Technical Report

SCALE - Serverless Chat Architecture for LLM Experiments

Sebastian Valet & Johannes Walter

June 5, 2025

Contents

Sebastian Valet, ZEW Mannheim and KIT, sebastian.valet@zew.de
Johannes Walter, ZEW Mannheim and KIT, johannes.walter@zew.de
This is the technical report for SCALE, a **S**erverless **C**hat **A**rchitecture for **L**arge **L**anguage **M**odels. It is meant to be a working document that is updated as we add new features.

1. Introduction

Online experiments involving chat interactions between participants and large language models (LLMs) are gaining popularity in the social sciences. Yet running these experiments at scale poses technical challenges. For example, while the oTree framework is widely used for online behavioral experiments, it struggles with handling many concurrent requests. This is primarily because oTree is single-threaded and not optimized for high-throughput API interactions, leading to performance bottlenecks.

These limitations can result in sluggish response times, a frustrating user experience, and increased server costs if concurrent user numbers must be restricted. A scalable and cost-effective solution is to offload the LLM interactions to AWS Lambda, a serverless compute service that allows you to run backend code without provisioning or managing servers. While AWS Lambda is a paid service, it includes a free tier that is sufficient for many academic experiments. It is easy to set up, highly scalable, and well-suited for chat-based applications.

In our architecture, the experimental back-end and the web front-end are built with oTree, while the chat between participant and AI is handled by an AWS Lambda function. The participant's front-end communicates with the Lambda function via standard API calls, enabling smooth and scalable LLM interactions.

Several alternative implementations exist for integrating LLMs into experimental designs. For example, Chopra and Haaland (2023) uses Qualtrics to conduct interviews with LLMs, and McKenna (2023) integrates LLMs directly into oTree's back-end. In contrast, our setup is more modular and scalable: while we use oTree here as a reference implementation, the approach is compatible with other platforms and experimental frameworks.

This guide will walk you through setting up and deploying an AWS Lambda function that connects to OpenAI's API. The code is flexible and can be easily adapted for use with other LLM providers.

2. Setup

The setup process involves configuring the AWS environment, creating necessary resources such as S3 buckets and DynamoDB tables, and deploying the Lambda function using AWS Serverless Application Model (SAM). The setup script `a_init.sh` automates these tasks, ensuring that all prerequisites are met before deployment.

The deployment script `b_deploy.sh` is responsible for building and deploying the Lambda function to AWS. It uses the AWS SAM CLI to package the application, including all dependencies, and deploys it to the cloud. The script ensures that the Lambda function is properly connected to the necessary AWS resources, such as the S3 bucket and DynamoDB table, and securely stores the OpenAI API key as a Lambda environment variable. We assume that you have created an OpenAI account and have an API key.

At the time of writing, the AWS free tier includes one million free requests per month and 400,000 GB-seconds of compute time per month (see AWS Lambda pricing). For DynamoDB, the free tier includes 25 GB of storage and 25 reads and writes per second (see Amazon DynamoDB pricing). Both should be sufficient for many academic experiments. To avoid unexpected charges when exceeding free tier limits, consider setting up billing alerts in the AWS Console and monitoring usage through CloudWatch metrics during active experiments.

2.1. Lambda function setup

a. **Create AWS Account and IAM User:**
   Sign up for AWS on the AWS website. In the AWS Console, go to the IAM service and create a new user with *programmatic access*. On the permissions step, choose *Attach policies directly* and select the following:
   - `AmazonS3FullAccess`
   - `AmazonDynamoDBFullAccess`
   - `AWSCloudFormationReadOnlyAccess`

    - `IAMFullAccess`
    - `AWSLambda_FullAccess`

   After creating the user, download the credentials (Access Key ID and Secret Access Key) for later use.

b. **Install Prerequisites:**

   Install the following tools on your system:

    - **Docker Desktop**: Required to build and run Lambda functions locally.
    - **AWS CLI**: The command-line tool to interact with AWS services.
    - **AWS SAM CLI**: Used to build, test, and deploy serverless applications.
    - **Shell Environment**: For Windows users: either **WSL (Ubuntu 22.04)** for Linux-like compatibility on Windows or **Git Bash** for a simpler shell interface.

c. **Clone the Repository:**

   Fork or clone this repository to your local machine.

d. **Configure the Setup Script:**

   Open `a_init.sh` and set your S3 bucket name (`BUCKET_NAME`) to any name you like. Optionally, adjust the action flags if needed (see `a_init.sh` for more details).

