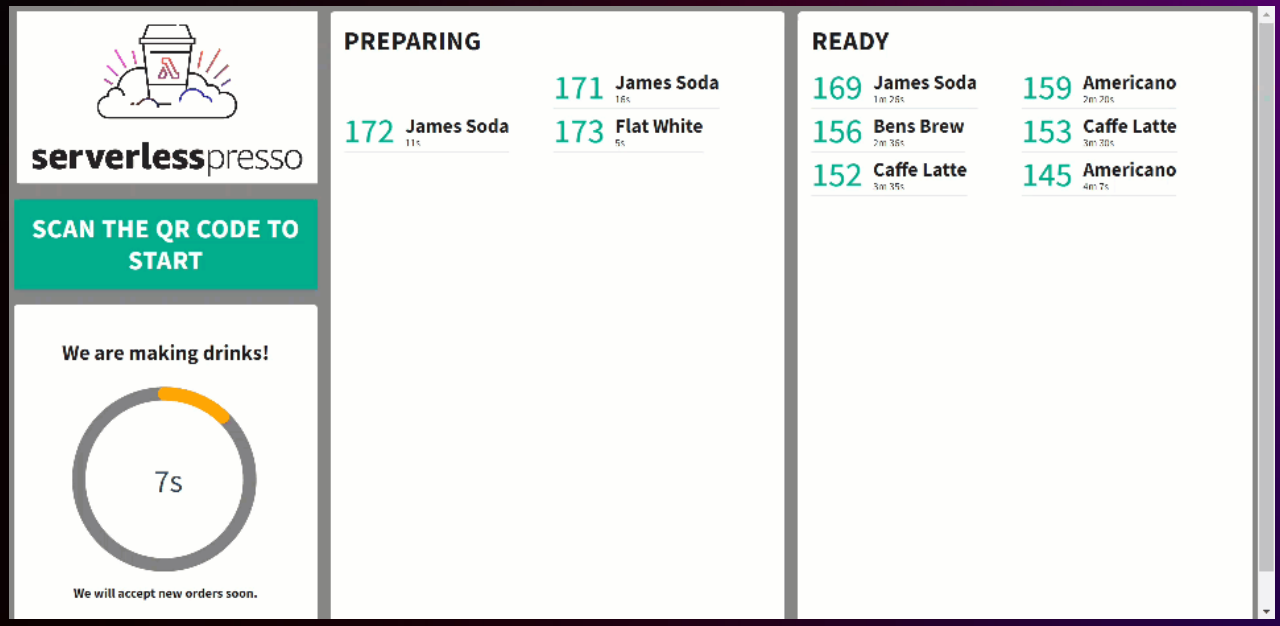
The workflow

1. Scan a dynamic QR code
2. Place an order with your mobile device
3. The order appears on the monitor and the barista's tablet app
4. You get a notification when the drink is ready

The Display Web App

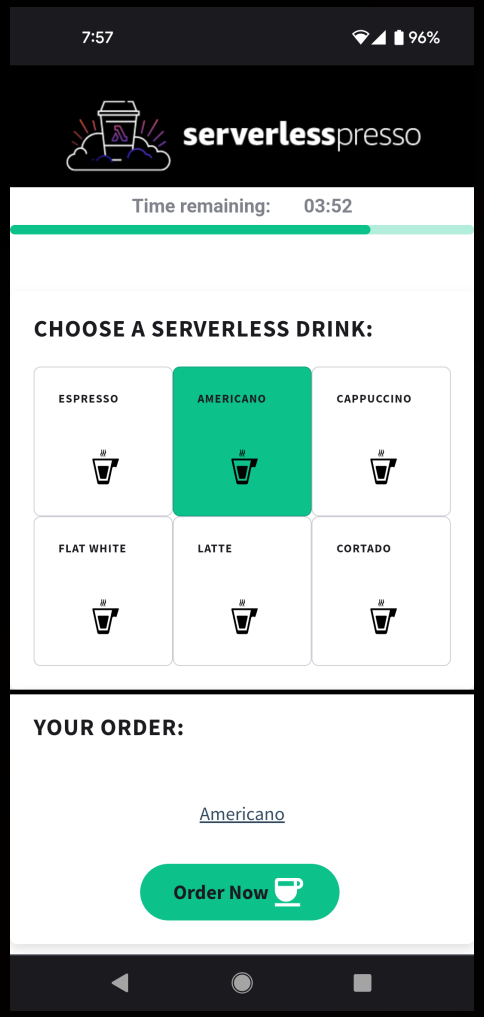
VUE.JS APPS HOSTED WITH AWS AMPLIFY CONSOLE

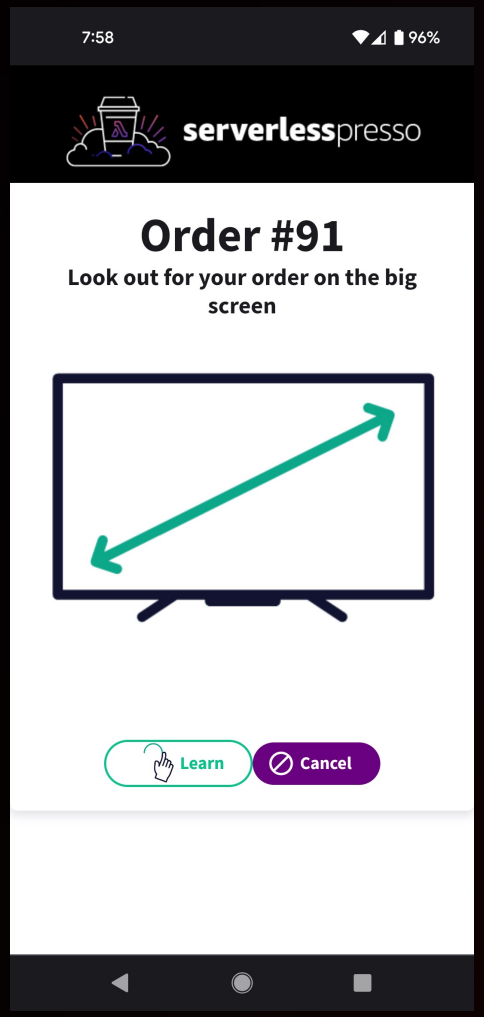
- Shows new barcode every 5 mins
- Receives order status updates
- Listens for store open/close events

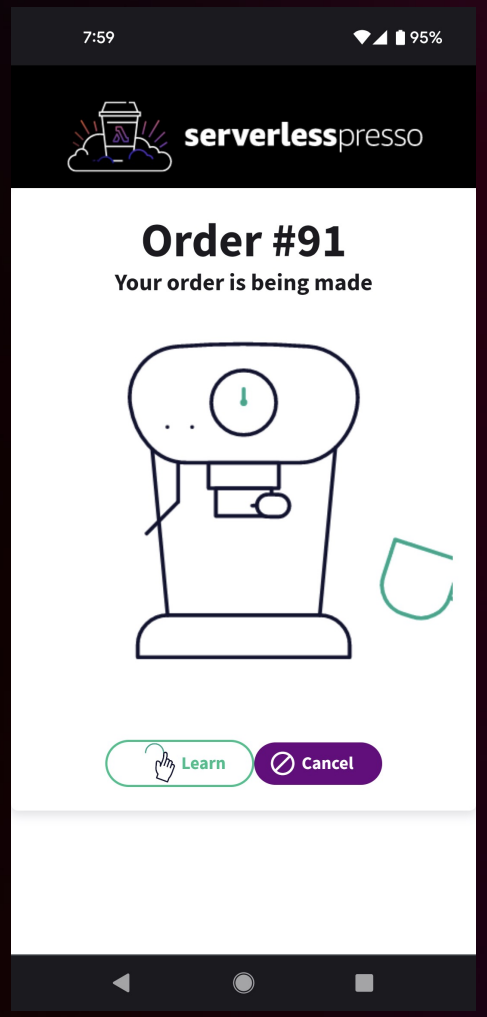


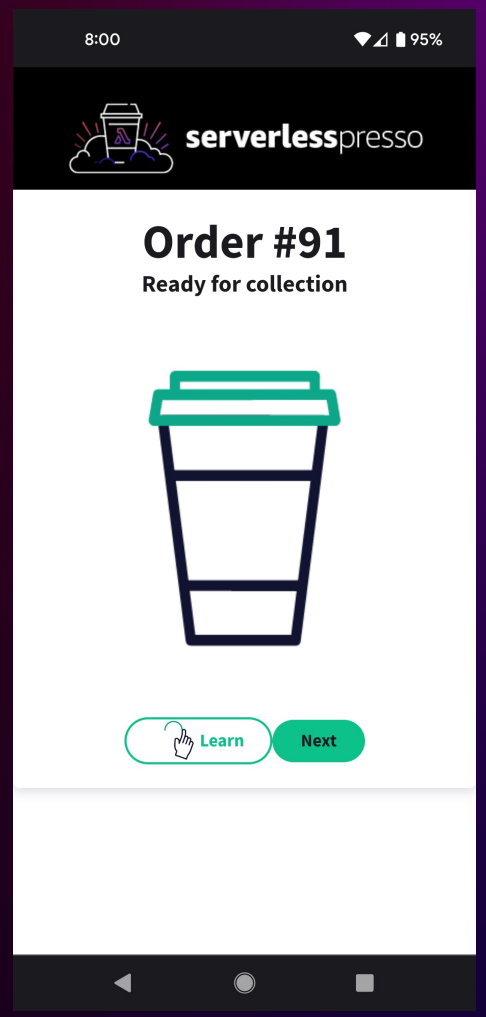
The Ordering Web App

VUE.JS APPS HOSTED WITH AWS AMPLIFY CONSOLE





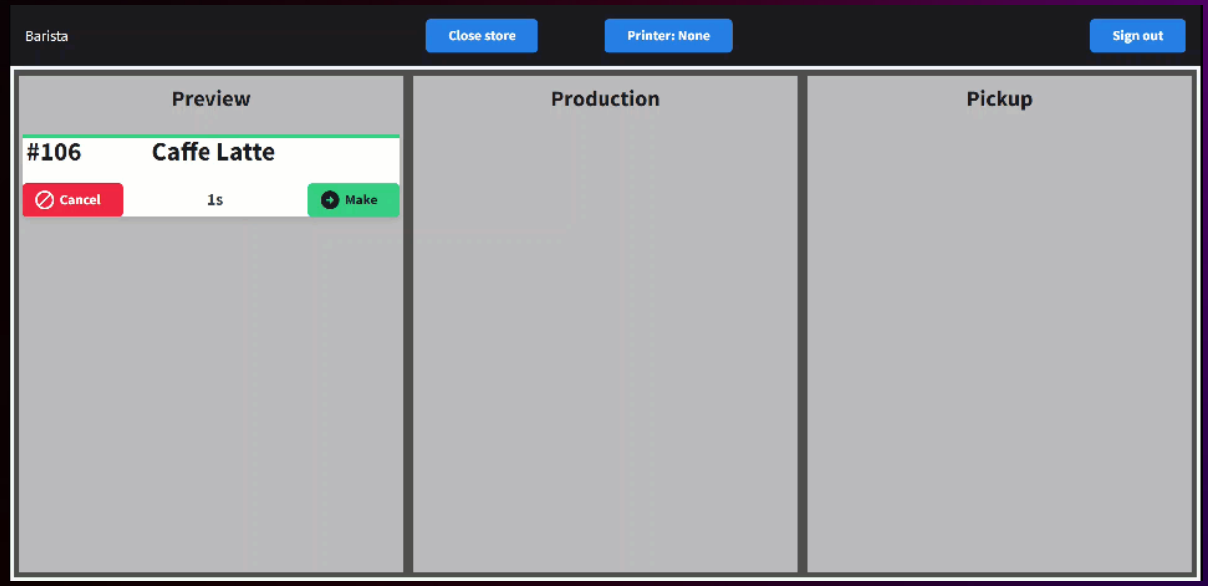




The Barista Web App

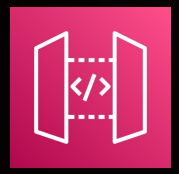
VUE.JS APPS HOSTED WITH AWS AMPLIFY CONSOLE

- Shows incoming orders
- Allows baristas to pick up incoming orders
- Enables order completion or cancellation
- Enables store open/close events
- Prints tickets



AWS services used

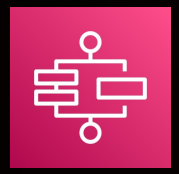
AWS Amplify console



Amazon API Gateway

Amazon DynamoDB

Amazon EventBridge



AWS Step Functions

AWS IoT Core

AWS Lambda

Design decisions

Guidelines we used

Architecture goals

- Minimal code
- Extensibility
- Scalability
- Cost efficiency

Tenets

- Each team member responsible for one component
- No implementation sharing
- Each microservice has API/events – no data sharing

Defining the workflow

The process for making a drink

1) Customer scans the barcode
2) Check the store is open
3) Get barista capacity
4) Wait for the customer order – cancel if > 5 mins
5) Generate an order number
6) Wait for barista to make the drink – cancel if > 15 mins
7) Handle cancelation by the customer or barista

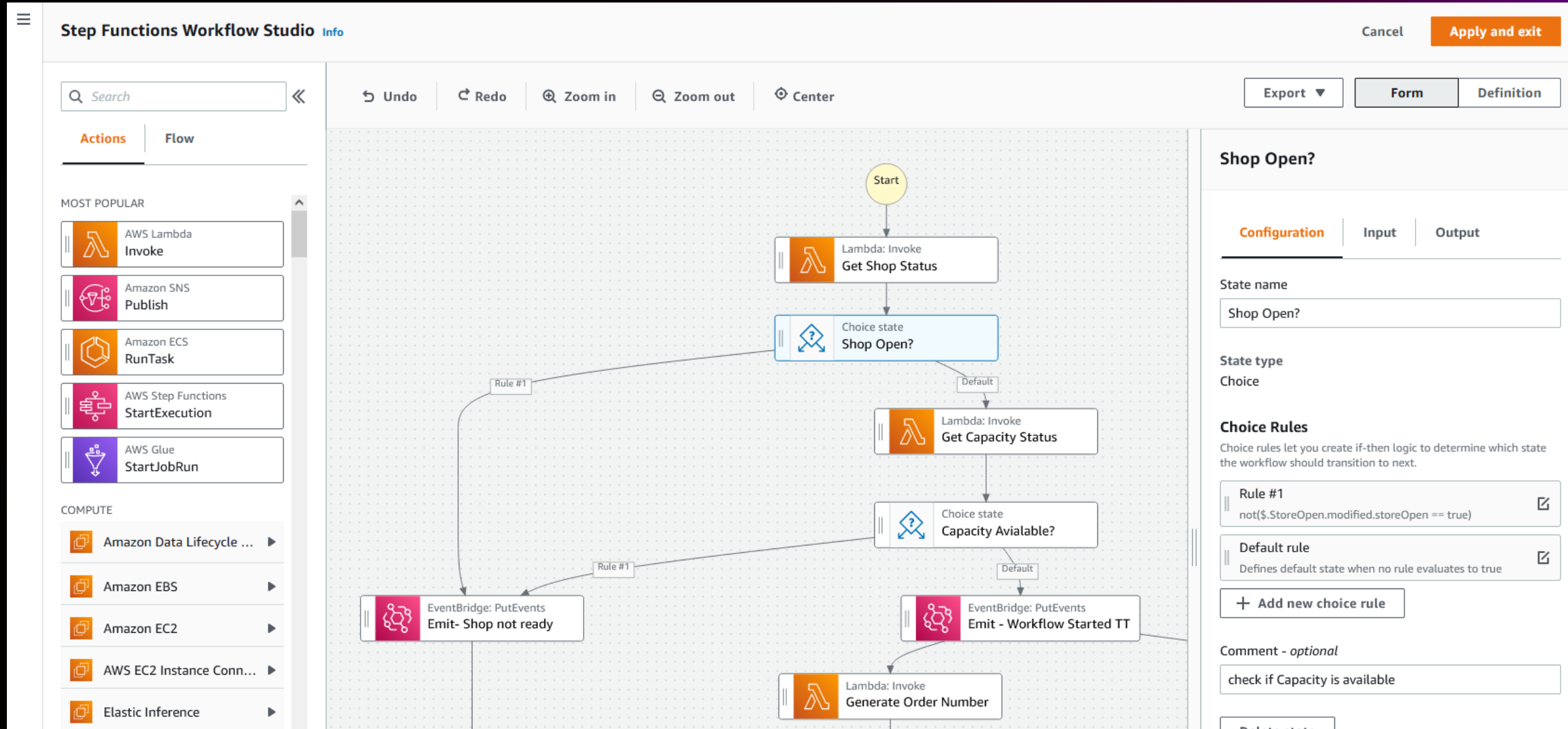
Write it all in (pseudo)code?

```
// Order acceptance

if (isStoreOpen) {
  if (isBaristaBusy) {
    saveOrderToDynamoDB(timeStamp)
  } else {
    rejectOrder('Barista busy')
  }
} else {
  rejectOrder('Store not open')
}
```

```
// Check for timeouts every minute

const orders = getAllOpenOrdersFromDDB
const now = currentTime

for (order in orders) {
  // Did customer time out?
  if (order.waitingForCustomer &&
     (now - order.timeStamp) > 5 mins)
  {
    rejectOrder('Customer timed out')
    updateDynamoDB
  }

  // Then check for barista time out
  ...
}
```

