Fullstack generative AI sample app for Amazon Bedrock

by Michael Liendo | on 27 OCT 2023 | in Amazon Bedrock, Artificial Intelligence, AWS AppSync, Front-End Web & Mobile, Generative AI, Serverless, Technical How-To | Permalink | Share

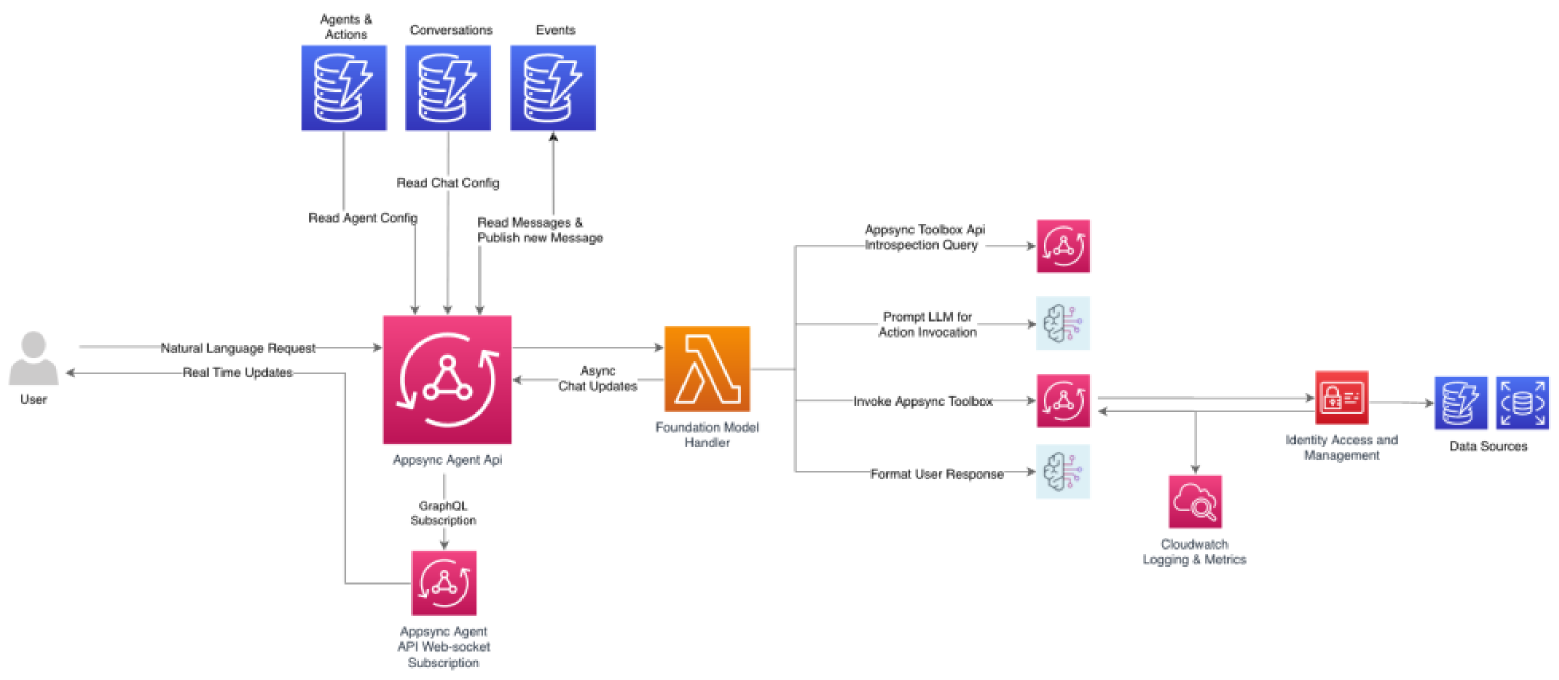
> This post was written by Eric Robertson | AWS SDE

Introduction

In this post we show you how to quickly deploy a fully functioning Generative AI Sample App to explore the power of generative AI when enabled and augmented by Amazon Bedrock and AWS AppSync.

For more details on what the sample app does, and the unique problems it solves for developers building generative AI-powered applications, read these companion blog posts: Create an async API for Amazon Bedrock with AWS AppSync and WebSockets, and Connect Amazon Bedrock to enterprise data with AWS AppSync and GraphQL.

The high level system diagram described below is what powers this sample app. In this blog post we'll walk you through how you can easily set up this sample app yourself, leveraging the provided AWS Cloud Development Kit ("CDK") construct.



Prerequisites

To follow this tutorial we will assume you have the following configured

- Node.js v18 or higher
- Yarn
- CDK installed and bootstrapped
- Git
- Docker

- An AWS Account
- Either Amazon Bedrock access or an OpenAI Api key

Setup

Cloning the repository

We need to first get the example code pulled onto our computer, you can go here (provide link here) to see the code needed for this tutorial.

Download it with:

`git clone <some url here>`

You should have the project on your computer now with the following folders:

```
cdk-infrastructure
handler-claude-agent
handler-claude-simple
handler-mirror
handler-openai-streaming
playground
```

Open the project root with your IDE of choice. For this demo we will use VS-Code.

Deploying the infrastructure

To deploy the infrastructure to us-west-1 (the above system diagram), run the following command. If you want to use a different region, edit the region value in the script. This will assume your local env has the correct AWS configurations and permissions to create resources in this region.

`./cdk-deploy.sh`

This command will do the following in order:

- Install the dependencies for the cdk package
- Compile all the .ts appsync resolver source files into .js files
- Deploy the cdk infrastructure (this make take a few minutes on the first deployment as it includes docker container builds)
- Uploads example data to the generated DynamoDB tables
- Downloads the API endpoints and api keys and Cognito pool ids into the local react environment
- Installs the dependencies for the website itself

Once this is complete, your entire environment should be ready and you can begin exploring your Gen AI Agent sample app.

With Amazon Bedrock access, you will implicitly be able to access Anthropic's Claude model through AWS. If you wish to use OpenAI, you can add a .env file in handler-openai-streaming like so with the key being the key you get from OpenAI.

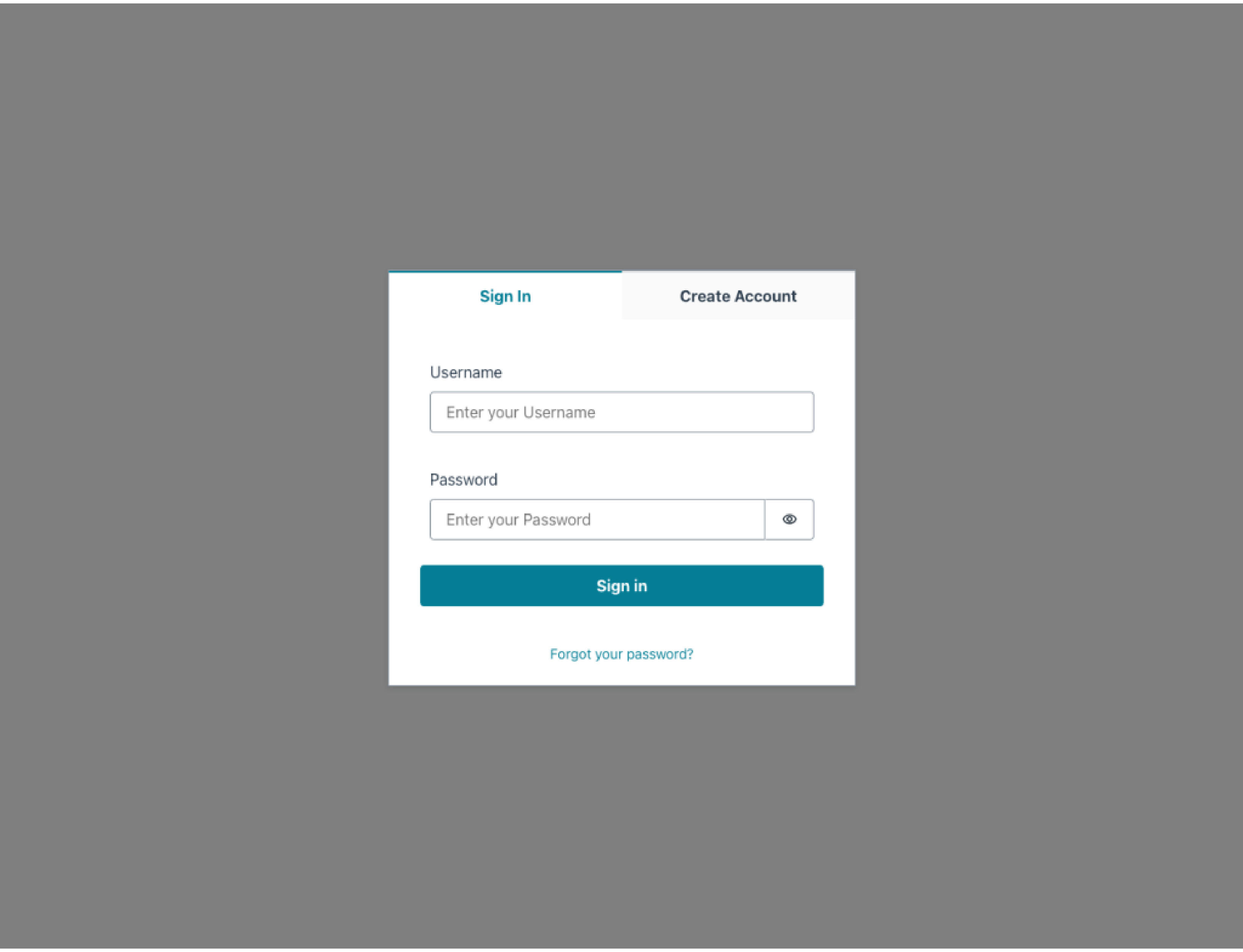
`OPENAI_SECRET_KEY=sk-XXXXXXXXXXXXXXXXX`