e. **Run the Setup Script:**
   In your terminal (e.g. WSL or Git Bash for Windows users), navigate to the project directory and run:

```
bash a_init.sh --aws-access-key-id <YOUR_KEY> --aws-secret-access-key <YOUR_SECRET>
```

f. **Configure the Deployment Script:**

   Open `b_deploy.sh` and ensure the bucket and table names match those you chose in the setup step when you ran `a_init.sh`. Add a production origin to the `ALLOWED_PROD_ORIGIN` variable if you want to have a stricter CORS policy; by default, it is set to `*` to allow all origins.

g. **Deploy the Lambda Function:**
   Make sure Docker Desktop is running. For this to work with OpenAI's API, you need an OpenAI account and an API key. Deploy with:

```
bash b_deploy.sh --openai-api-key <YOUR_OPENAI_API_KEY>
```

h. **Note the API Gateway Endpoint:**

   After deployment, the script will output an API Gateway endpoint URL. Use this

URL to send requests to your Lambda function.

Tips: If you encounter permissions errors, ensure your IAM user has the correct policies. You can observe errors online in the AWS CloudWatch logs. If you get errors about resources already existing, you can safely ignore them if you're re-running the setup. Always keep your AWS credentials and OpenAI API key secure.

Security Considerations: This setup exposes the API Gateway endpoint in frontend JavaScript, making it publicly accessible. Anyone with knowledge of the API endpoint can send requests to it. To limit potential misuse, we have implemented a CORS policy that restricts access to the API Gateway endpoint to the origin of the oTree front-end, and have have rate limited the API Gateway to 15 requests per second. Additionally, you can specify a list of allowed user IDs in the `user_config.py` file, such that only participants with a known user ID can use the chat. In `main.py`, limits for the maximum number of messages per chat, and the number of new chats per user can be set. For temporary deactivation of the Lambda function between experiments, you can set the concurrency limit to 0 in the AWS Console under *Configuration* → *Concurrency and recursion detection* to block execution of the Lambda function.

2.2. Key Lambda function files

For the basic setup, you don't need to change anything in the lambda.py or app.py files. Therefore, here is a brief overview of the files:

lambda.py: This is the main entry point for the AWS Lambda function. It handles incoming API Gateway requests and routes them to the appropriate function based on the 'route' field in the request. The supported routes are:

- `initialize`: to start a new chat session,
- `chat`: to send a user message and receive an AI response,
- `history`: to retrieve the chat history.

main.py: This file contains the core logic for managing chat sessions and interacting with the OpenAI API. It includes the functions to initialize a chat (`initialize_chat`), send a message and receive an AI response (`add_message_and_get_response`), and retrieve the full chat history (`get_chat_history`). Messages are stored in DynamoDB, and OpenAI's responses are generated using the GPT-4o model. If needed, you can customize the default model, token limits, or system prompts directly in this file.

2.3. Otree implementation

The oTree framework (Chen, Schonger, and Wickens 2016) is a popular tool for running experiments in the social sciences. In this project, we use the oTree-template repository as a starting point front-end that you can adapt to run your own experiments.

a. In your command line, navigate to the `oTree-template` folder.
b. Set the environment variable `AWS_LAMBDA_API_ENDPOINT` to the API Gateway endpoint URL of your AWS Lambda function. The endpoint URL is displayed in your terminal after deploying the Lambda function, and can also be found in the AWS web interface.
c. Customize the system prompt and any initial user or assistant messages to your preferences in the `chat(Page)`-class in `chat/__init__.py`.
d. You can now run the oTree-front end in development mode by running the `otree devserver` command.
e. That's it!

The setup doesn't use the oTree back-end to make API calls to the Lambda function as this is what causes the problems with concurrent requests. Instead, API calls are made directly in the front-end, see `chat.html` and `chat.js`. Be aware that this means that oTree variables such as `participant.code` are passed to the front-end as a `js_vars` object.

3. Application Examples

Coming soon to a paper near you!

References

Chen, Daniel L, Martin Schonger, and Chris Wickens. 2016. “oTree—An open-source platform for laboratory, online, and field experiments.” *Journal of Behavioral and Experimental Finance* 9: 88–97.

Chopra, Felix, and Ingar Haaland. 2023. “Conducting Qualitative Interviews with AI.” *CESifo Working Paper No. 10666*.

McKenna, Clint. 2023. “oTree GPT.” Available at https://github.com/clintmckenna/oTree_gpt.