Instead design visually with Workflow Studio

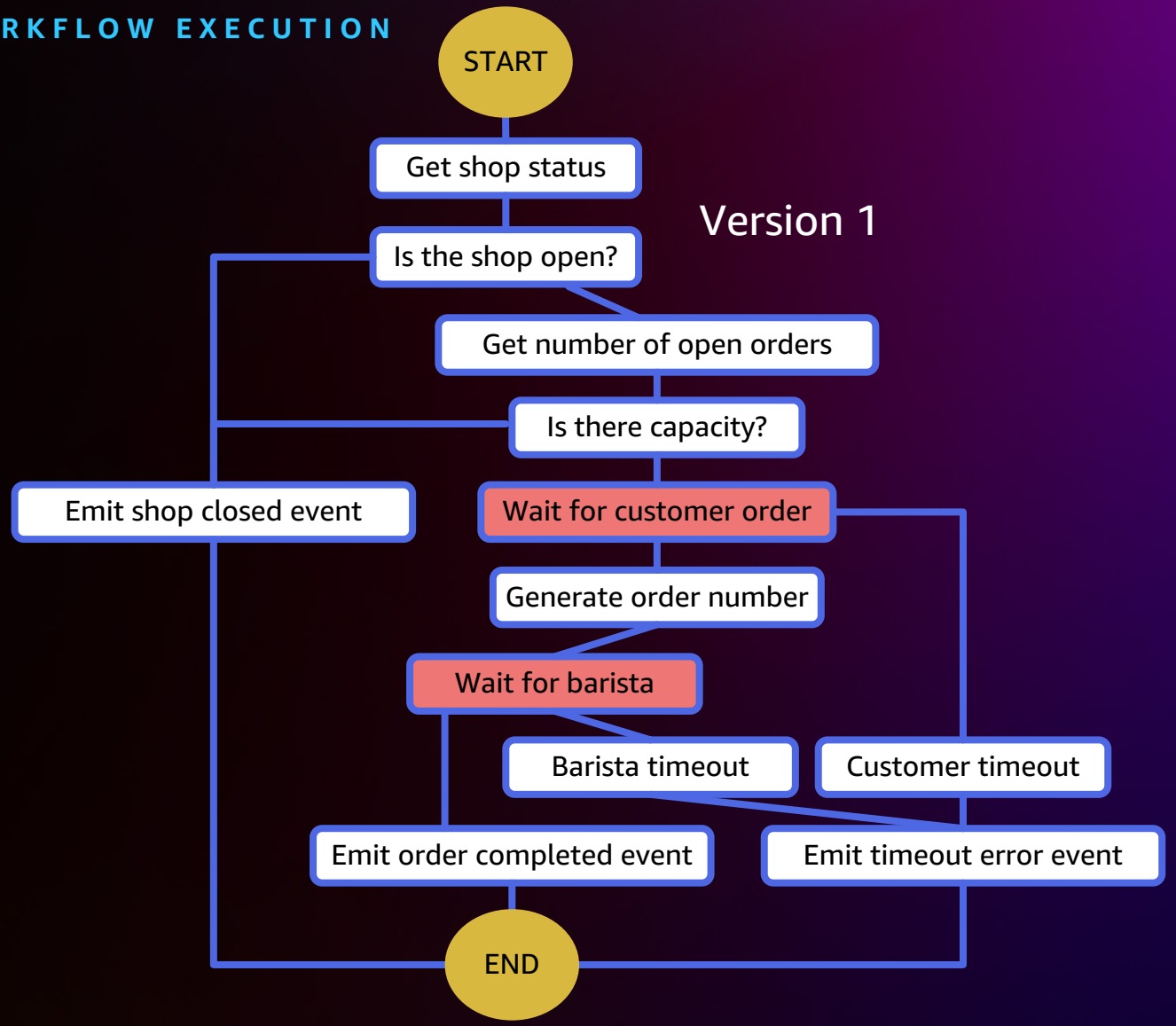
REPLACE SPAGHETTI CODE WITH STATE MACHINES



Managing each coffee's journey

USING AWS STEP FUNCTIONS TO MANAGE EACH WORKFLOW EXECUTION

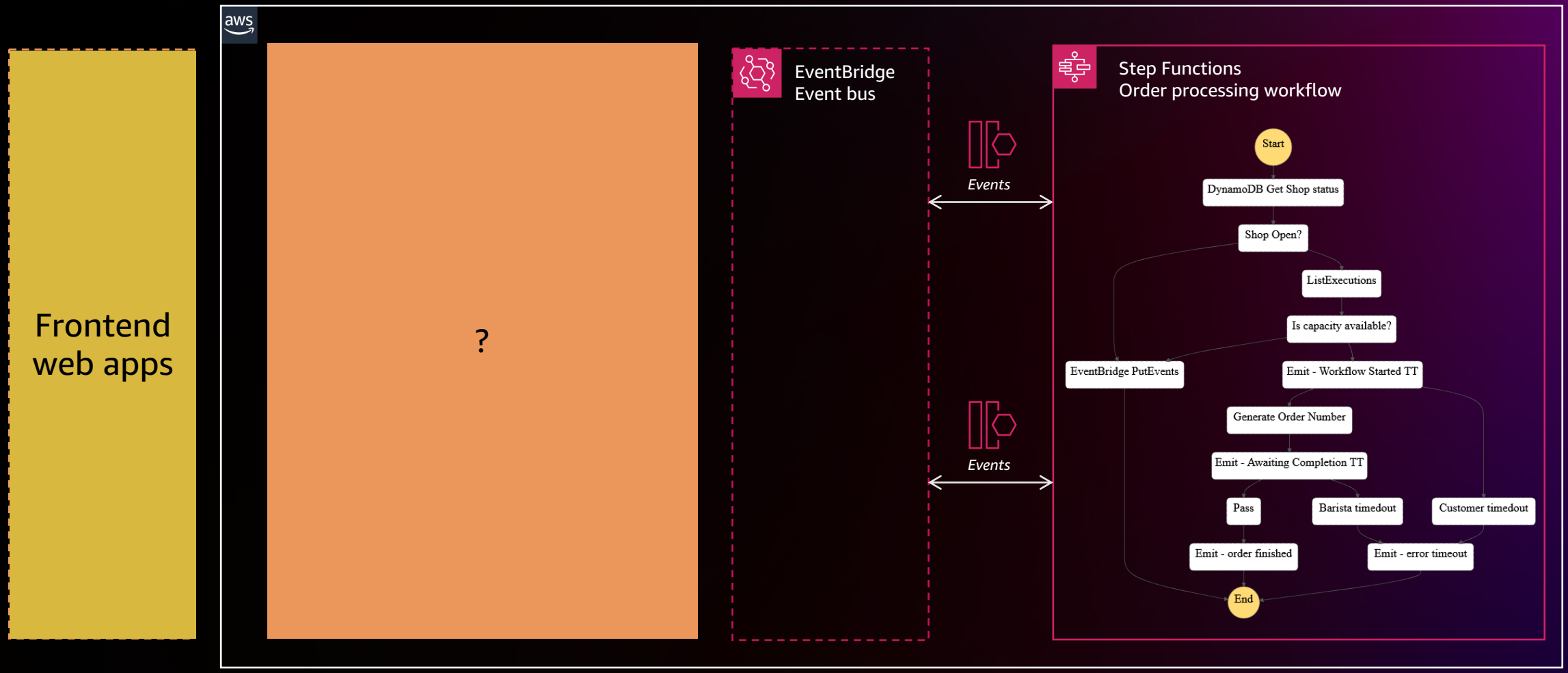
- Ensure store is open and baristas have capacity
- Allow customer 5 minutes to order before timing out/cancelling
- Allow barista 15 minutes to make before timing out/cancelling
- Uses Lambda functions for custom logic



Version 1

serverlesspresso

Architecture so far

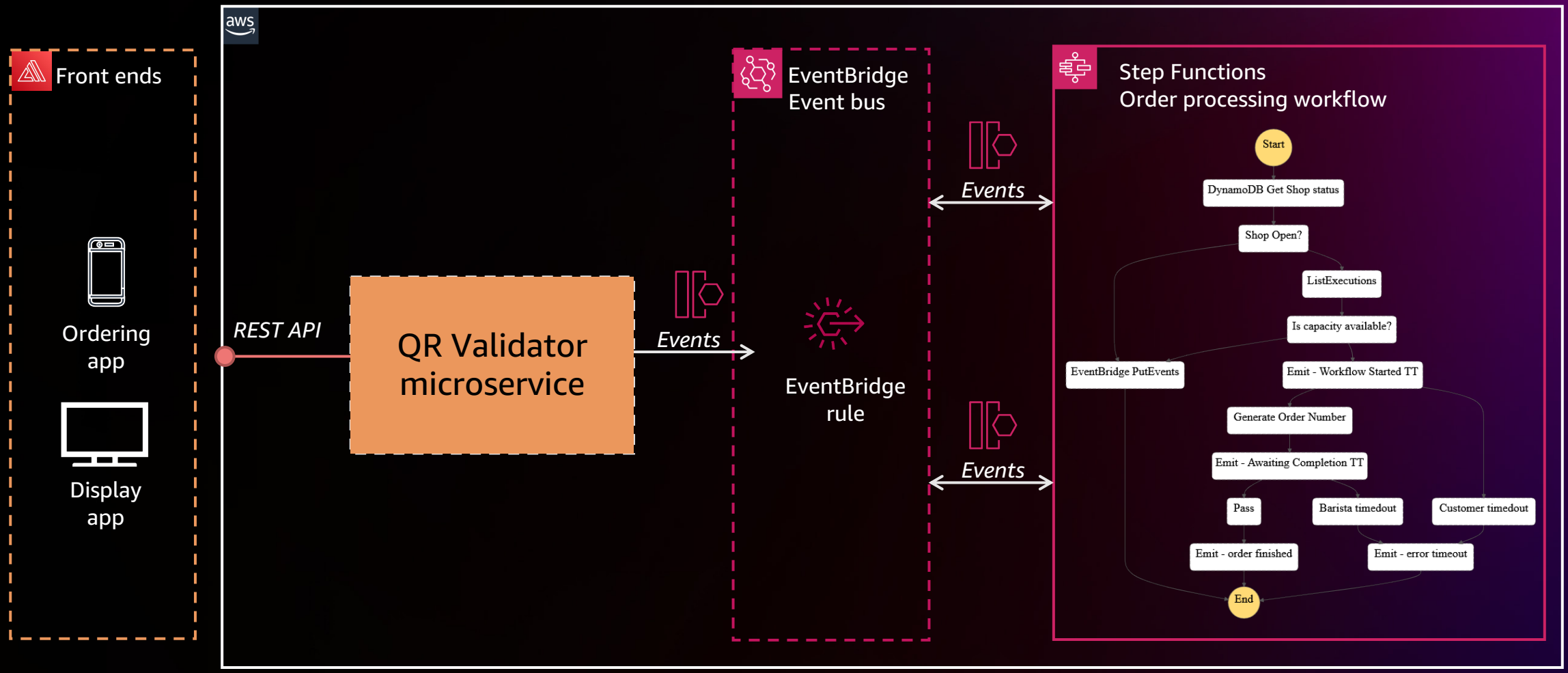


Starting the order

The QR validator microservice

- Throttles the queue to 10 drinks per interval
- Interval is 5 minutes by default
- Generates new QR codes
- QR disappears once all scans "used”
- Successful scan starts the order

Introducing the QR validator microservice



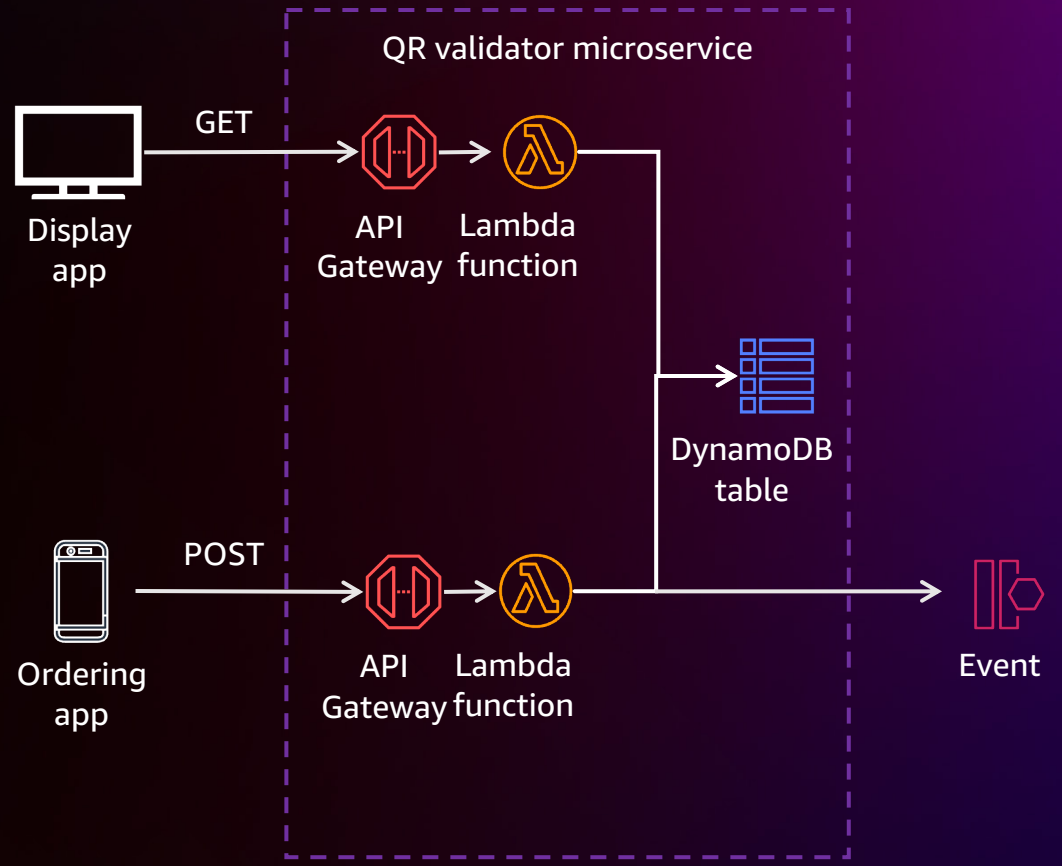
Inside the QR validator service

Generate

- Display app requests new QR code
- Random ID stored in DynamoDB

Validate

- Phone scans the QR code
- ID is passed as a parameter
- Returns success or failure
- Emits an event if successful



The QR validator DynamoDB table

PK	Available tokens	End_ts	Last_code	Last_id	Start_ts
1234423	10	1645200599999	41g_-KJHGT	1234423	1645200300000
1234123	8	1645628399999	6KJH_-FJ5Lh	1234123	1645628100000
5412322	1	1645449599999	91HHFFJHF	5412322	1645449300000
3435657	2	1645435199999	OCZomT756	3435657	1645434900000

When a valid QR is scanned, it decrements **Available tokens**

Managing orders

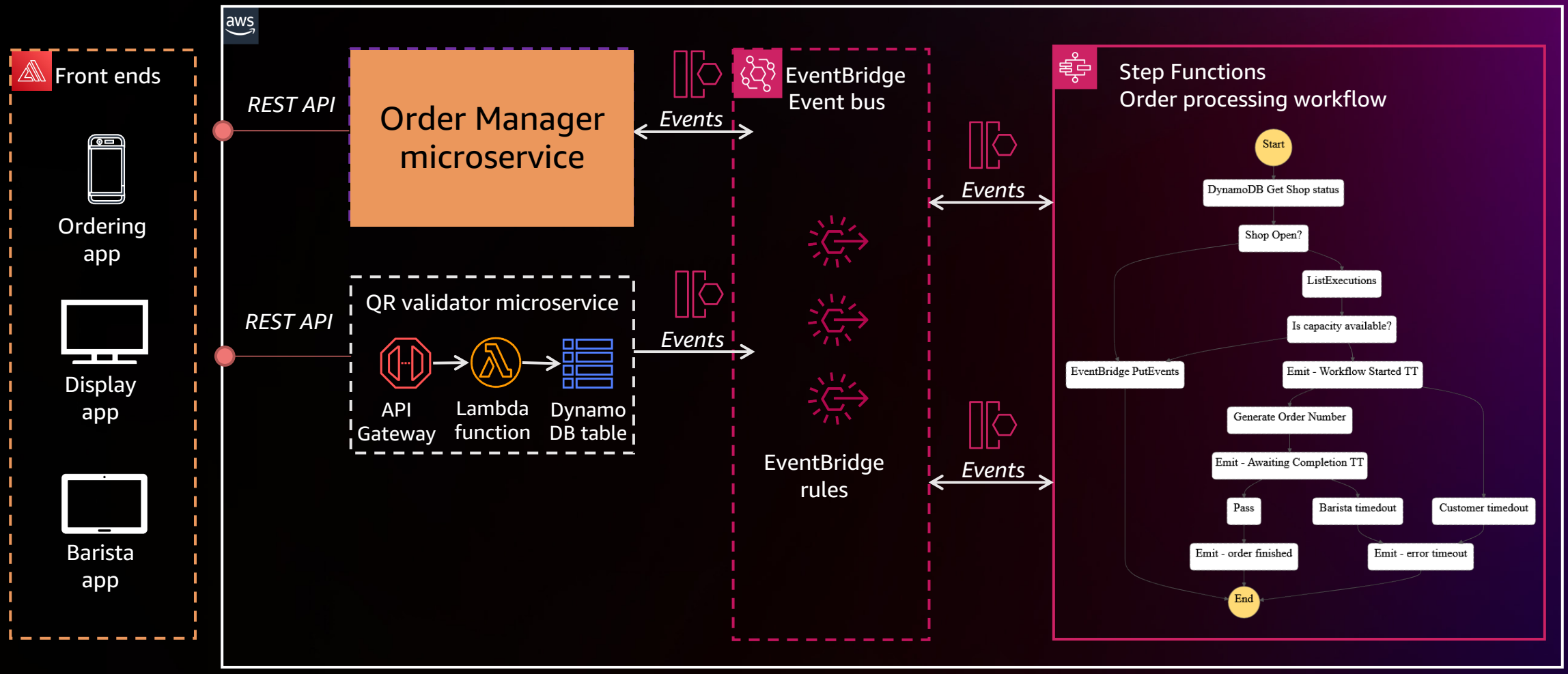
Need to

- Update and cancel orders
- Store and use TaskTokens
- Get list of open or completed orders

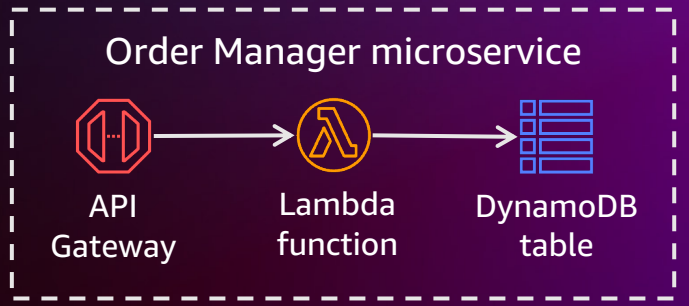
Should we

- Build a monolithic workflow?
- Keep the TaskTokens at the client?
- Query all open workflows?