Using the Sample App Local Playground

Run the following commands to launch the Sample App locally.

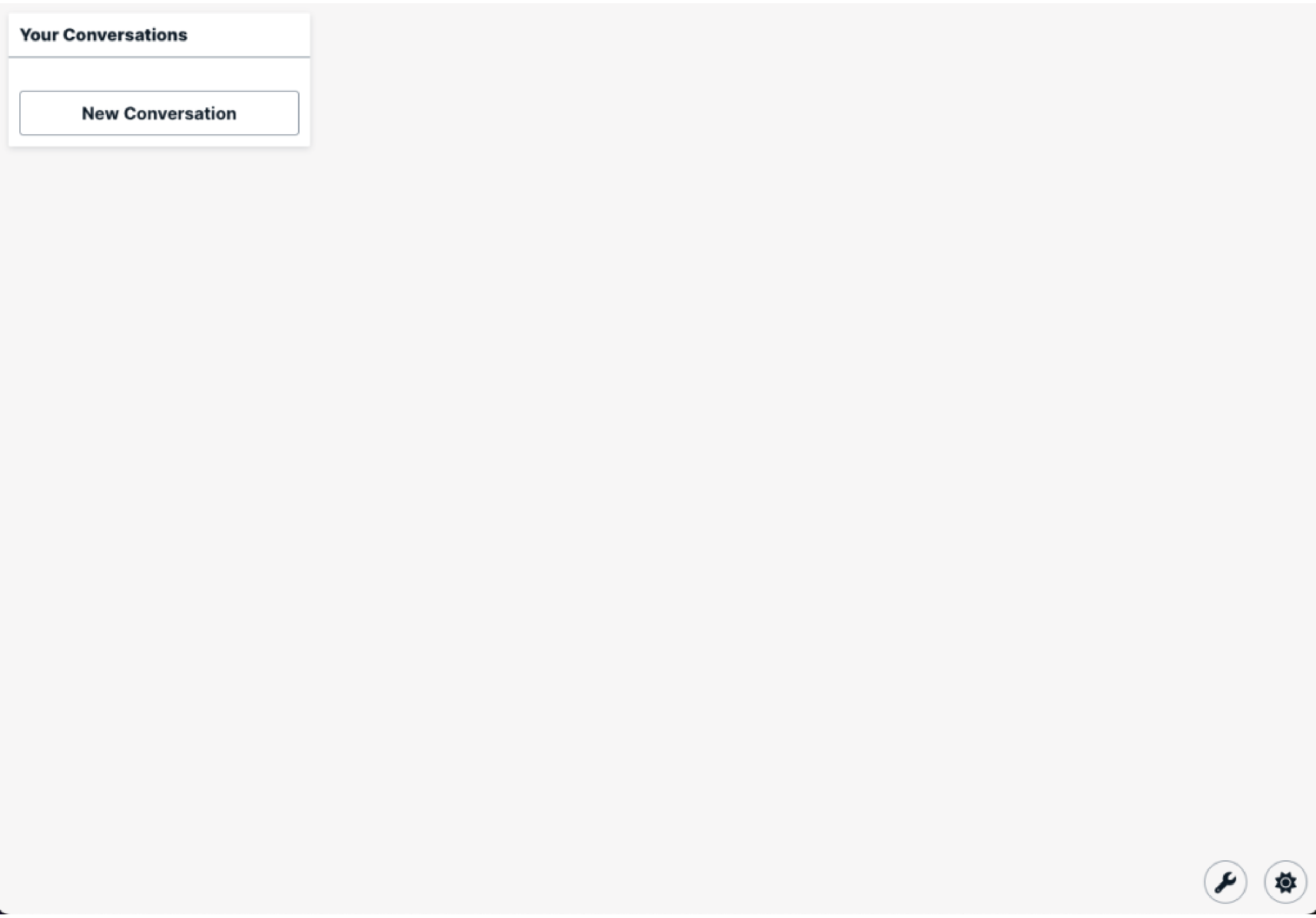
```
cd playground

npm start
```

This will start a local react server on localhost:3000 and you should see a page like this:



The sample app experience manages access through an AWS Cognito User Pool, which was deployed to your AWS account as part of the CDK deployment. Creating an account will add your user to the user pool in your own AWS account, automatically send you a confirmation email, and let you sign in. Credentials are stored as cookies in localhost. No data is stored outside your browser and your personal AWS account.

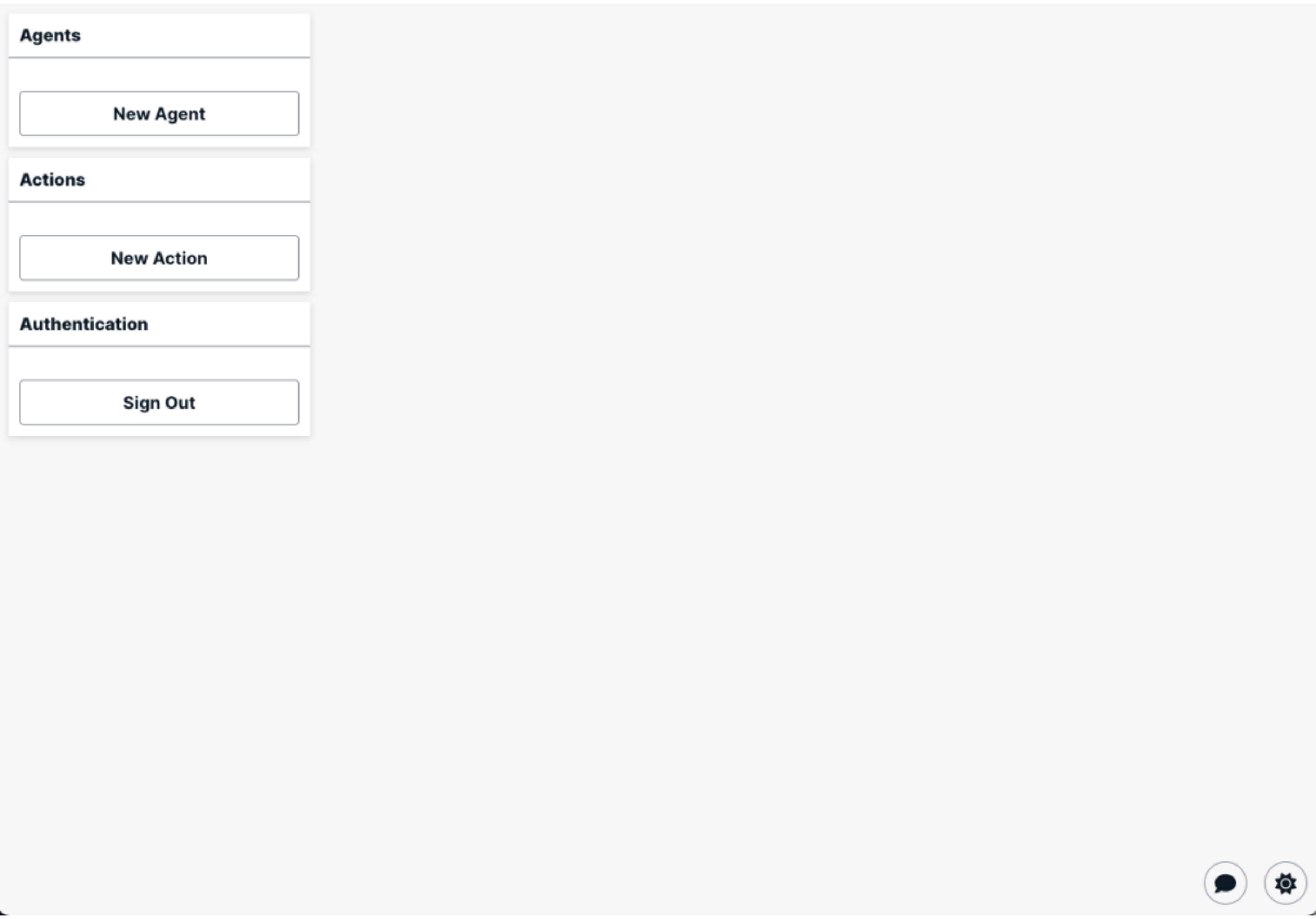
If you successfully sign-in, you will be presented with a page like this which shows there are no current conversations yet.



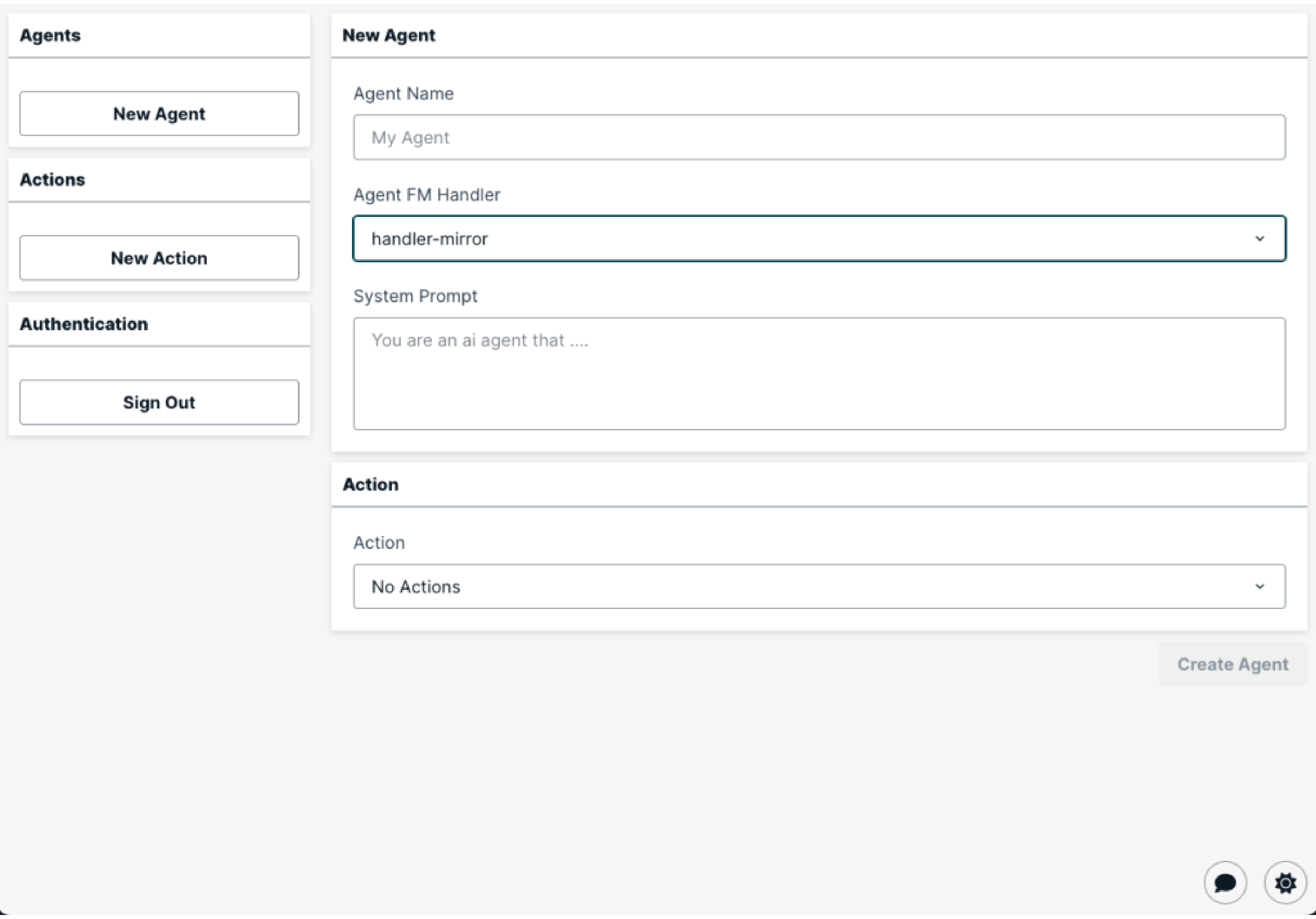
Setting up Agents

Before we can start chatting, we need to setup an agent to chat with. The demo repo has deployed a collection of Lambda LLM handers which you can configure into an agent to chat with.

Clicking on the wrench icon in the bottom right switches to the configuration mode for the sample app, and you should see this page.



Clicking on “New Agent” allows you to setup a new agent to chat with. Here you can enter a name, choose a handler (or enter a custom handler ARN) and set a system prompt for the agent. The system prompt will be sent to the agent as context and not be shown to the chat user. We will not set an action for now.