Introducing the Order Manager microservice



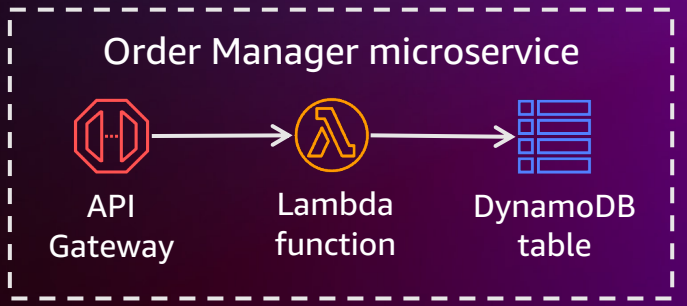
Updating orders



- Persists each order to a DynamoDB table
- Item updated at various stages of the order lifecycle
- Responsible for managing task tokens
- GSI on OrderState attribute for querying open/completed orders

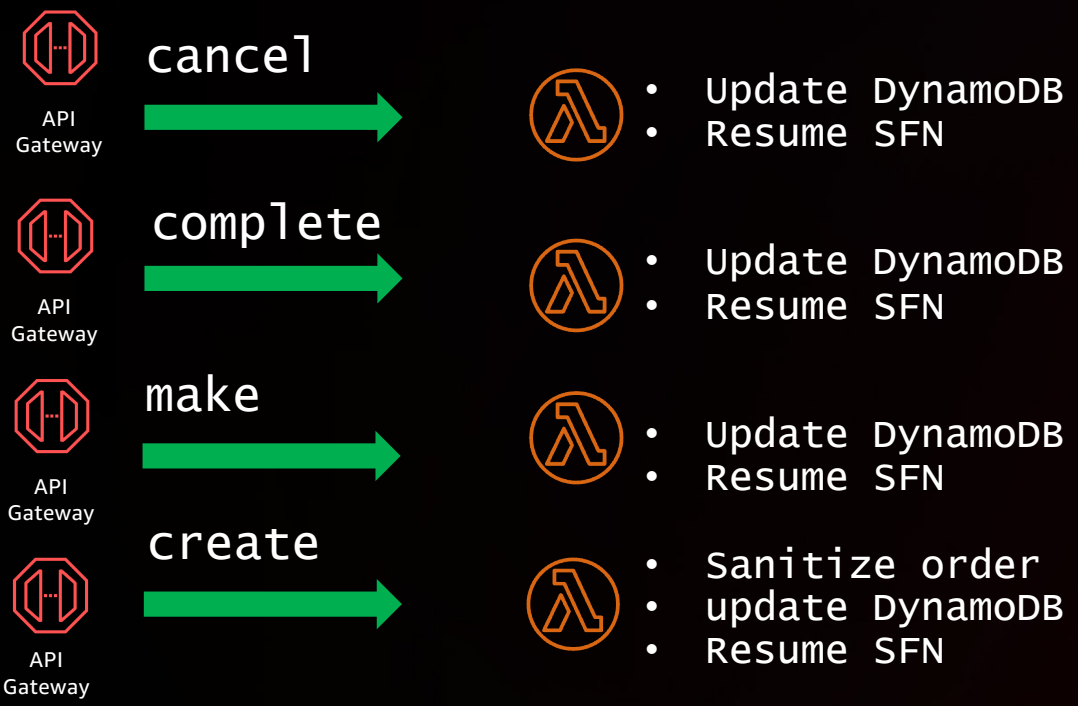
PK	SK	TS	UID	OrderNo	TaskToken	OrderState	DrinkOrder
Orders	2	1645726346199	1	10	AAAAKgAAAAIAAAAAAAAAAAlp4um0gPw/FO3rqpCDvlEAL+l/h+	COMPLETED	{"userId":"1","drink":"Cappuccino","modifiers":[],"icon":"barista-icons_cappuccino-alternative"}

Order Manager service



Version 1

Each operation invokes a Lambda Function



Application grew more complex over time, performing multiple tasks to handle increasingly complex business logic

Led to

- Tightly coupled code base
- Slower release cadence
- Poor discoverability
- Additional complexity

Order Manager service: Direct integration

Version 1.5

/myOrders - GET

/orders - GET

/orders/{id} - GET

Front ends query *Order* table directly from API Gateway

Velocity mapping templates modify the incoming request


```
#set($subFromJWT = $context.authorizer.claims.sub)

{
    "TableName": "serverlesspresso-order-table",
    "IndexName": "GSI-userId",
    "KeyConditionExpression": "#USERID = :USERID",
    "ExpressionAttributeNames": {
    "#USERID": "USERID"
    },
    "ExpressionAttributeValues": {
    ":USERID": {
      "S": "$subFromJWT"
    }
    },
    "ScanIndexForward": true,
    "ProjectionExpression": "PK, SK, orderNumber, robot, drinkOrder,
ORDERSTATE, TS"
}
```

Order Manager service: CRUD operations

Version 1.5

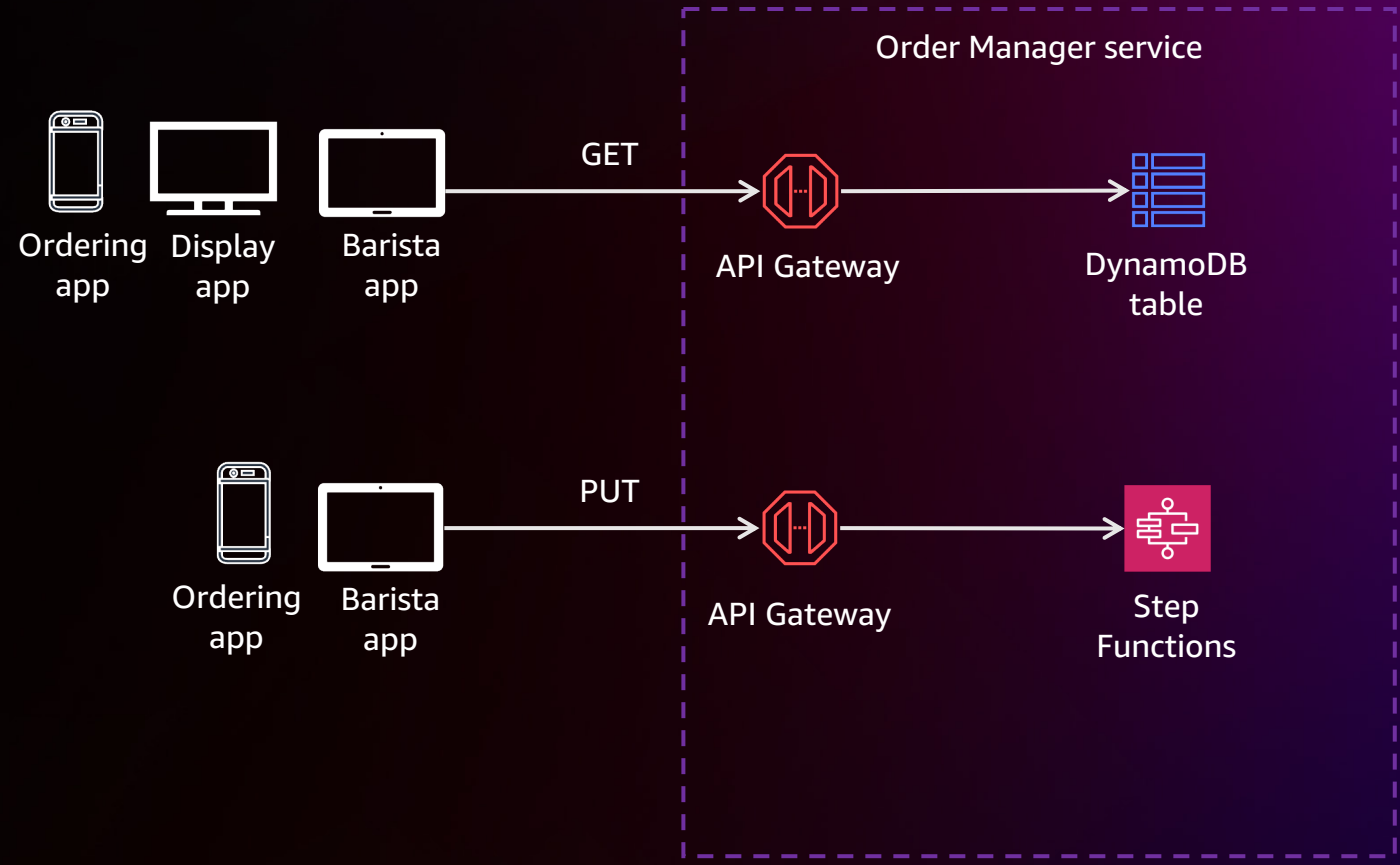
/myOrders - GET

/orders - GET

/orders/{id} - GET

/orders/{id} - PUT

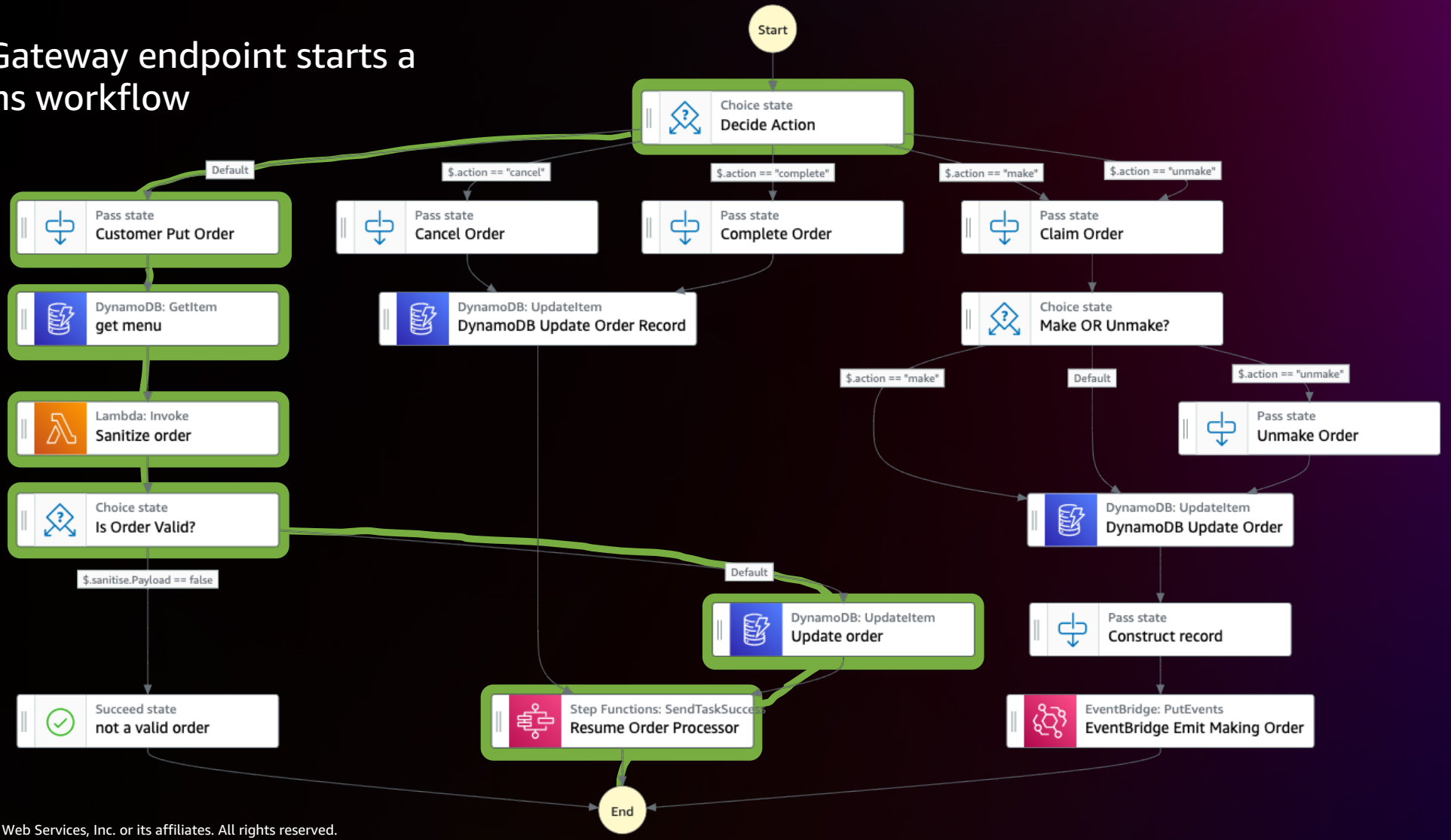
Updates made to *Order* table via a PUT request



Order Manager service as a workflow

Version 2

A single API Gateway endpoint starts a Step Functions workflow



Notifying the web apps

Need to

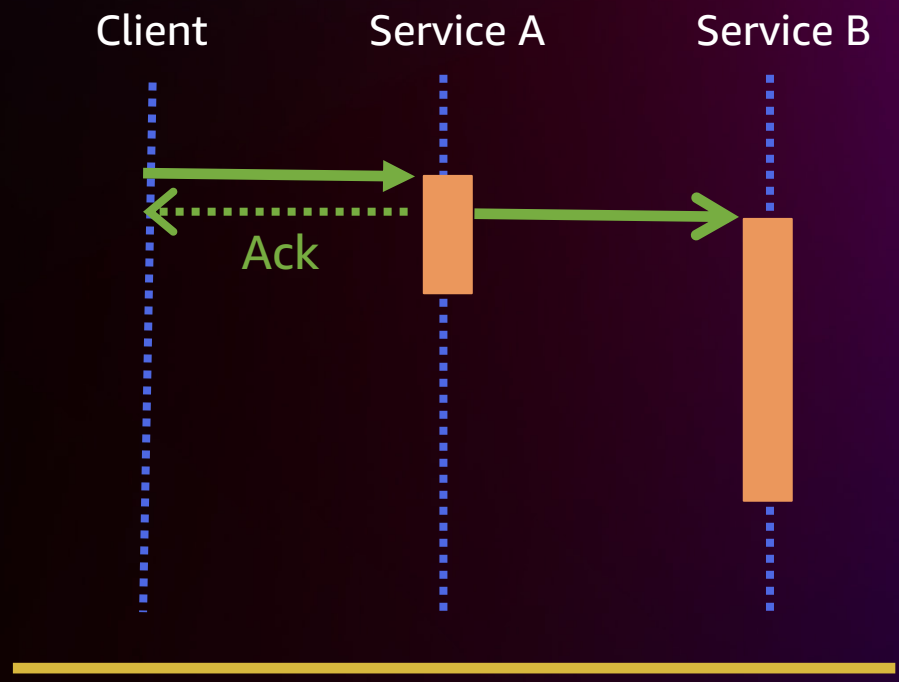
- Keep web apps in sync with order state
- Respond to global events, such as store open/close
- Approximate real-time without refreshing
- Be resilient, given mobile network dropout

Should we

- Create polling APIs?
- Use mechanisms like SMS?
- Use API Gateway WebSockets?

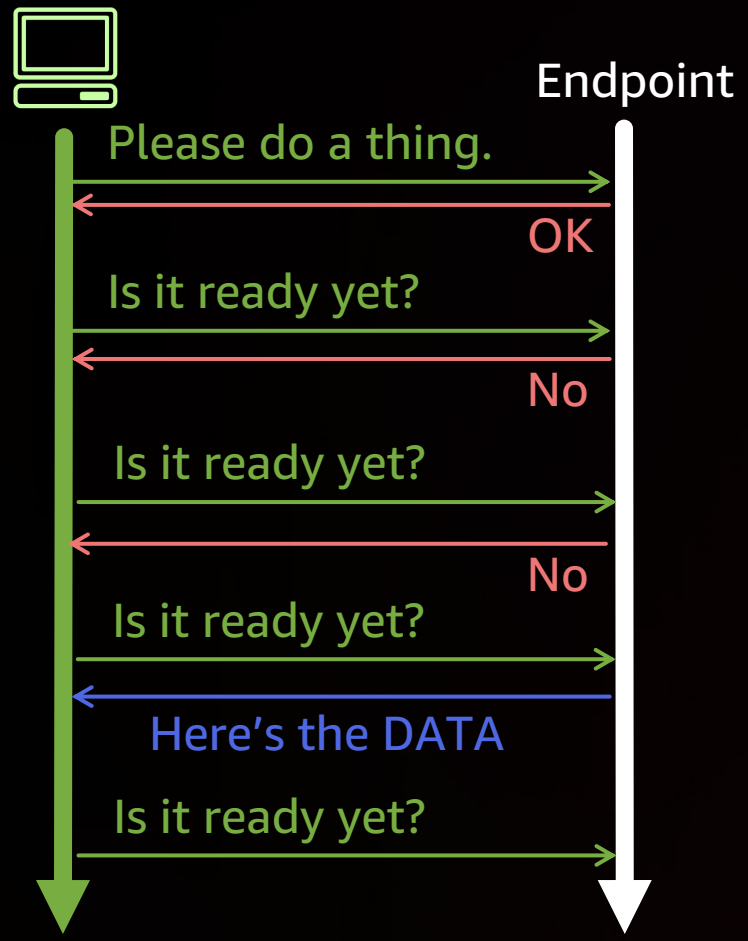
Handling response values and state for asynchronous requests

No return path to provide further information beyond the initial acknowledgement



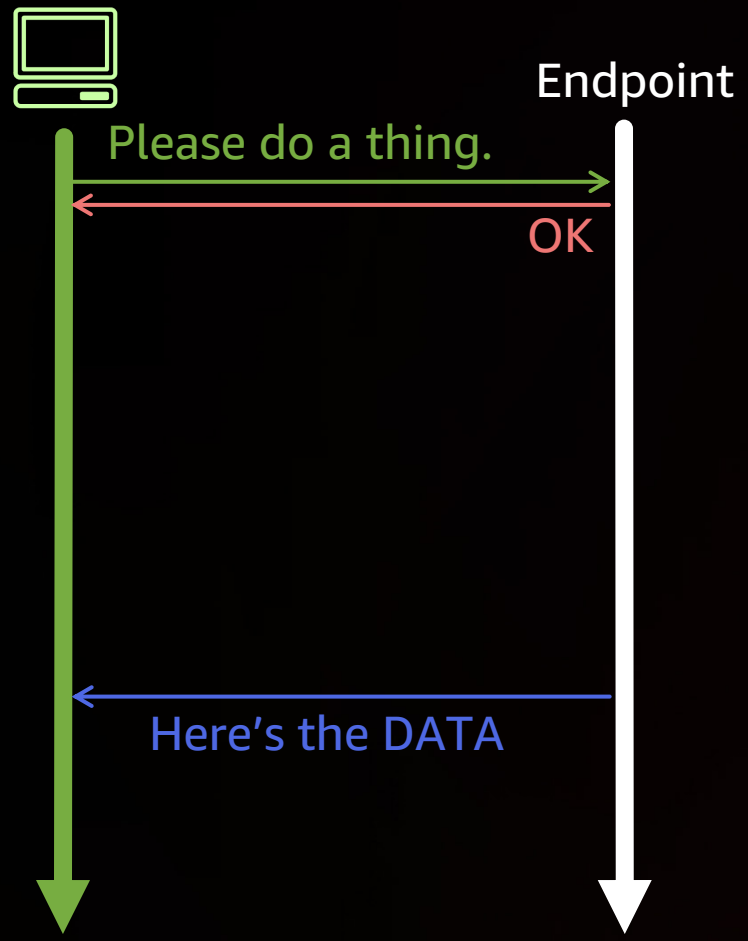
Asynchronous events

Tracking an inflight request: Polling



- Initial request returns a tracking identifier
- Create a second API endpoint for the front end to check the request status, referencing the tracking ID
- Use DynamoDB to track the state of the request
- Simple mechanism to implement
- Can create many empty calls
- Delay between availability and front-end notification

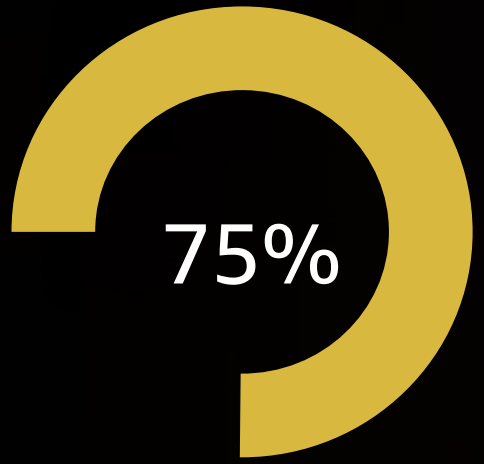
Tracking an inflight request: WebSocket



- A bidirectional connection between the front end client and the backend service
- Your backend services can continue to send data back to the client by using a WebSocket connection
- Closer to real time
- Reduces the number of messages between the client and backend system
- Often more complex to implement

Using AWS IoT Core for real-time messaging

WEB APPLICATIONS OFTEN REQUIRE PARTIAL INFORMATION

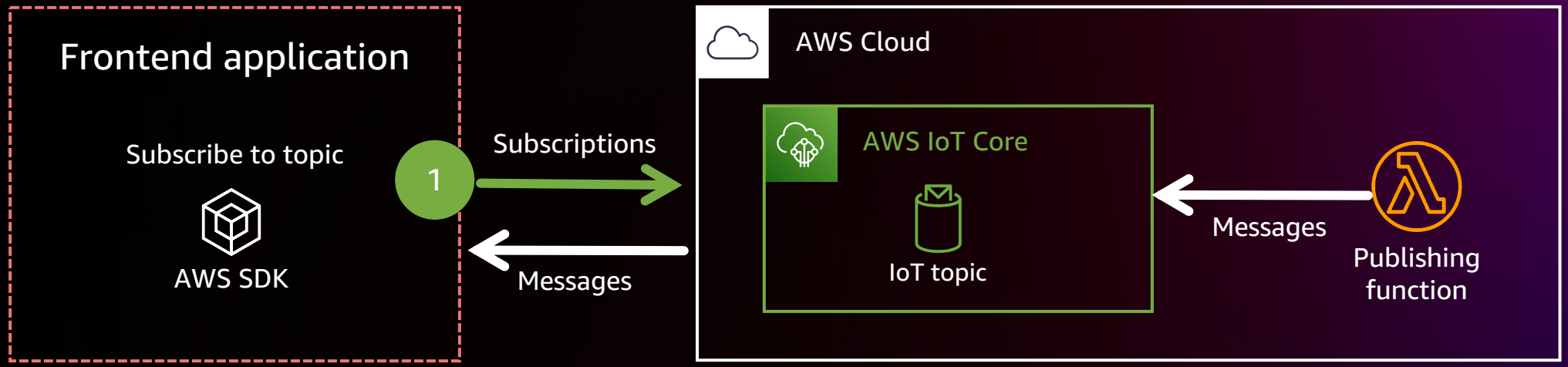


Completion percentages

Continuous data changes

Using AWS IoT Core for real-time messaging

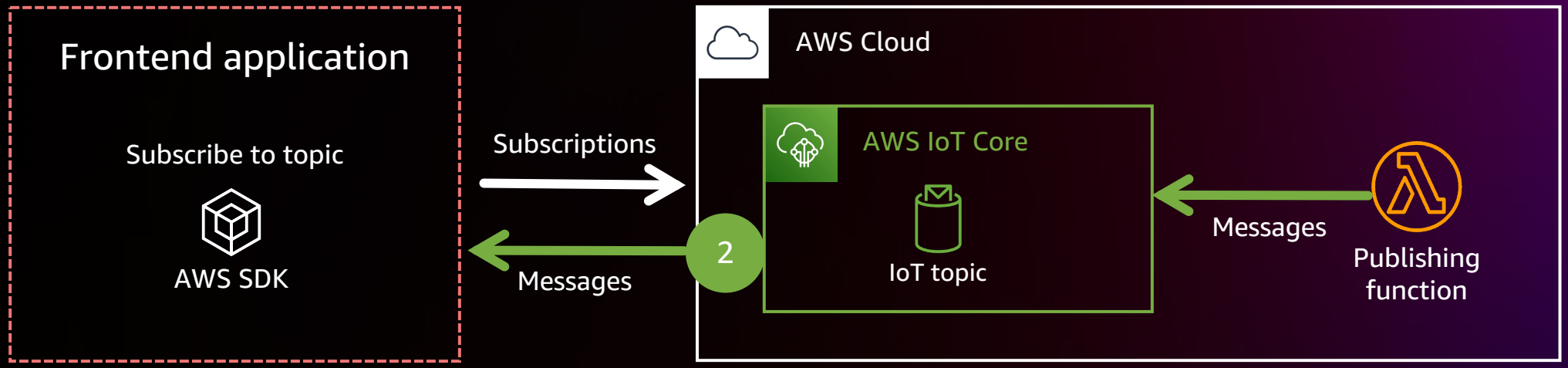
Frontend application uses pub-sub to “listen” for event updates



Frontend uses AWS SDK to subscribe to a topic based on user’s unique user ID

Using AWS IoT Core for real-time messaging

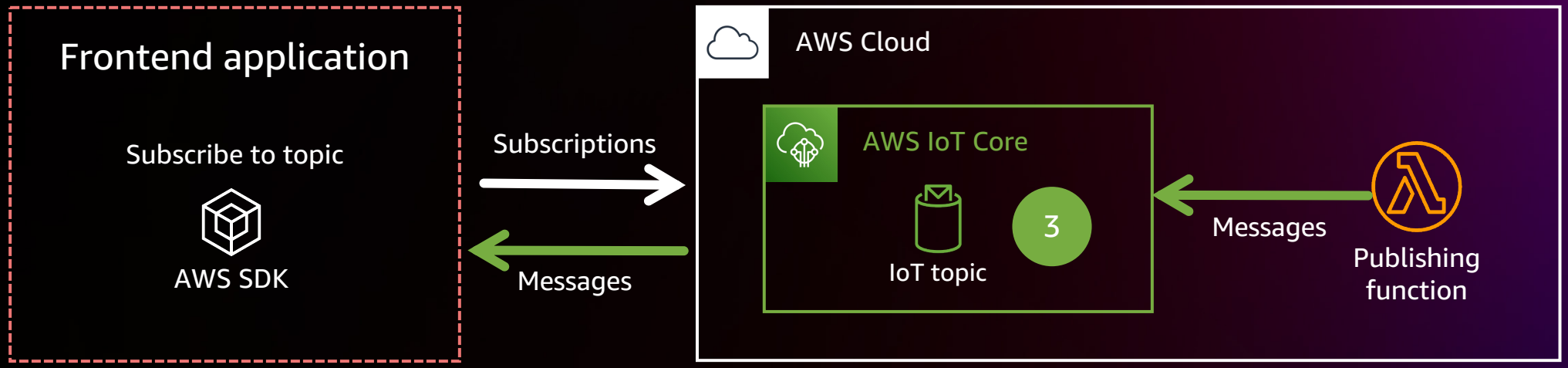
Frontend application uses pub-sub to “listen” for event updates



Receives messages published by the backend to this topic

Using AWS IoT Core for real-time messaging

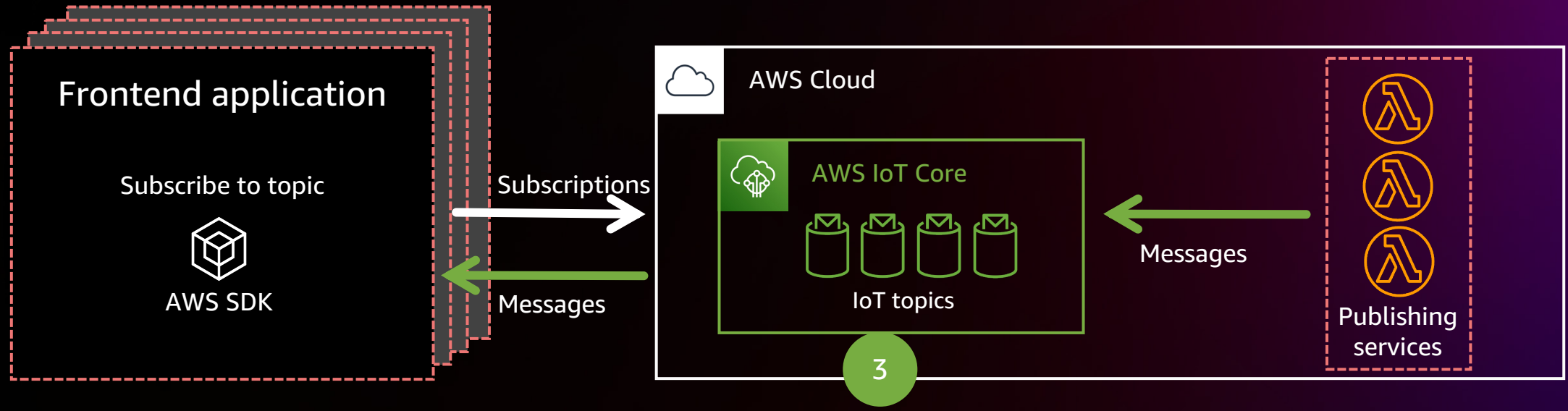
Frontend application uses pub-sub to “listen” for event updates



Messages are categorized using topics

Using AWS IoT Core for real-time messaging

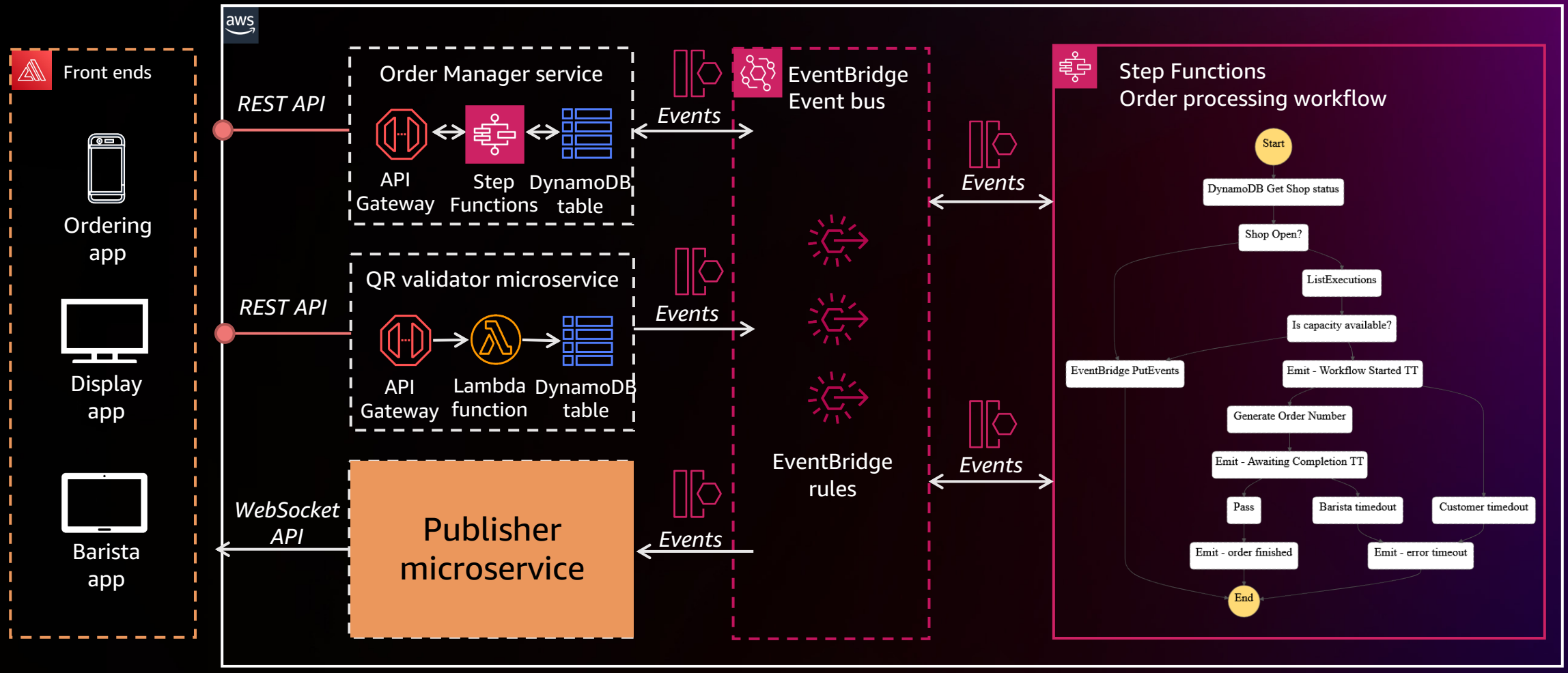
Frontend application uses pub-sub to “listen” for event updates



The AWS IoT Core service manages the WebSocket connection between publishers and subscribers

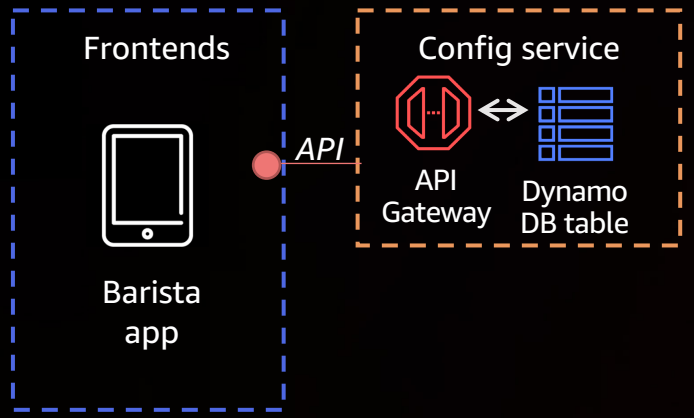
This enables fanout functionality to thousands frontend devices

Introducing the Publisher microservice



Combining multiple approaches for your frontend application

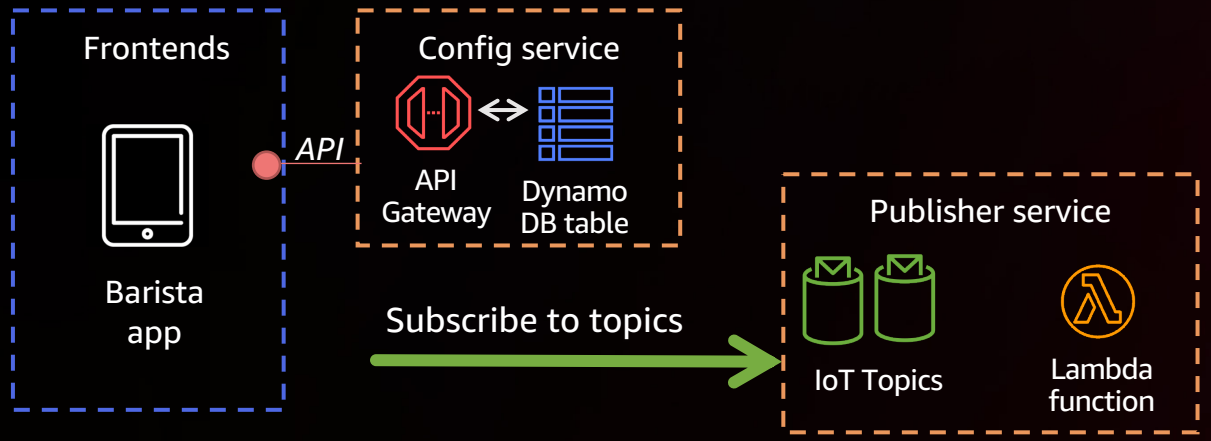
Many frontend applications can combine synchronous and asynchronous response models



Serverlesspresso sends an initial synchronous request to retrieve the current “state of things”