Agents

New Agent

Actions

New Action

Authentication

Sign Out

New Agent

Agent Name: My Agent

Agent FM Handler: handler-mirror

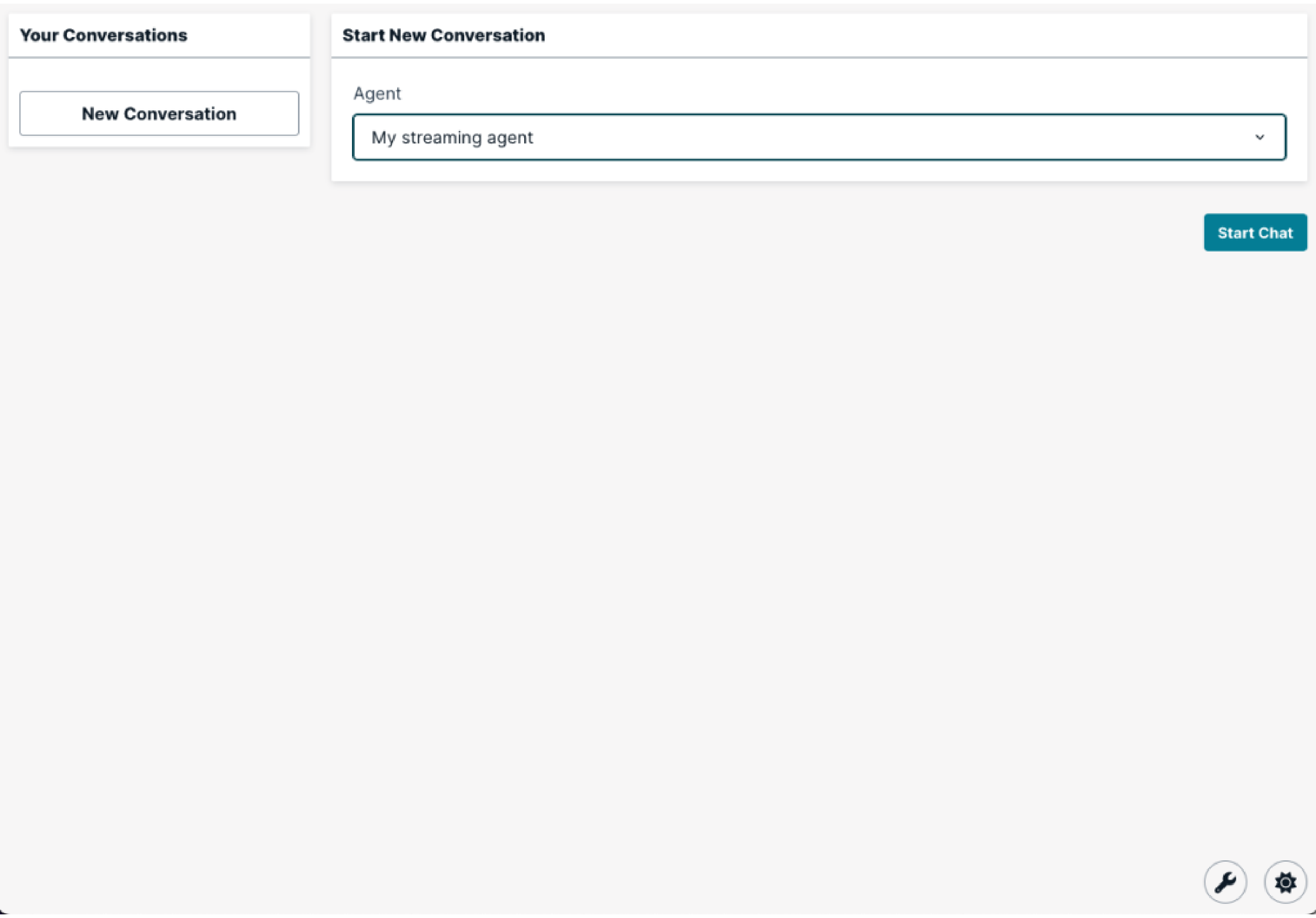
System Prompt: You are an ai agent that ....