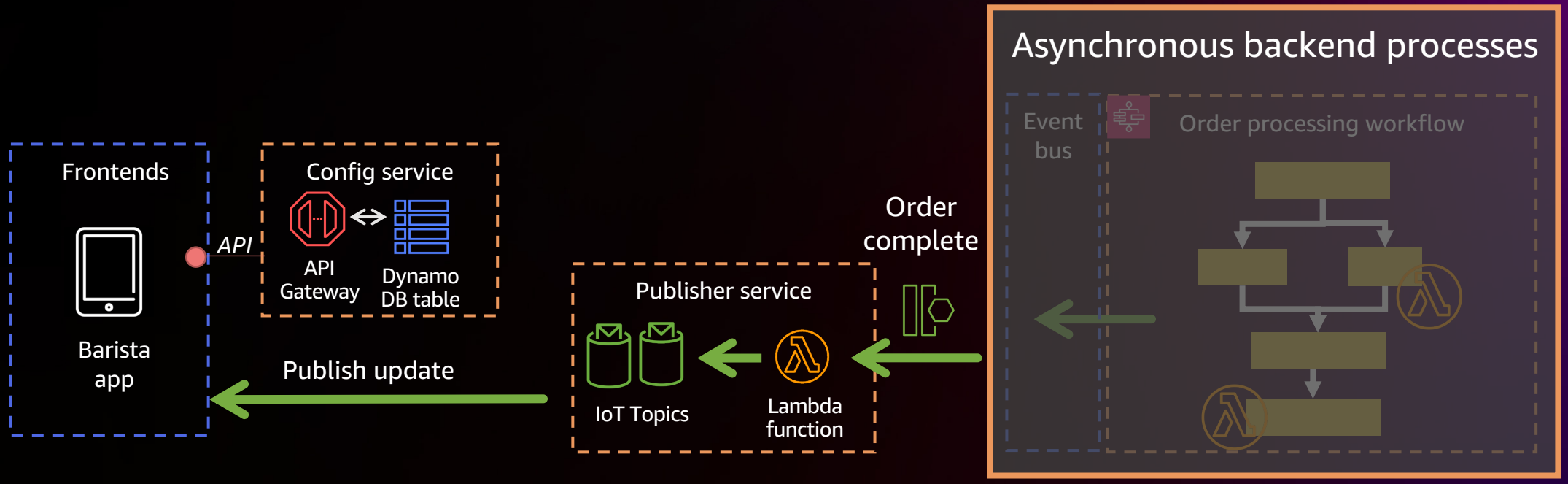
Combining multiple approaches for your frontend application

Many frontend applications can combine synchronous and asynchronous response models



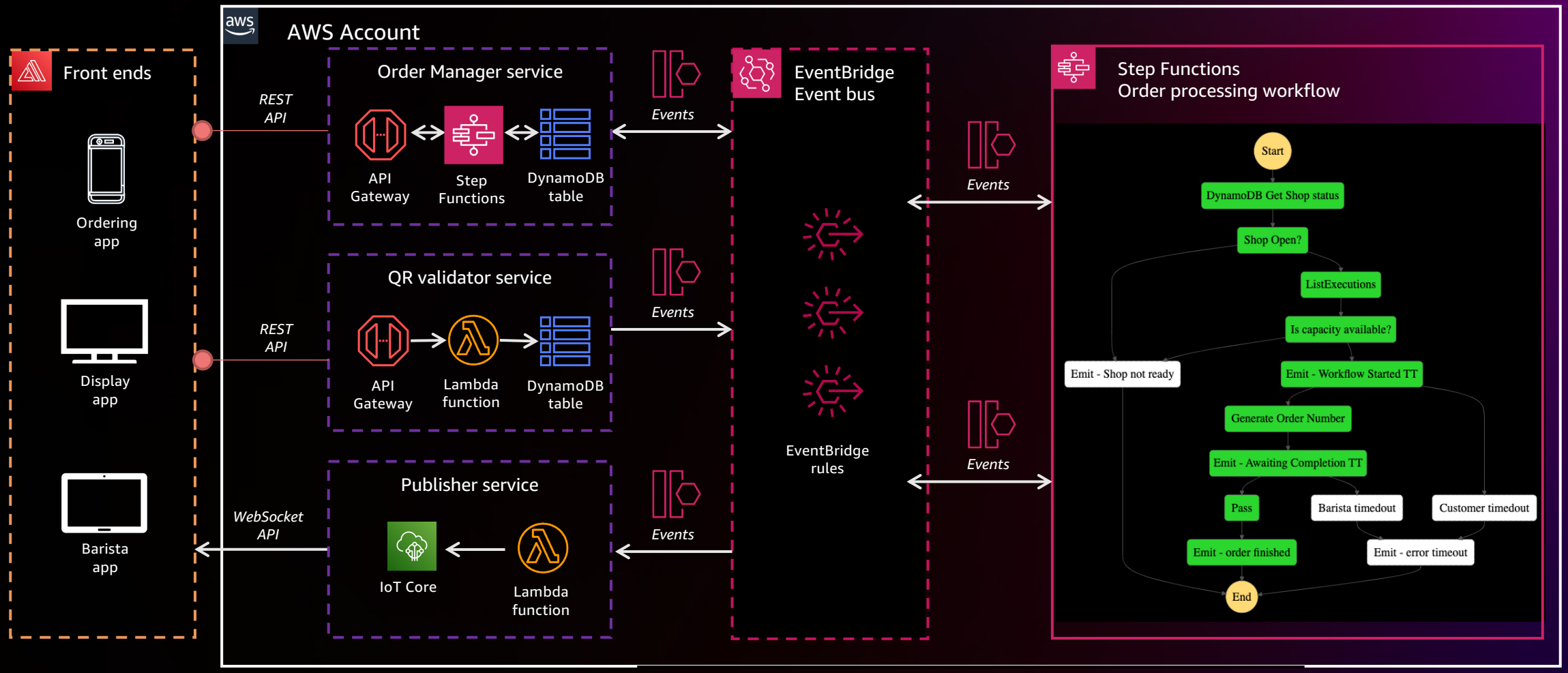
Simultaneously the frontend subscribes to global and user-based IoT topics

Combining multiple approaches for your frontend application



New orders are posted to the application, processed asynchronously, with updates published to the frontend via the IoT topic

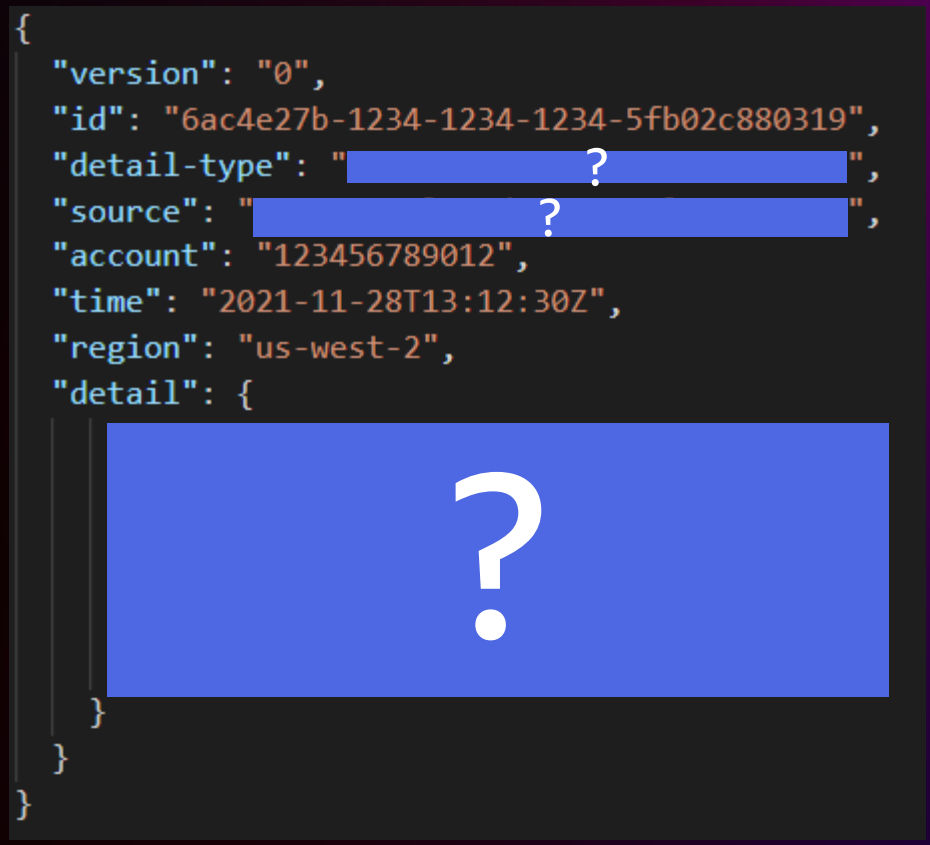
Final architecture



Lessons learned

What should events contain?

- Producers create events
- An event is defined in JSON
- Envelope attributes are provided by Amazon EventBridge
- `detail`, `detail-type`, and `source` are entirely your choice
- Fat events or thin events?
- Include metadata like versioning



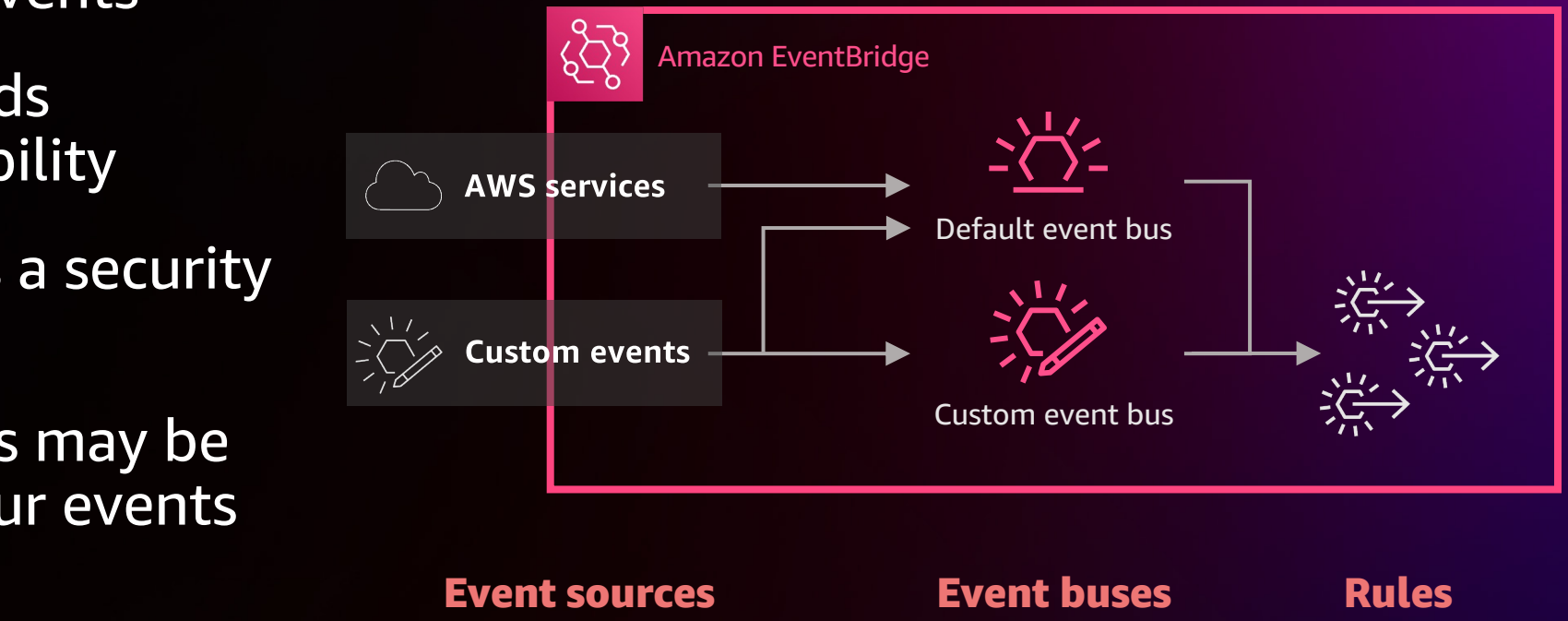
What events should you generate?

- Publishing more events than you currently use helps with extensibility
- Step Functions generates some events automatically; can also manually emit events
- Naming conventions: `Microservice.ThingThatHappened`

```
ConfigService.ConfigChanged
OrderJourney.AllEventsStored
OrderManager.MakeOrder
OrderManager.OrderCancelled
OrderManager.OrderCompleted
OrderManager.WaitingCompletion
OrderProcessor.OrderTimeOut
OrderProcessor.ShopNotready
OrderProcessor.WaitingCompletion
OrderProcessor.WaitingProduction
OrderProcessor.WorkflowStarted
QueueService.OrderCancelled
QueueService.OrderCompleted
QueueService.OrderStarted
Validator.NewOrder
```

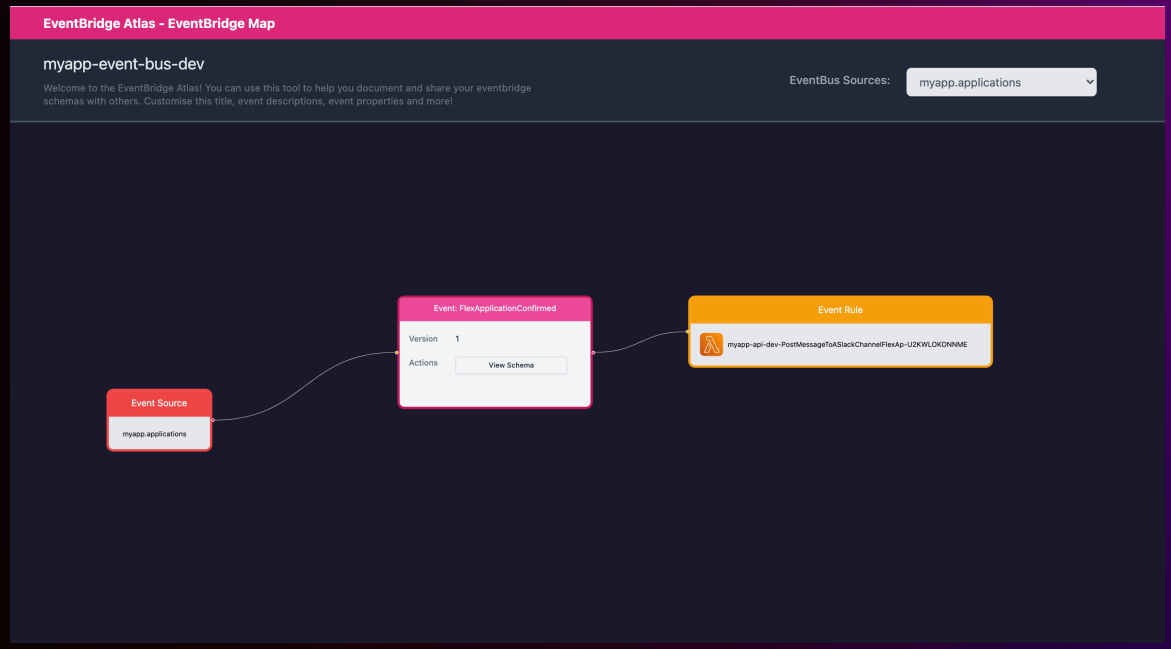
Amazon EventBridge: One bus or two?

- Default bus already exists – only destination for AWS events
- Adding events here aids discovery and extensibility
- Custom bus can act as a security boundary
- Trade-off: other teams may be unable to build off your events



Aiding discovery of events

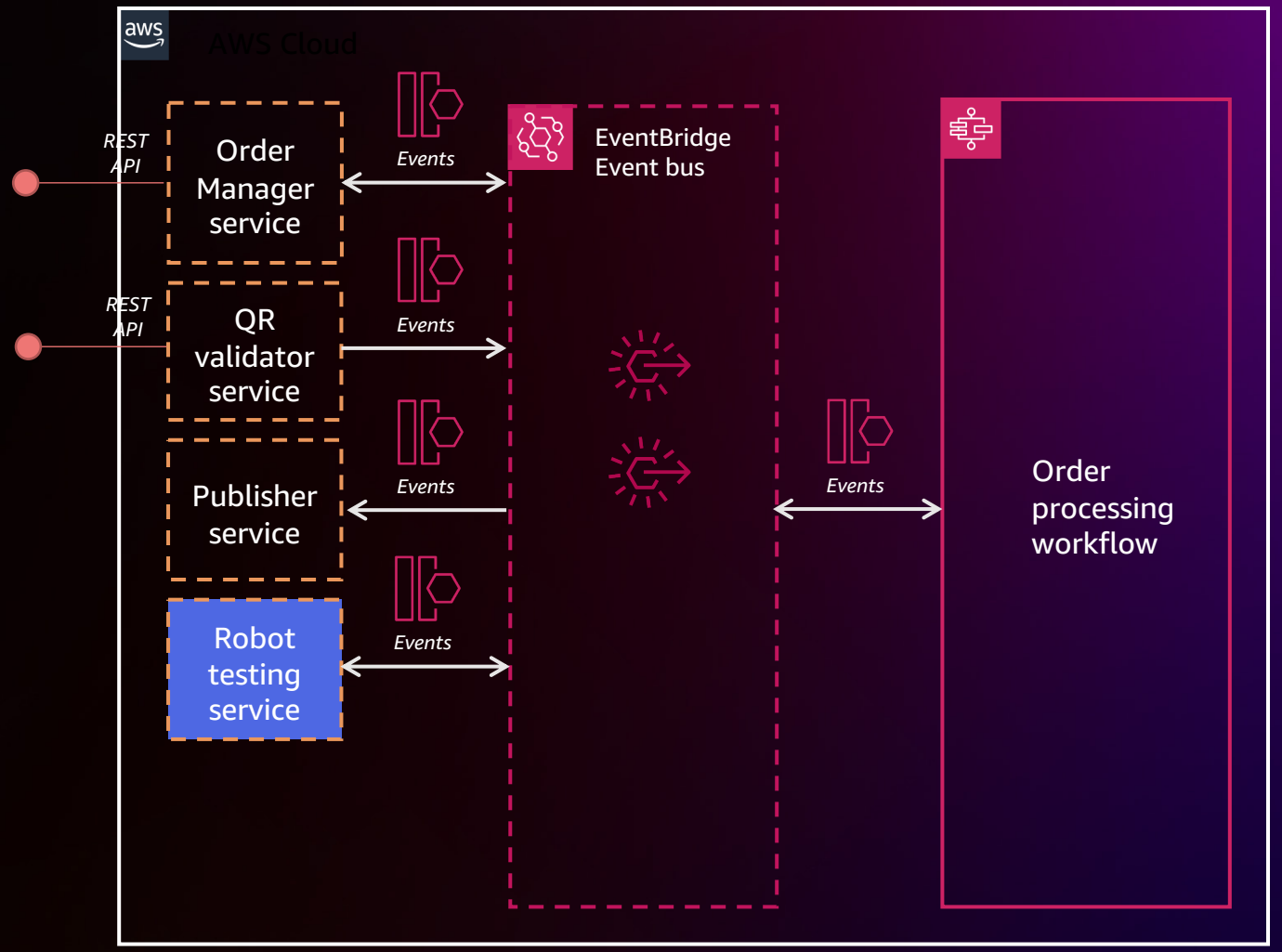
- Events will change in development. Use versioning to implement contracts.
- Use EventBridge Schema Registry to keep track of all your events
- Use open-source tools like EventBridge Atlas to visualize and document flows
- Start documenting events from the beginning