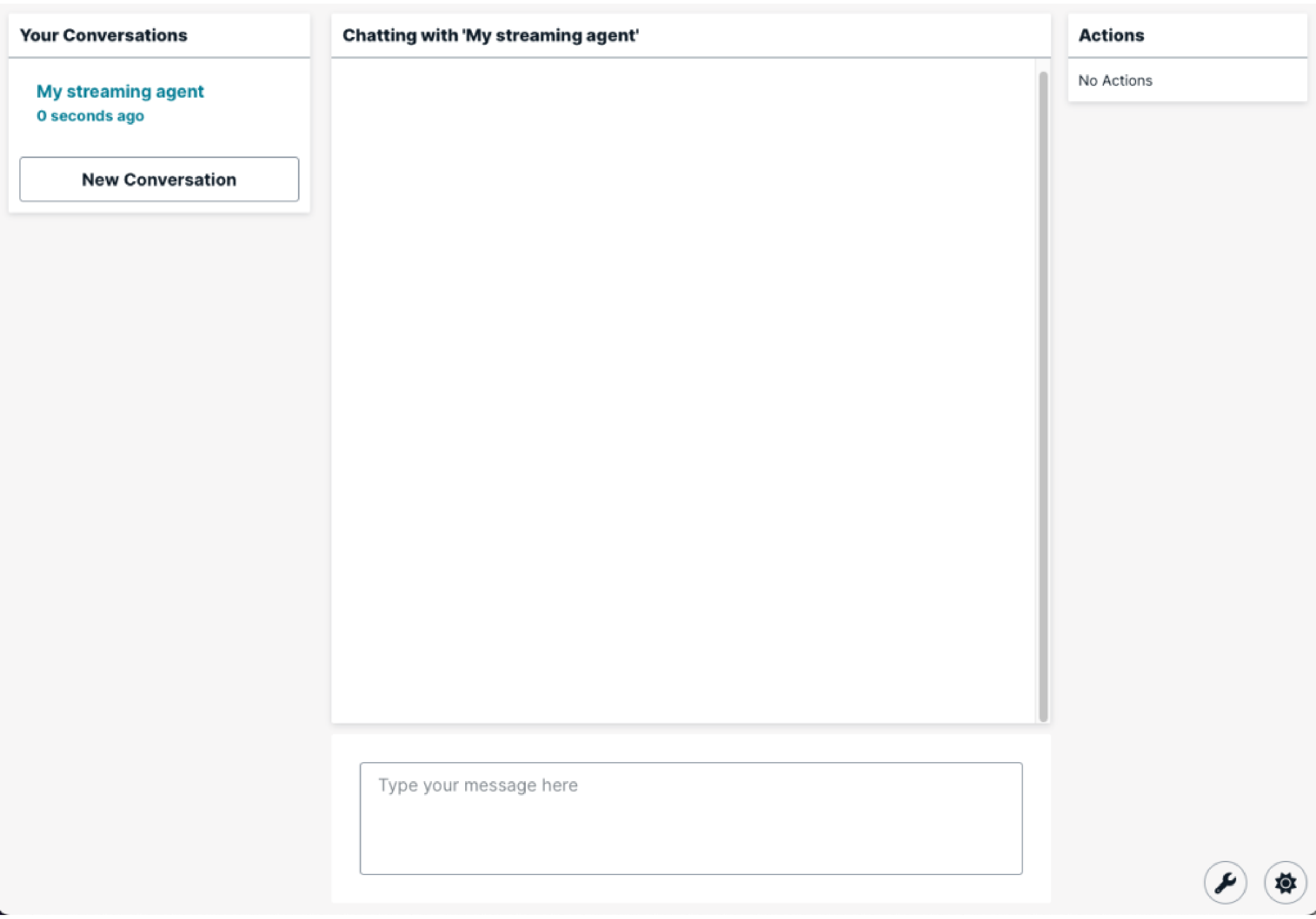
Action

Action: No Actions

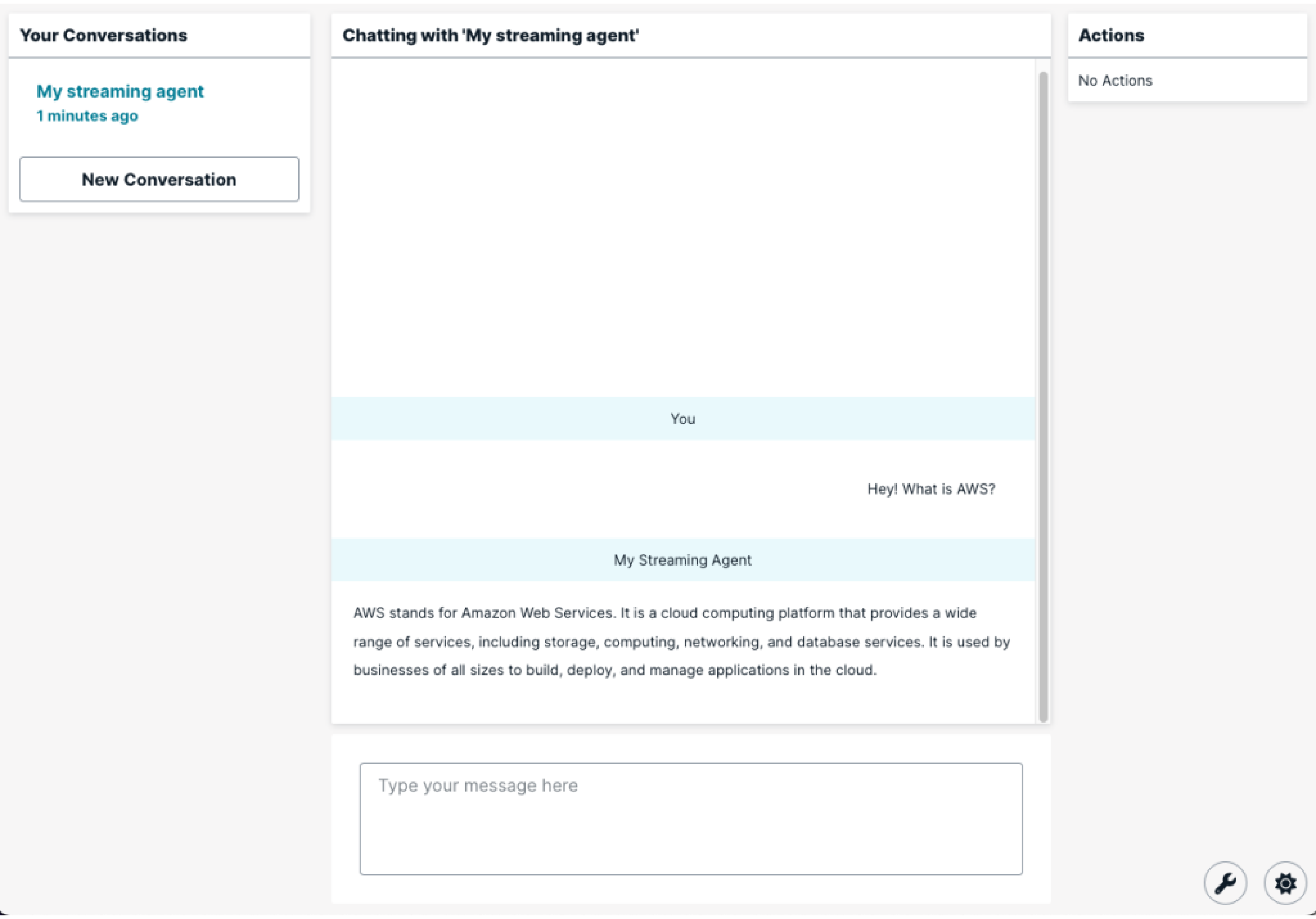
Create Agent

In this example I will create a streaming based handler, give it a name, and create the agent. Navigating back to the conversations page, we can now start a chat with our agent.


Your Conversations
New Conversation
Start New Conversation
Agent
My streaming agent
Start Chat



If we want, we can now converse directly with the agent we built.



You

Hey! What is AWS?

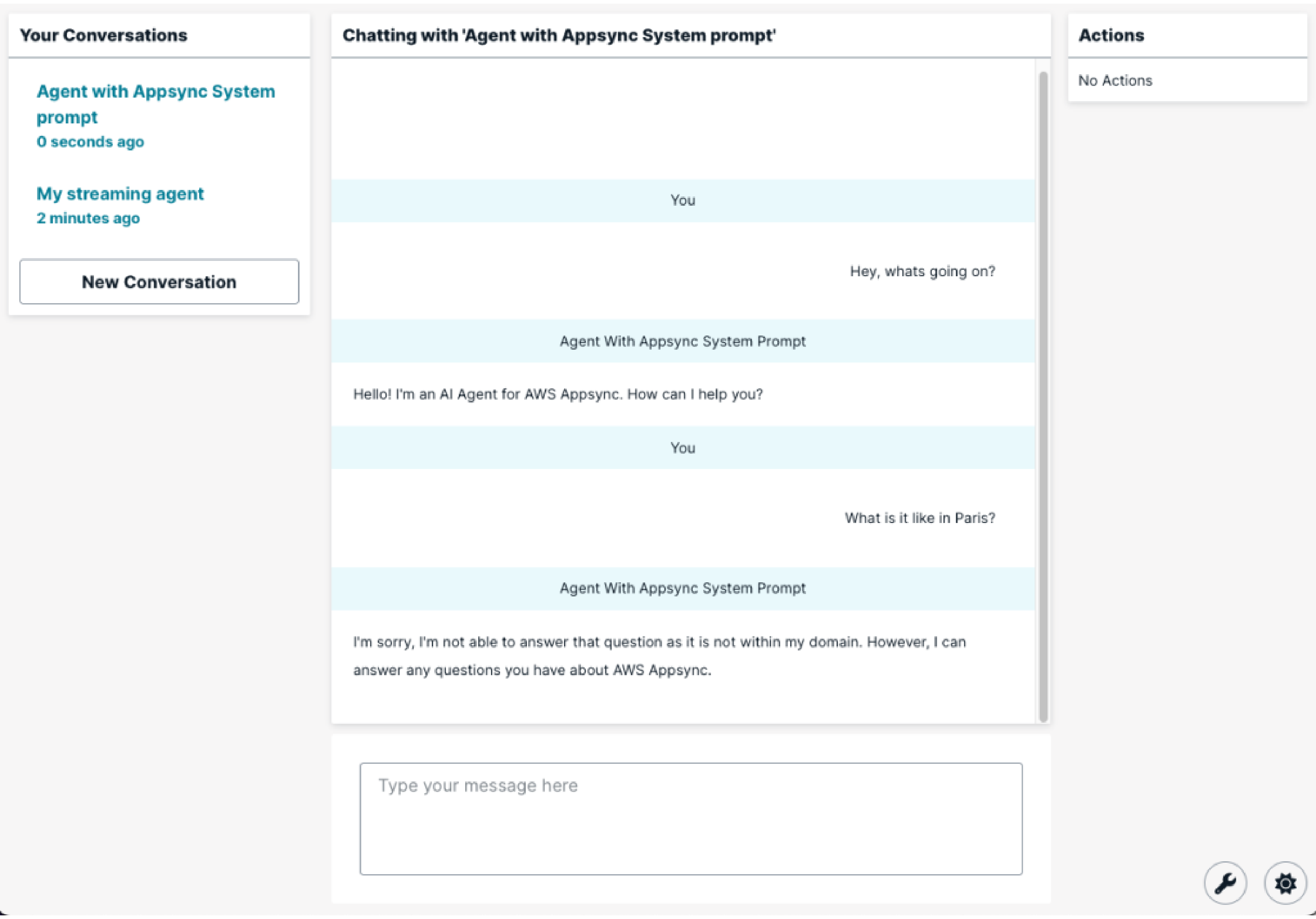
My Streaming Agent

AWS stands for Amazon Web Services. It is a cloud computing platform that provides a wide range of services, including storage, computing, networking, and database services. It is used by businesses of all sizes to build, deploy, and manage applications in the cloud.

Or if, we configure a system prompt under a new agent like so:

```
You are an AI Agent for AWS Appsync.
Keep all conversations on-brand and refuse to answer anything that is not within this
```

Then we can build an agent which responds to messages as we want.


Your Conversations

Agent with Appsync System prompt
0 seconds ago

My streaming agent
2 minutes ago

New Conversation

Chatting with 'Agent with Appsync System prompt'

You

Hey, whats going on?

Agent With Appsync System Prompt

Hello! I'm an AI Agent for AWS Appsync. How can I help you?

You

What is it like in Paris?

Agent With Appsync System Prompt

I'm sorry, I'm not able to answer that question as it is not within my domain. However, I can answer any questions you have about AWS Appsync.

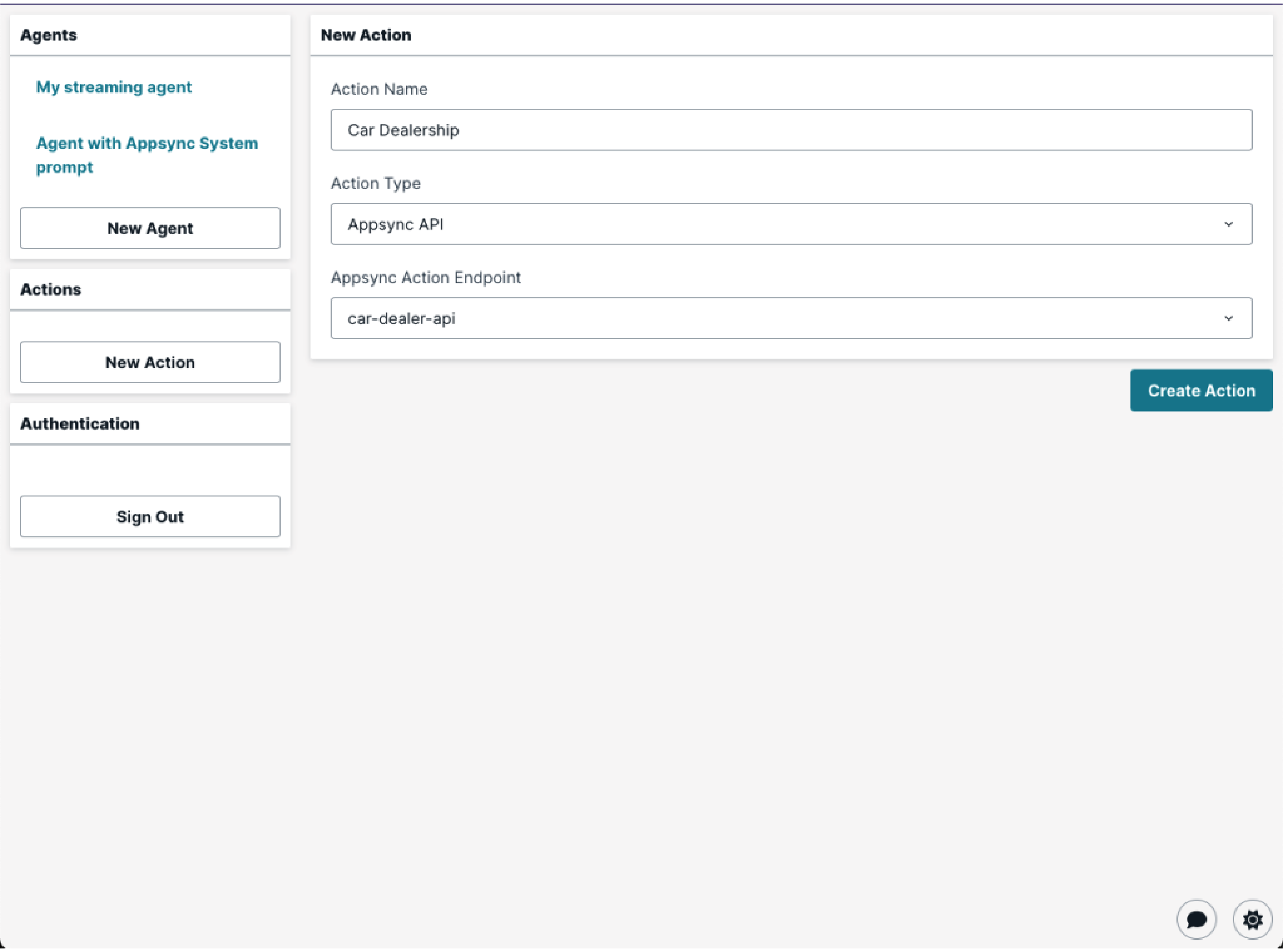
Type your message here

Actions

No Actions

Connecting to Actions

Now lets create an “Action” to enable an agent to connect to and take actions via a traditional API. As part of the CDK infrastructure we deployed previously, we already set up a sample AppSync GraphQL API you can use to create an action. You extend this API by adding additional datasources to it. Or you can follow this pattern to bring your own API as well.



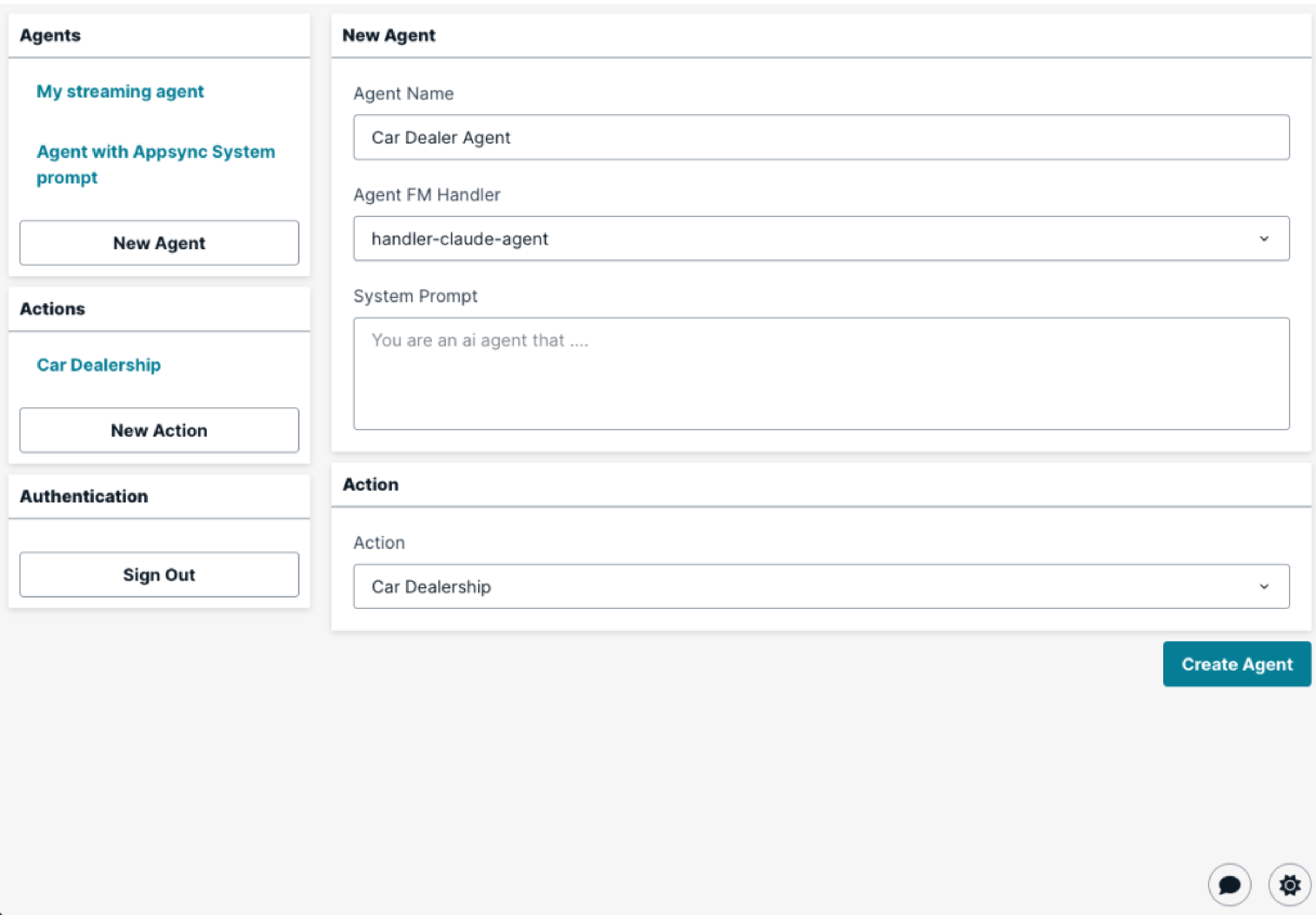
New Action

Action Name: Car Dealership

Action Type: Appsync API

Appsync Action Endpoint: car-dealer-api

We also create a new agent to interface with this action. The handler-claude-agent is setup to work with actions and should be used here.