EventBridge Atlas: https://eventbridge-atlas.netlify.app/

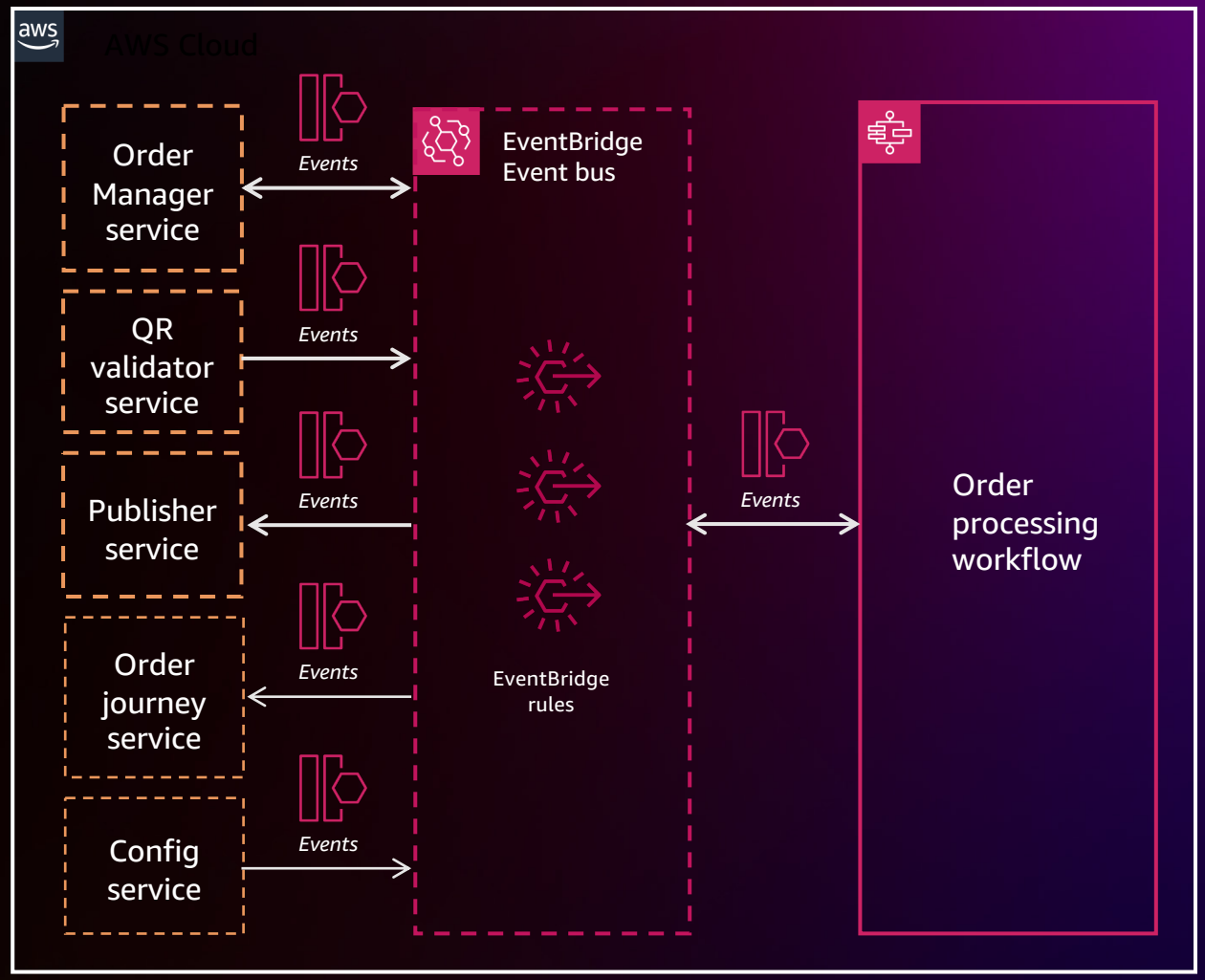
Testing by event injection

- Instead of calling APIs, put events on the bus
- Our Robot Tester is another microservice – it generates and completes orders at scale
- Use archive/replay to test services in dev with real data
- Create one rule to send all events to CloudWatch Logs (in dev/test). Tail the logs with the CLI.



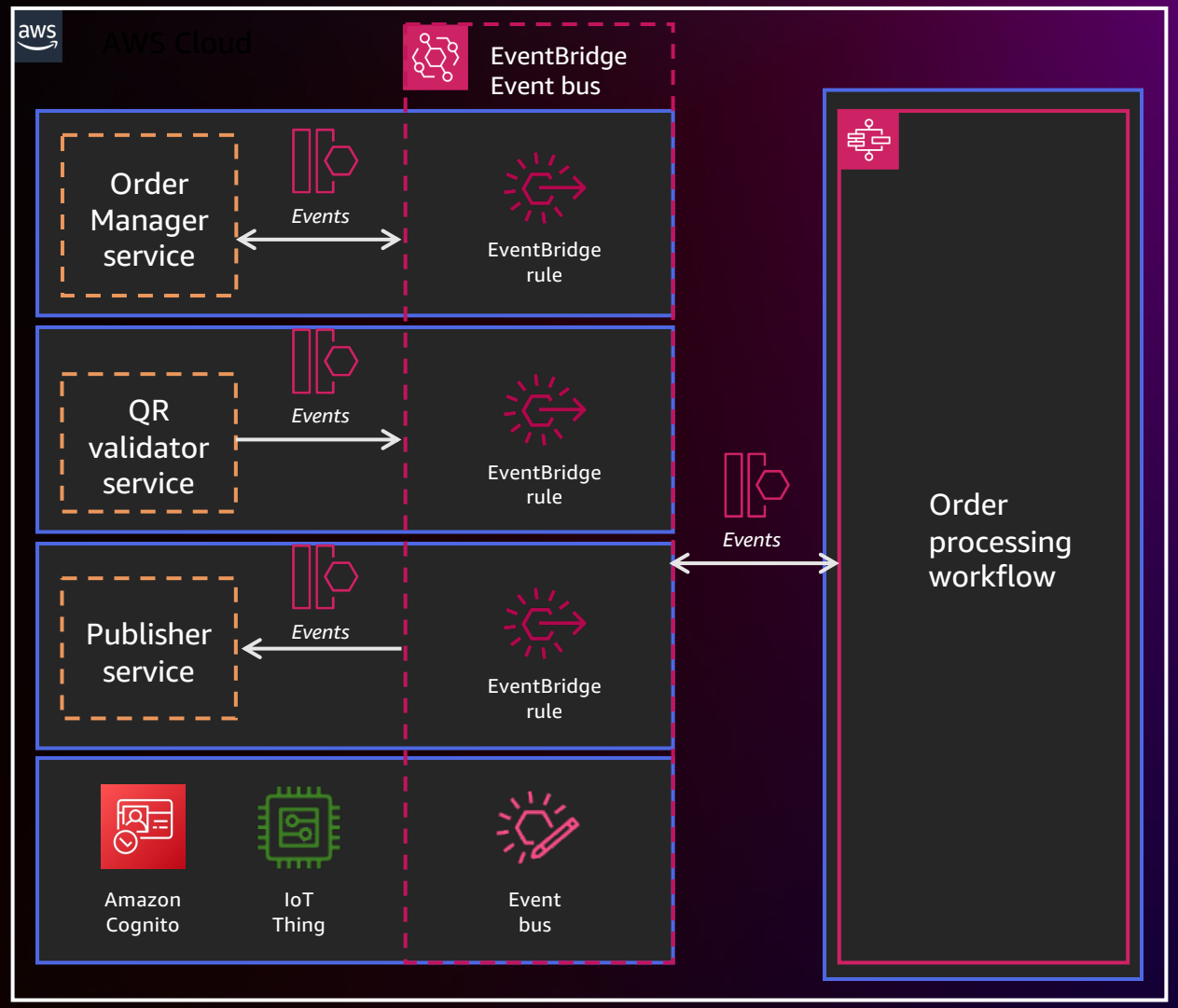
Communicating between microservices

- Event flow drives the application
- Events choreograph the services, while Step Functions orchestrates the transactions
- Replace private APIs with event where possible
- Add new microservices without changing existing code
- Microservices emit events independently of consumers



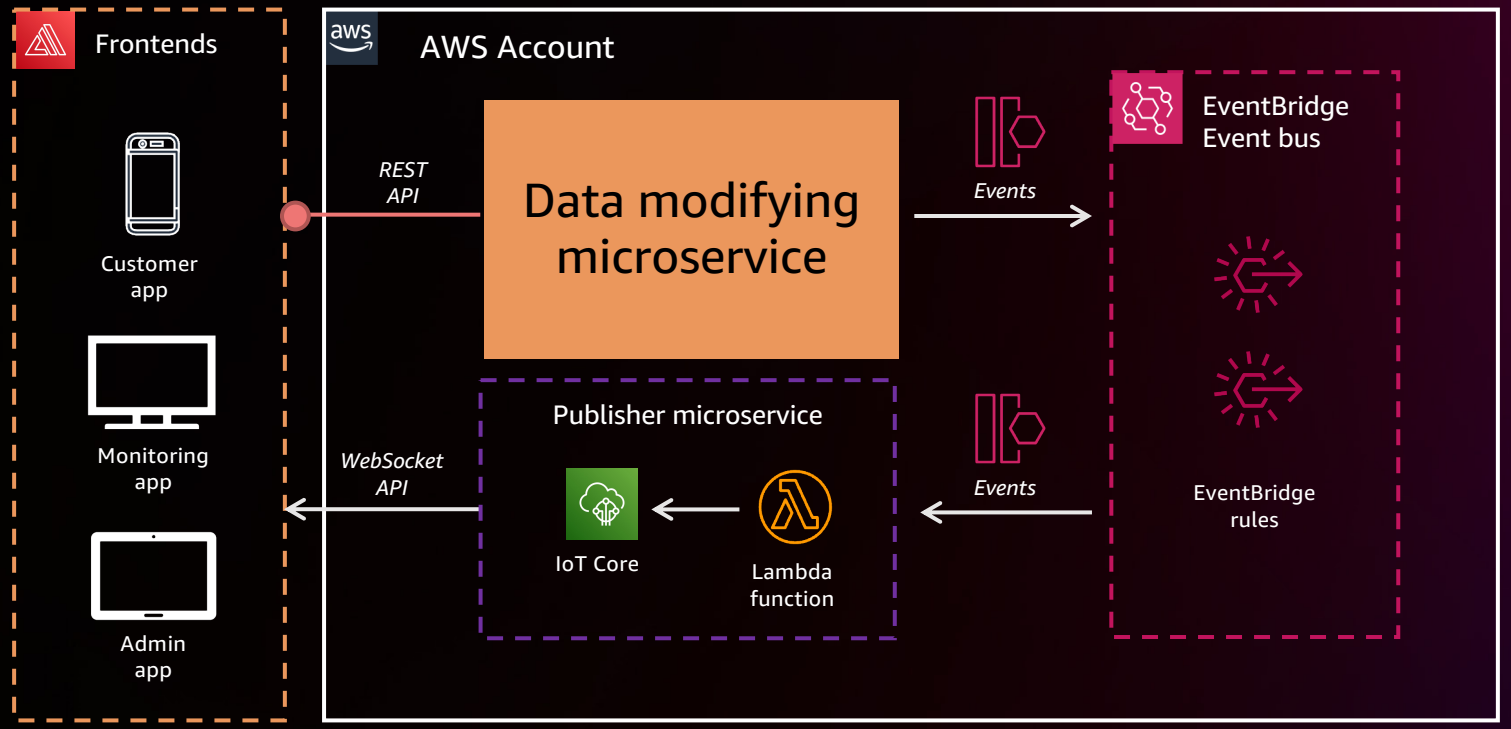
Breaking up monolithic deployments

- Originally one large AWS SAM template
- Use a core template for common resources
- Each microservice is an AWS SAM template including a rule
- Each workflow is an AWS SAM template
- Faster and safer to deploy

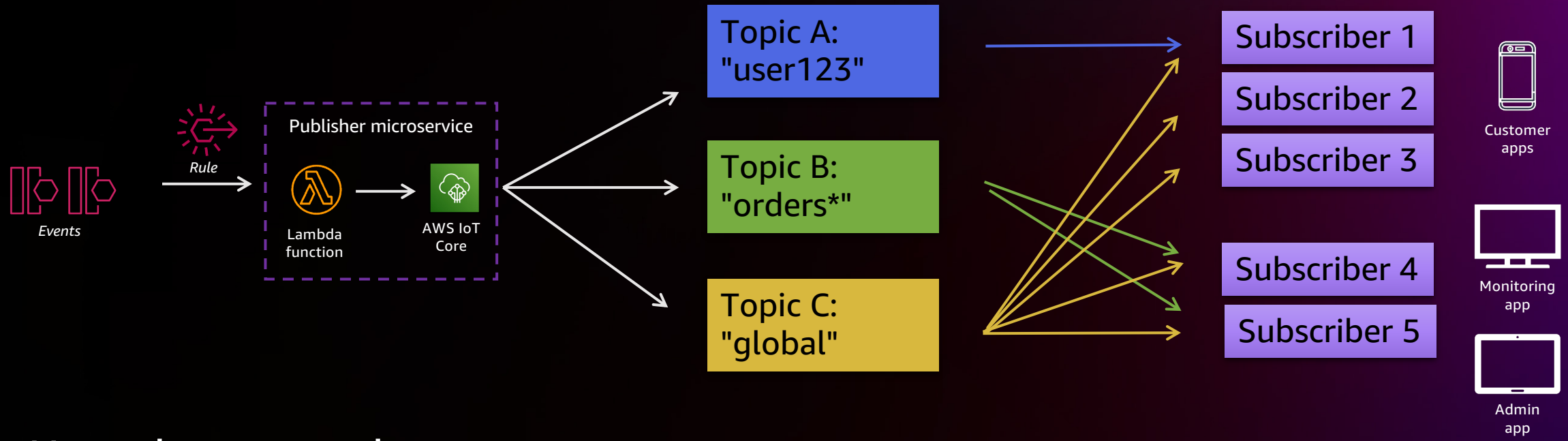


Useful patterns

Command Query Responsibility Separation (CQRS) ... with WebSockets



CQRS with WebSockets



- Use rules to map between event attributes and topics
- Use content filtering rules to match multiple values (e.g., wildcards)

Orchestration and choreography

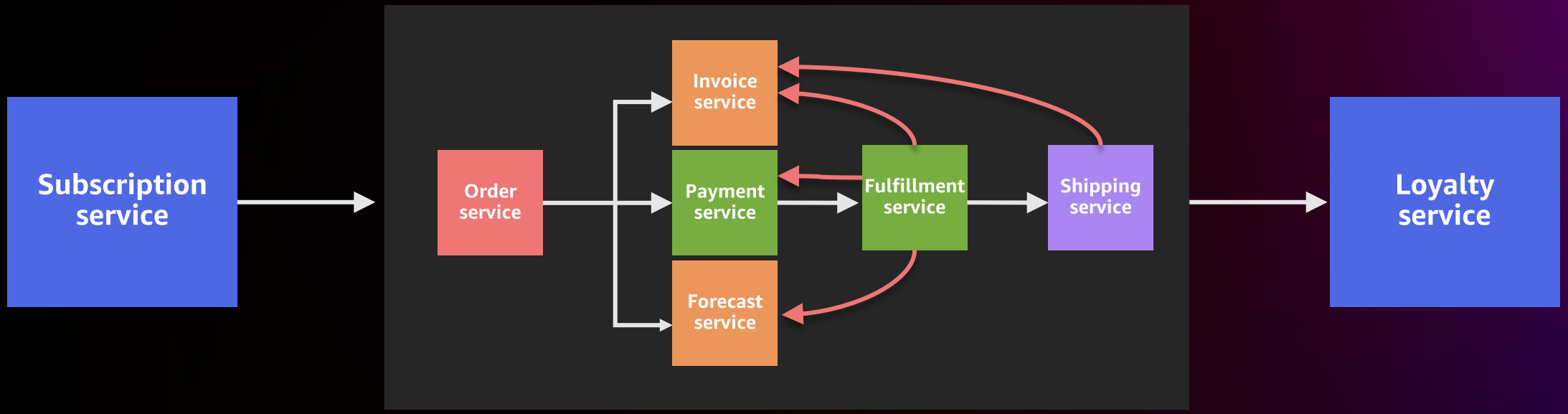
Orchestration with AWS Step Functions

- One system controls the flow between components
- Easier end-to-end monitoring
- Robust timeouts, retries, error handling
- Centralized business logic