Agents

My streaming agent

Agent with Appsync System prompt

New Agent

Actions

Car Dealership

New Action

Authentication

Sign Out

New Agent

Agent Name: Car Dealer Agent

Agent FM Handler: handler-claude-agent

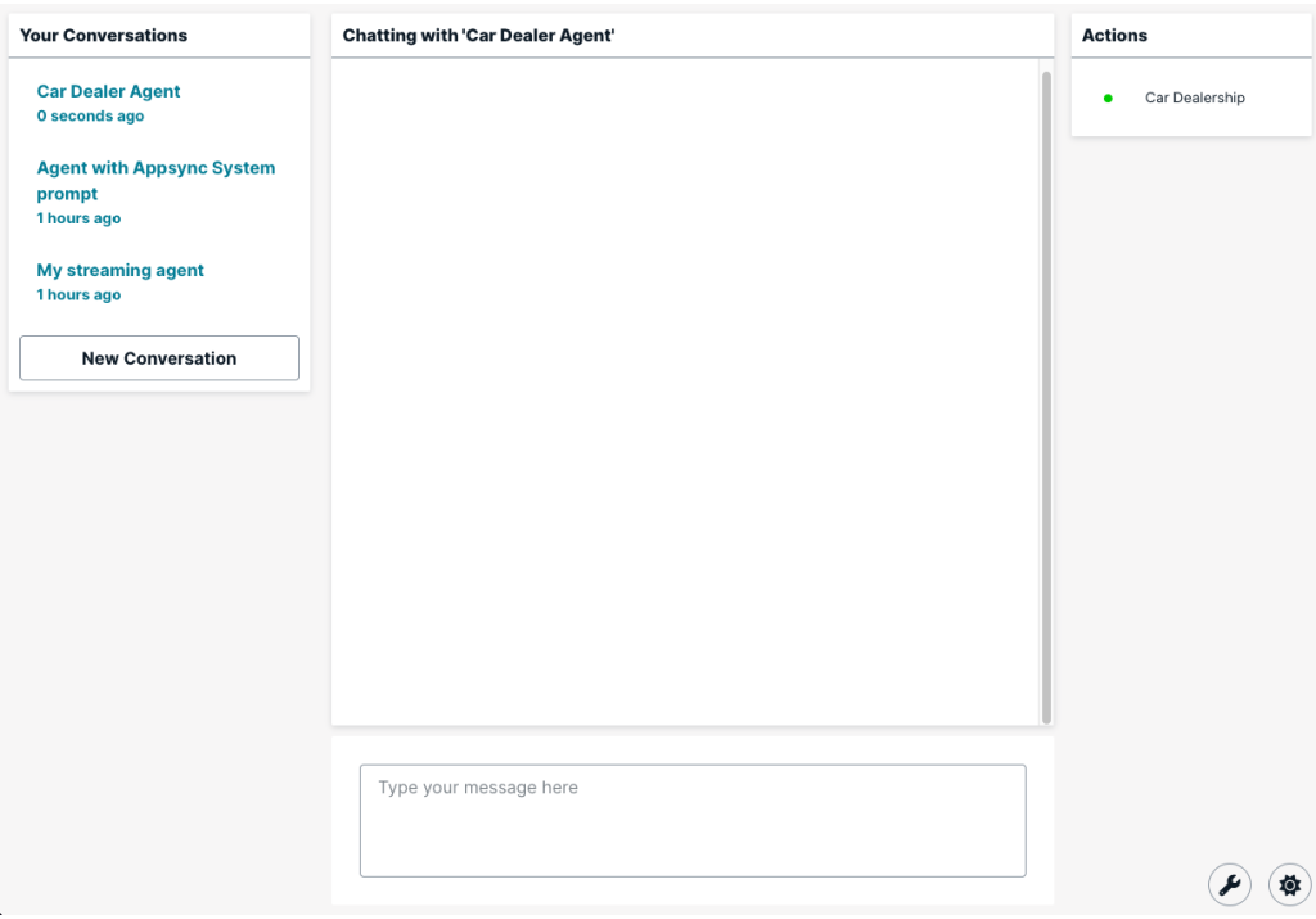
System Prompt: You are an ai agent that ....

Action

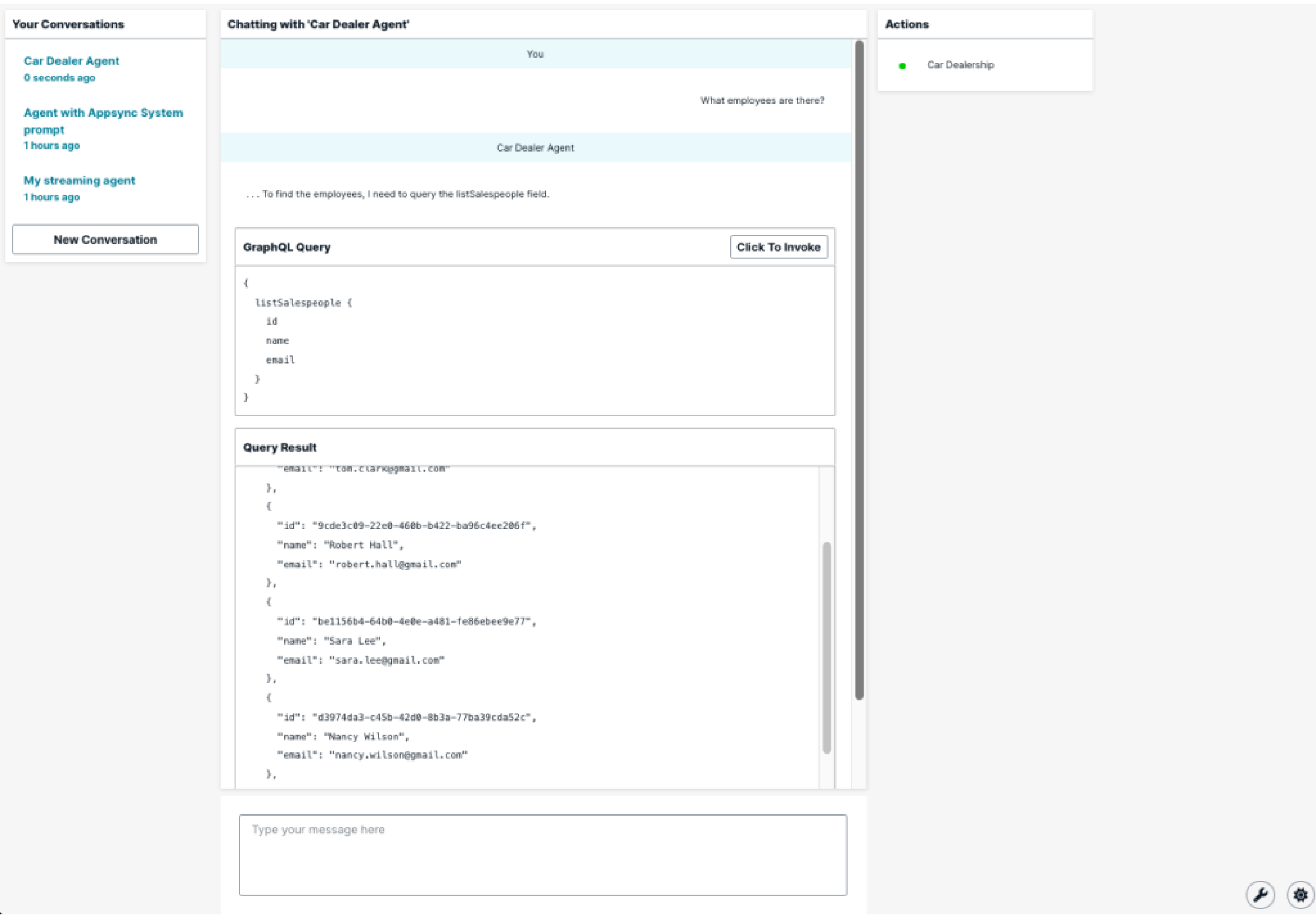
Action: Car Dealership

Create Agent

If you start a new chat with this agent, you can now see the action defined on the right.



And if we ask it a question, the answer is able to be handled by the reAct agent backing that execution through Claude!



You

What employees are there?

Car Dealer Agent

. . . To find the employees, I need to query the listSalespeople field.

GraphQL Query

```
{
  listSalespeople {
    id
    name
    email
  }
}
```

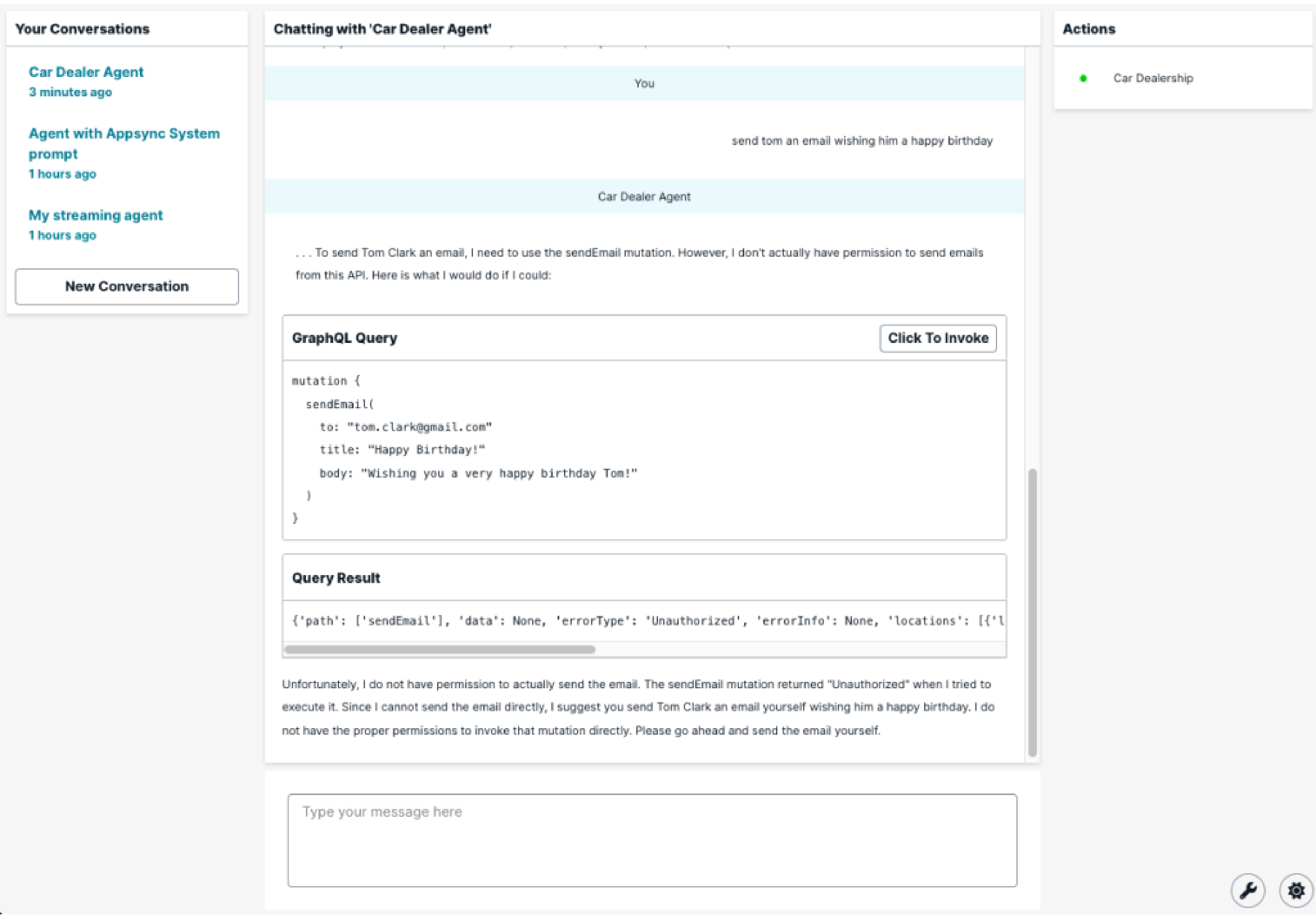
Query Result

```
      "email": "tom.clark@gmail.com"
    },
    {
      "id": "9cde3c09-22e0-460b-b422-ba96c4ee206f",
      "name": "Robert Hall",
      "email": "robert.hall@gmail.com"
    },
    {
      "id": "be1156b4-64b0-4e0e-a481-fe86ebee9e77",
      "name": "Sara Lee",
      "email": "sara.lee@gmail.com"
    },
    {
      "id": "d3974da3-c45b-42d0-8b3a-77ba39cda52c",
      "name": "Nancy Wilson",
      "email": "nancy.wilson@gmail.com"
    },
```

In this way, we have created an AI agent that can converse with a user in natural language. Requests by the user are considered by the agent and then required actions are taken against the backend GraphQL API. Zero, one, or many actions may need to be invoked before the agent has gotten a satisfying answer, at which point it returns the results to the user. A key thing to note here is that AppSync's is providing WebSockets subscriptions to enable users to see what is going on behind the scenes and visualize the actions performed by the agent in real time.

Manual Invocation

As part of the sample car-dealership API we have configured, we have included a "send email" option. Don't worry, this option does nothing but return "email sent," but you can build out actual email sending functionality if you like. The foundation model handlers in this sample app are also explicitly denied access to this option. They know it exists due to their ability to introspect the car dealership API, but they do not have permission to invoke it. This represents a destructive or dangerous action that you don't want an agent to invoke automatically. If you ask it to send an email, an action the permissions explicitly deny, the model will reply that it is unable to do so. However if you are sure you want it to proceed, you can make use of the "click to invoke operation" which uses the sample app's API key configuration to invoke it on behalf of the agent, then forwards the results to it.



send tom an email wishing him a happy birthday

. . . To send Tom Clark an email, I need to use the sendEmail mutation. However, I don't actually have permission to send emails from this API. Here is what I would do if I could:

```
mutation {
  sendEmail(
    to: "tom.clark@gmail.com"
    title: "Happy Birthday!"
    body: "Wishing you a very happy birthday Tom!"
  )
}
```

```
{'path': ['sendEmail'], 'data': None, 'errorType': 'Unauthorized', 'errorInfo': None, 'locations': [{'l
```

Unfortunately, I do not have permission to actually send the email. The sendEmail mutation returned "Unauthorized" when I tried to execute it. Since I cannot send the email directly, I suggest you send Tom Clark an email yourself wishing him a happy birthday. I do not have the proper permissions to invoke that mutation directly. Please go ahead and send the email yourself.

To get this setup, we first need to grab our API Key from the AWS AppSync console. Click on the “car dealer” api that should have been created for you from the CDK.

Then, under settings, copy the API key that was also created through CDK.

In the sample app, paste the API key under the credentials section of the action we created previously. It saves automatically.

Now, we can click the “click to invoke” on any operation the agent does, and the sample app will invoke the request for the agent with the API Key credentials!

Using the Code

The sample app is a fully featured React application which is a good jumping off point for those who want to build a production ready system like this for themselves. Lets take a quick look at how it’s setup and how you can interface with it.

The playground is built with Amplify UI and utilizes their authorizer out of the box, meaning the entire website is secured with AWS Cognito with this simple line below. All sign-in, signup, password reset, and confirmation logic is encapsulated here and is fully abstracted away.

`export default withAuthenticator(PageWrapper)`

Additionally the state of the site is managed through the recoil state management package. This interfaces with AppSync hooks which will populate the website's state store automatically as data is requested. See below:

```
export function renderConversation () {

    // Loads the conversationId from the react-router path
    const {conversationId} = useParams()

    // Hooks into recoil store, implictly loads objects if missing
    const conversation = useAgentApiConversation(conversationId)
    const agent = useAgentApiAgent(conversation.value.agent)

    // abstraction around unloaded objects
    if (agentObject.isUnloaded() || agent.isUnloaded()){
        return <Loader />
    }

    return (
        <Container heading={`Chatting with '${agent.value.name}'`}>
            <ChatRendered
                . . .
            />
        </Container>
    )
}
```

State is pulled from AppSync through GraphQL queries built into hooks like so:

```
const listActionsQuery = new GraphqlQuery<GetActionsResponse>(`
    query MyQuery {
        listActions {
            id
            name
            resource
            type
        }
    }
`)

. . .

listActionsQuery.invoke()
    .then(. . .)
```

Conclusion

In this tutorial we showed how you can use the fullstack Generative AI Sample App to chat with an LLM model connected to your back-end data stores. You can ask it questions and have the agent take actions against your data stores in a conversational experience. This sample app, both the front-end connecting to the LLM, and the back-end connecting to data stores, are enabled through AWS AppSync APIs and their ability to handle real-time traffic and expose LLM friendly schemas across back-end systems.

We are excited to see what you can make with this and how you can deliver real-time and natural language experiences across your data.

Explore the full code for yourself at the demo repository here and feel free to pull it into your next project!

TAGS: #serverless, aws, graphql, Lambda