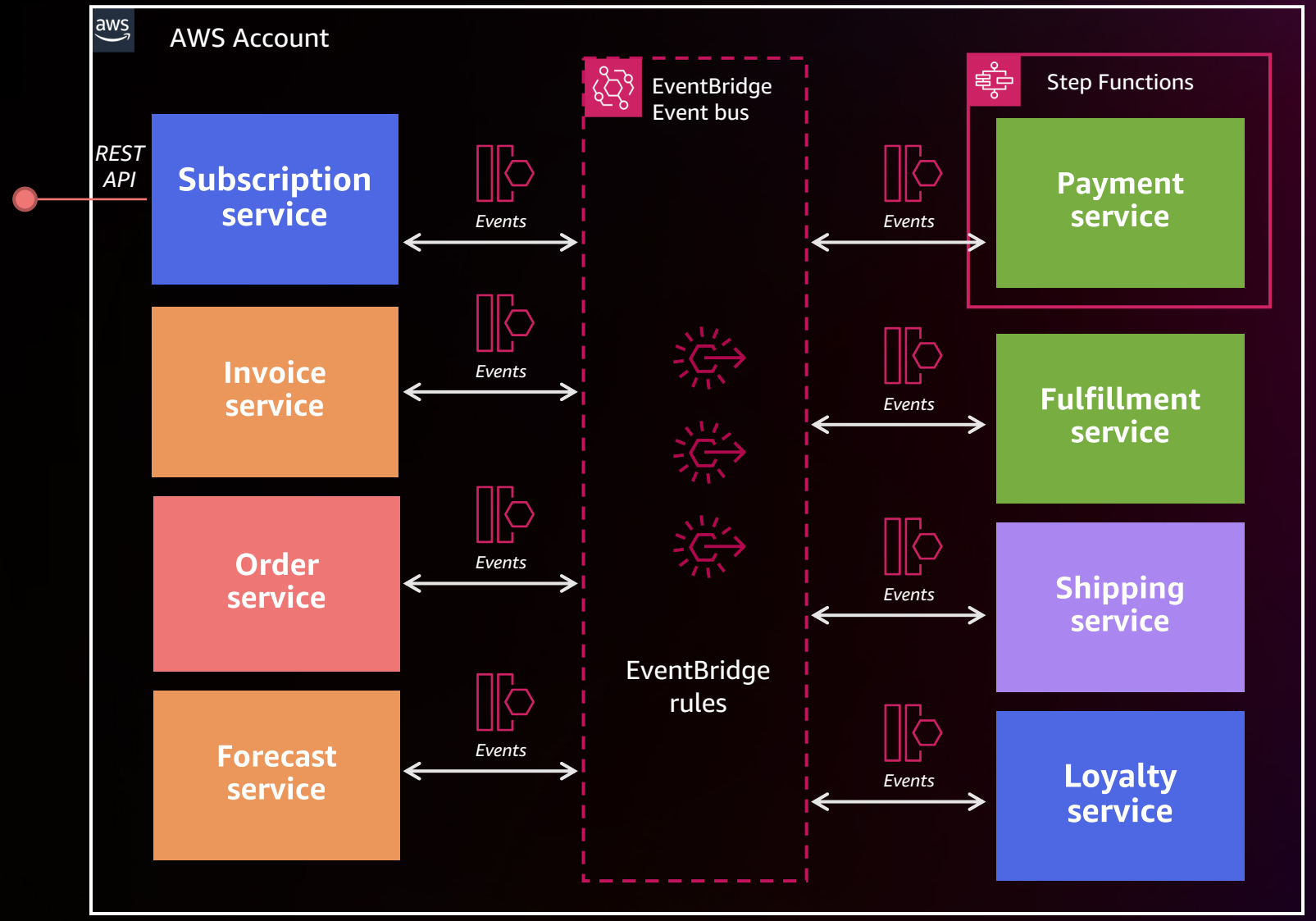
Choreography with Amazon EventBridge

- Pass messages between bounded context of services
- The flow is an emergent property of events being sent
- Easier to extend, modify, and build upon the messages being passed

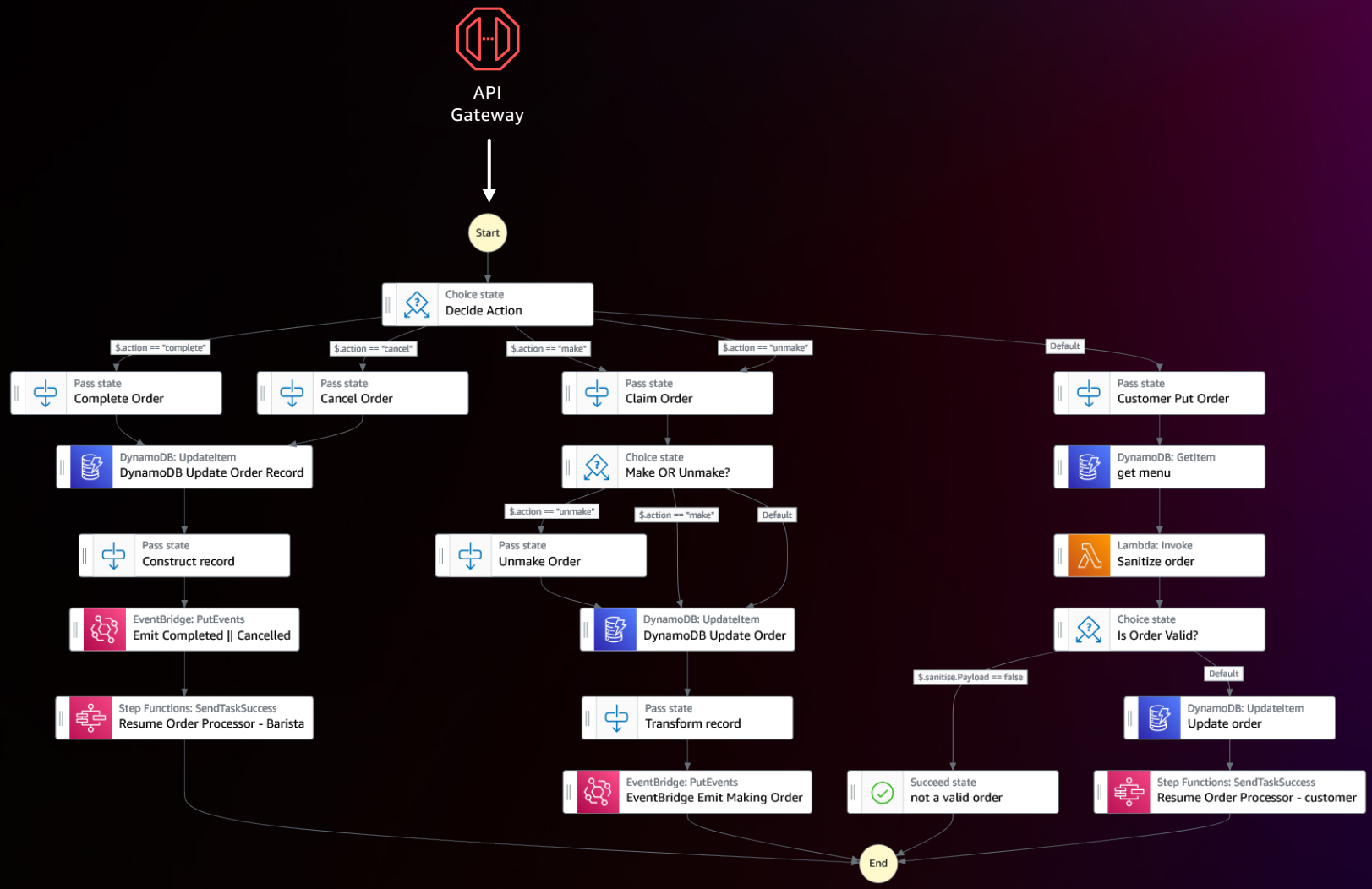
Moving from API driven...



… to event-driven architecture



The Step Functions CRUD workflow



The Step Functions CRUD workflow

How?

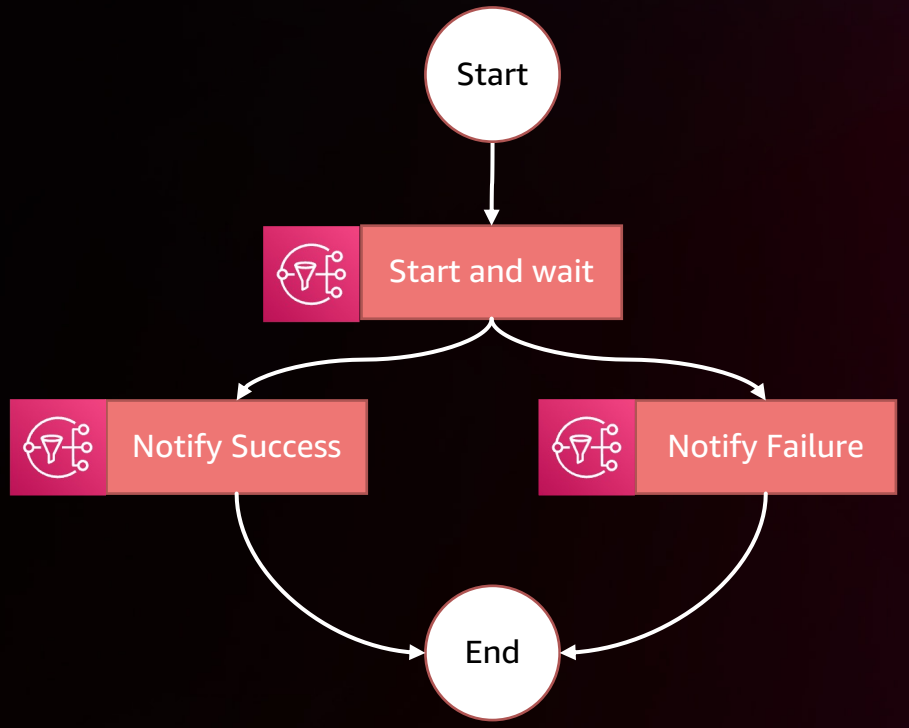
- One API Gateway endpoint starts the workflow
- First transition routes CRUD action
- Uses service integrations wherever possible to reduce compute
- Uses Lambda for custom logic
- Can also use Express workflows if return value needed

Benefits

- Reduced complexity (single ASL file)
- Easier monitoring and debugging
- Simpler to implement retries and error handling
- May be more cost effective
- May reduce latency
- Allows versioning of workflows

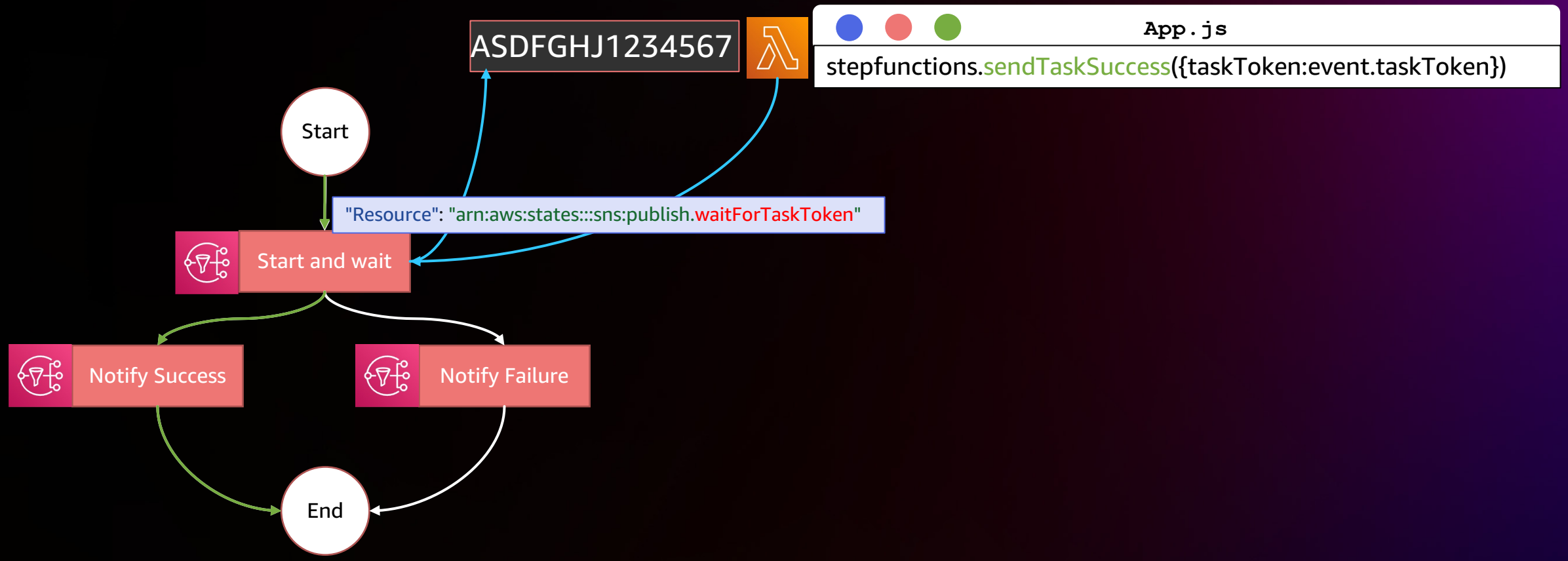
Using workflows to manage waiting

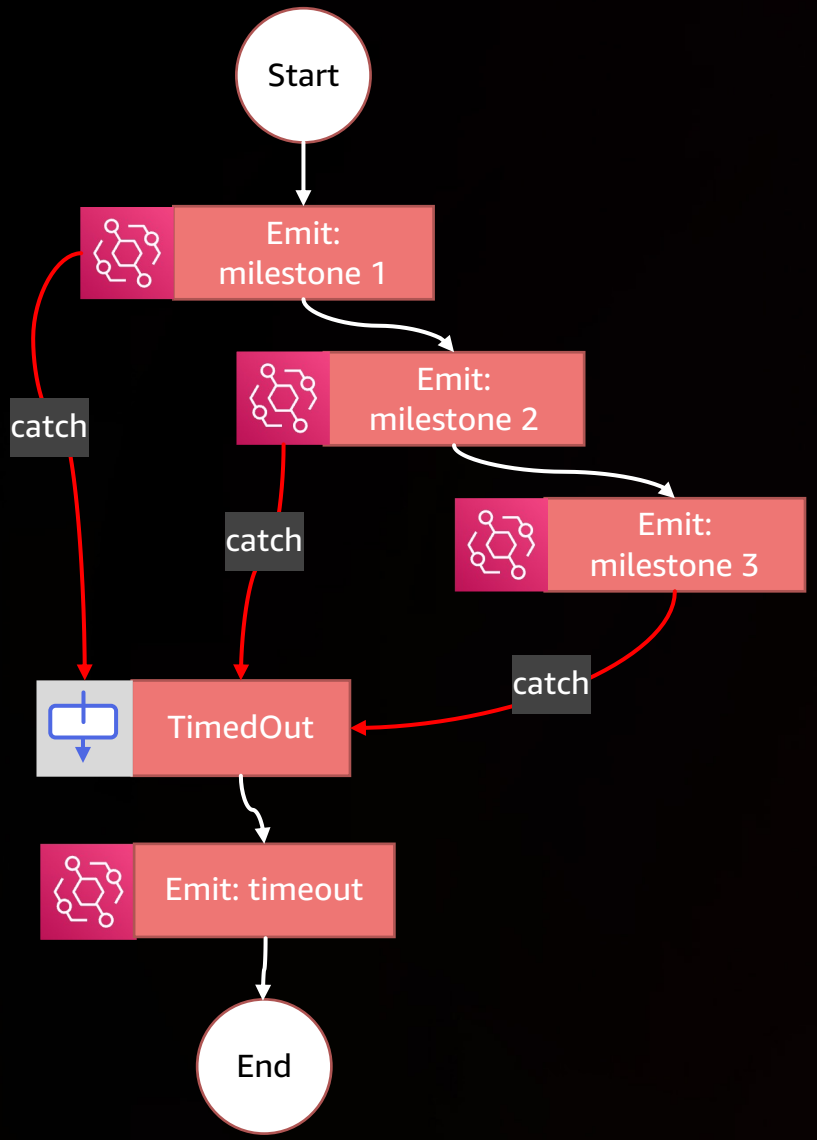
PAUSE A WORKFLOW FOR UP TO 1 YEAR UNTIL A TASK TOKEN IS RETURNED



Using workflows to manage waiting

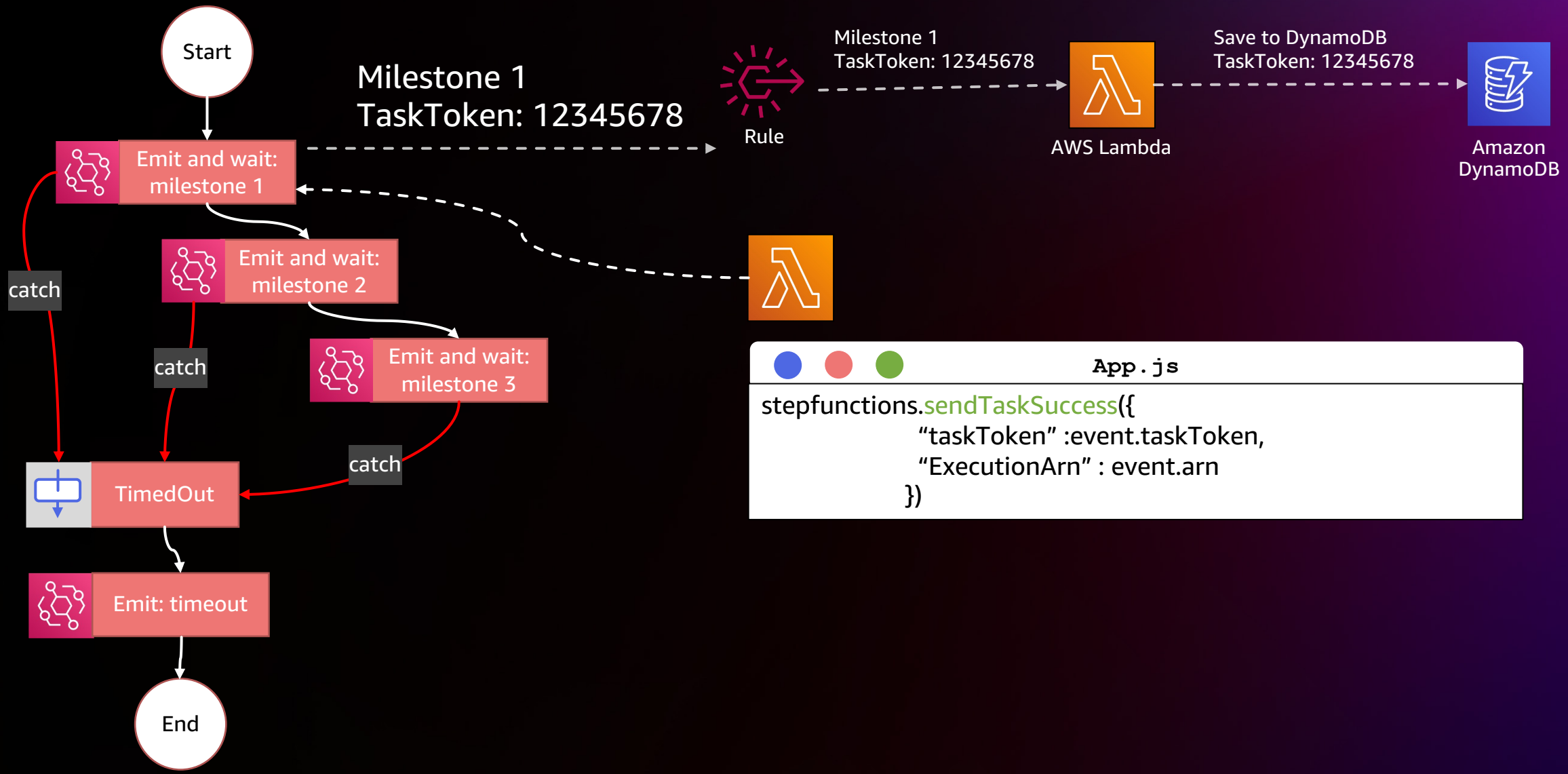
PAUSE A WORKFLOW FOR UP TO 1 YEAR UNTIL A TASK TOKEN IS RETURNED

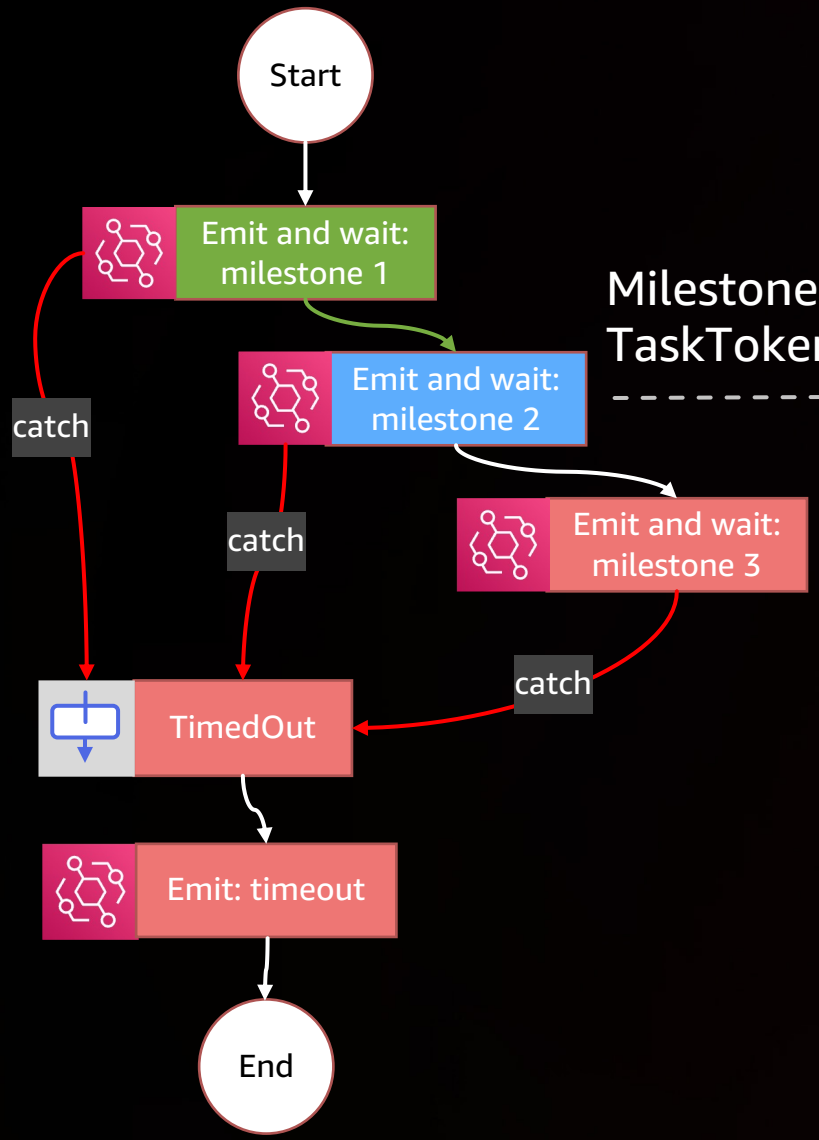




The “emit and wait” pattern

Emit milestone events that invoke external decoupled microservices. Emit error events if they don’t respond in time





Milestone 2
TaskToken: 12345678

Rule

Milestone 2
TaskToken: 12345678

AWS Lambda

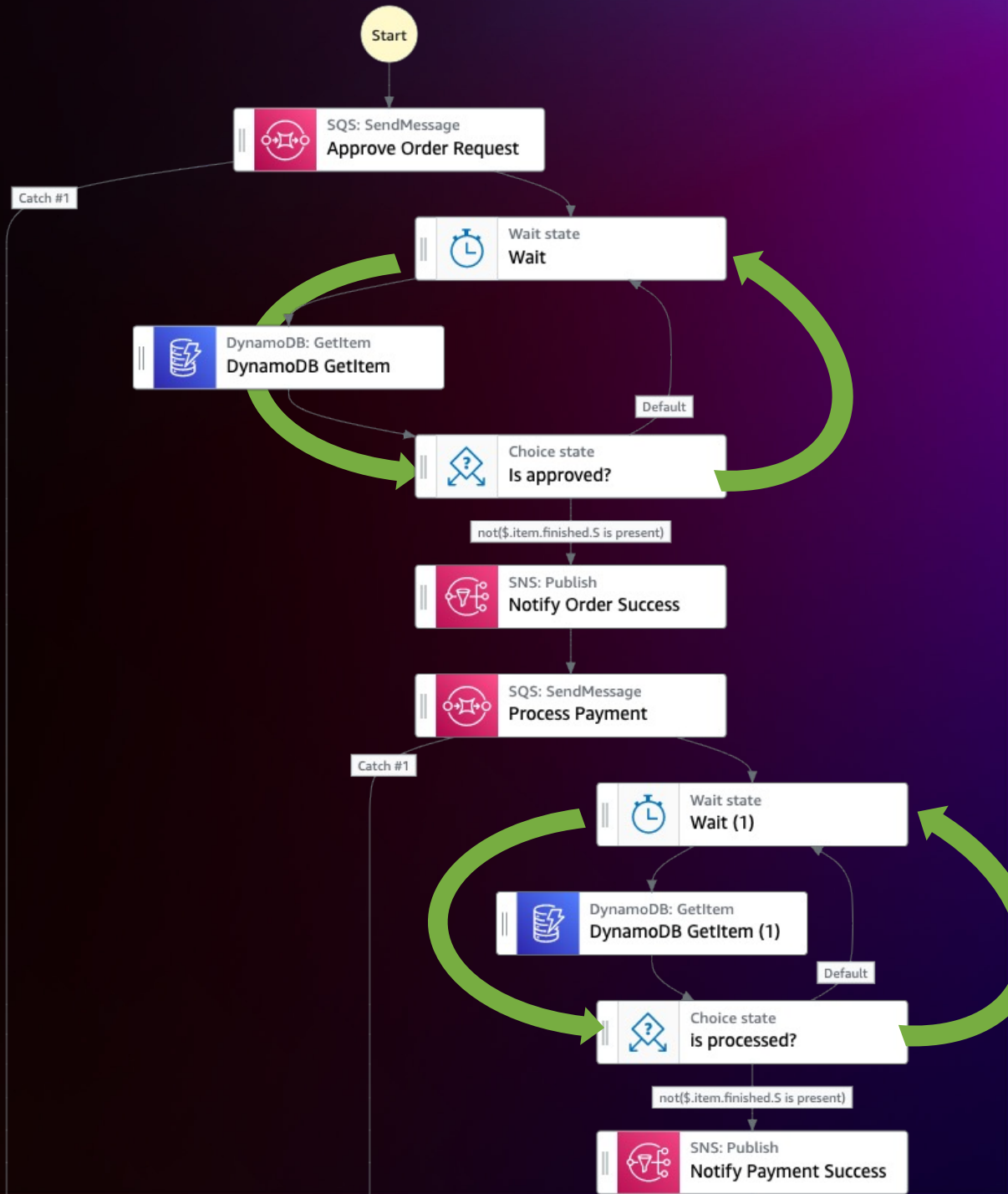
Save to DynamoDB
TaskToken: 12345678

Amazon DynamoDB

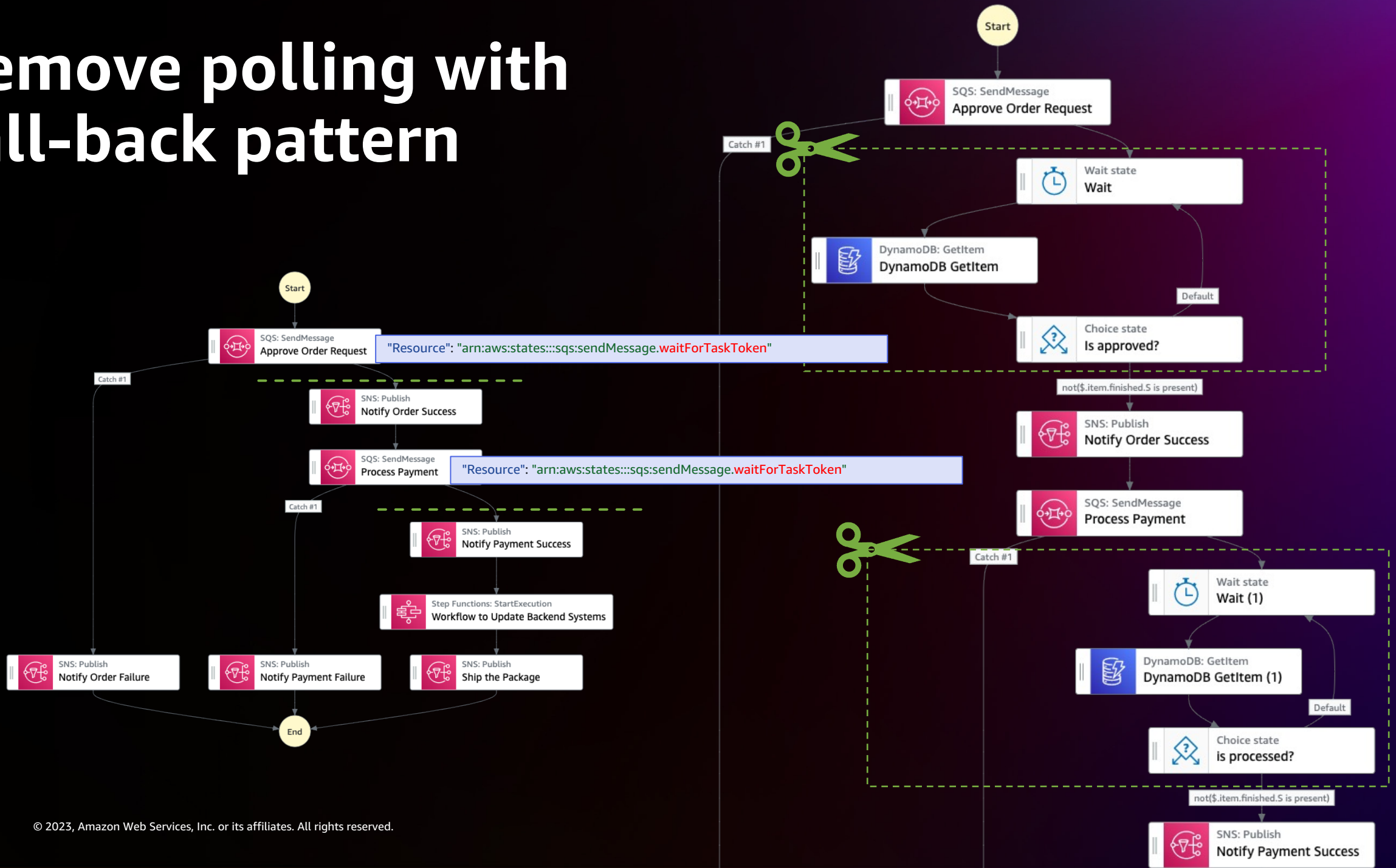
Set a time limit to wait for task token

"HeartbeatSeconds": 20

Remove polling with call-back pattern



Remove polling with call-back pattern



What does it cost to run this workload?

We can serve up to 960 drinks to 960 customers every day

Service	Daily cost	With Free Tier
AWS Amplify console	$0.28	Free
Amazon API Gateway	$0.01	Free
Amazon Cognito	$0.00	Free
Amazon DynamoDB	$0.01	Free
Amazon EventBridge	$0.01	Free
AWS IoT Core	$0.01	Free
AWS Lambda	$0.01	Free
Amazon SNS (SMS messaging)	$7.98	$7.98
AWS Step Functions	$0.29	Free

serverlesspresso

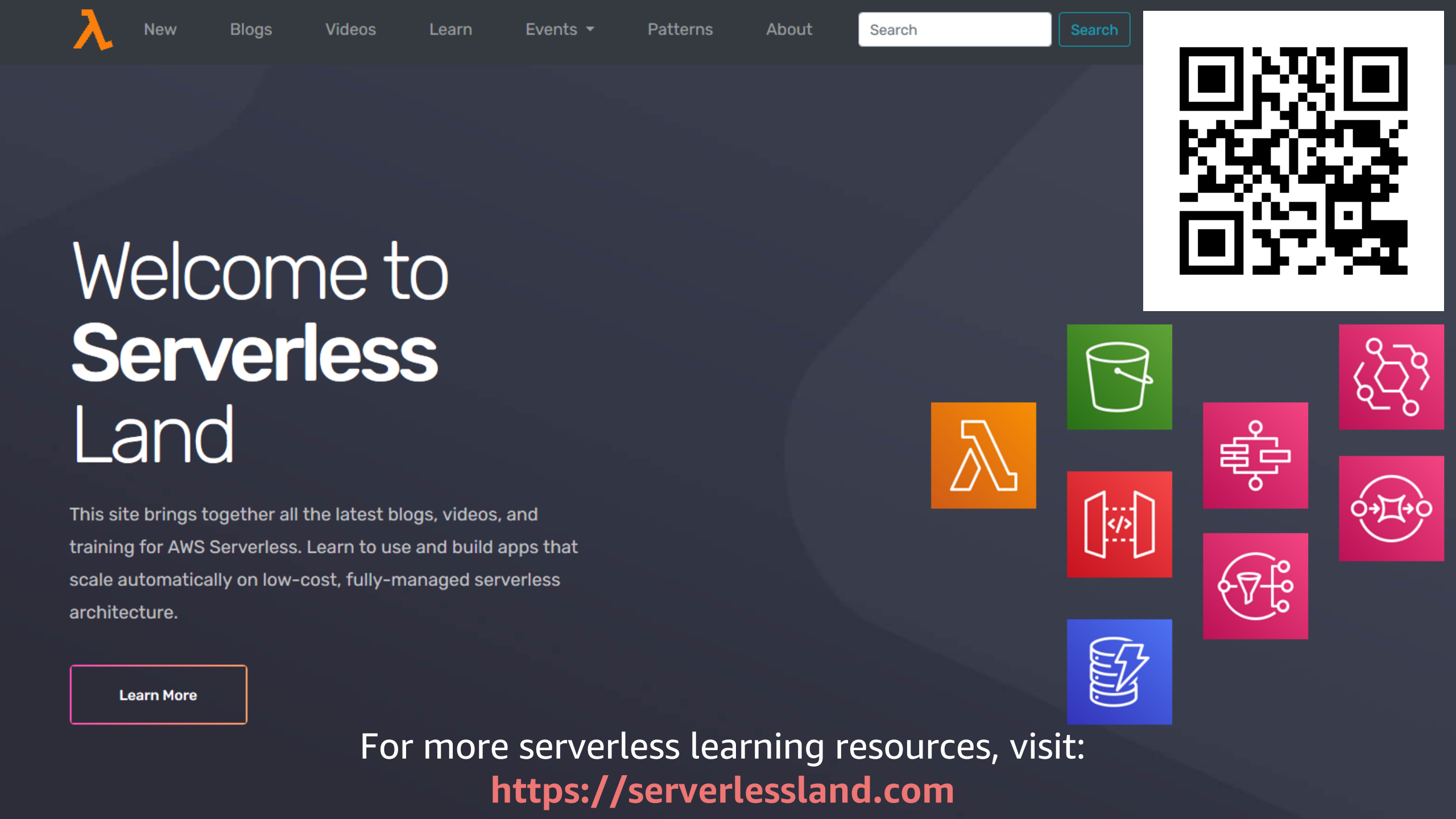
Learn about the AWS Free Tier:
https://aws.amazon.com/free

aws

Summary

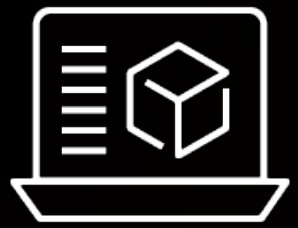
- Design: start with the workflow; attach microservices; add frontends
- Microservices: communicate with events; use APIs if outward facing
- Real-time frontends: using AWS IoT Core for serverless WebSockets
- Use Step Functions to orchestrate within microservice boundaries. Use EventBridge to communicate across those boundaries.
- Combining orchestration with choreography can create highly extensible, low-code, cost-effective workloads

Resources from this talk at
https://s12d.com/svs312


New
Blogs
Videos
Learn
Events
Patterns
About
Search
Search
Welcome to
Serverless
Land
This site brings together all the latest blogs, videos, and training for AWS Serverless. Learn to use and build apps that scale automatically on low-cost, fully-managed serverless architecture.
Learn More
For more serverless learning resources, visit:
https://serverlessland.com

Continue your AWS Serverless learning

Learn at your own pace



Expand your serverless skills with our Learning Plan on **AWS Skill Builder**

Increase your knowledge

Use our **Ramp-Up Guides** to build your serverless knowledge

Earn AWS Serverless badge

Demonstrate your knowledge by achieving **digital badges**

https://s12d.com/serverless-learning

Thank you!

Bianca Mota
biamota@amazon.com
linkedin.com/in/bianca-smota/