**Solution Architect Rest-Endpoint**

**Rough Verbal Idea given by the customer for their upcoming Product Requirement**

1. Expose a rest end point with the post method
   1. It accepts only json pay load
      1. Pay load size can be as kbs to max 10 mb
2. Receive the data and generate a unique id
   1. Respond that unique id to client
   2. Store these data to permanent storage
3. We can expect a traction or spike in traffic (no. of request)
   1. We need to respond within 2 second
   2. As soon as I respond back my data needs to be stored permanently
4. AWS

**Solution:**

1. By using an API Gateway WebSocket API in front of Lambda.
2. API Gateway handles connections and invokes Lambda whenever there’s a new event.
3. Scaling is handled on the service side.
4. To update our connected clients from the backend, we can use the [API Gateway callback URL](https://docs.aws.amazon.com/apigateway/latest/developerguide/apigateway-how-to-call-websocket-api-connections.html). The AWS SDKs make communication from the backend easy.

### **Solution Features:**

1. Web application that enables a user to make a request against a large dataset. It returns the results in a flat file over WebSocket. In this case, we’re querying, via Amazon Athena, a data lake on S3 populated
2. However, this solution could apply to any data store, data mart, or data warehouse environment, or the long-running return of data for a response.
3. **Amazon Athena** lets you parse JSON-encoded values, extract data from JSON, search for values, and find length and size of JSON arrays.
4. **About Athena**
   1. Athena is easy to use. Simply point to your data in Amazon S3, define the schema, and start querying using standard SQL. Most results are delivered within seconds.
   2. With Athena, there’s no need for complex ETL jobs to prepare your data for analysis. This makes it easy for anyone with SQL skills to quickly analyze large-scale datasets.

**For the frontend architecture**

1. create this web application using a JavaScript framework (such as JQuery).
2. Use API Gateway to accept a REST API for the data and then open a WebSocket connection on the client to facilitate the return of results:

**Key observations:**

1. WebPack improves the Initialization time across the board.
2. **Requiring only the DynamoDB client saves up to 176ms! In 90% of the cases, the saving was over 130ms. With WebPack, the saving is even more dramatic.**

**Architecture Diagram:**

Diagram

Description automatically generated

<https://lucid.app/lucidchart/invitations/accept/inv_230eecd0-e80c-4465-970b-d4e33dd71d89?viewport_loc=-218%2C16%2C2492%2C1320%2C0_0>

**Environment Variable: uuid**

I programmatically updates an environment variable before invoking the function.

Graphical user interface

Description automatically generated with medium confidence

**Steps**

1. The client sends a REST request to API Gateway. This invokes a Lambda function that starts the Step Functions state machine execution. The function then returns the execution ID to the client.
2. Using the data returned by the REST request, the client connects to the WebSocket API and sends the Step Functions execution ID to the WebSocket connection.
3. The WebSocket triggers a Lambda function which creates a record in Dynamo DB. The record is a key-value mapping of Execution ARN – ConnectionId.
4. After RunAthenaQuery is successful, Lambda will query DynamoDB to retrieve the ConnectionId associated with the current Execution.
5. Using the client-associated API Gateway WebSocket ConnectionId, Lambda updates the connected client over the WebSocket API that their long-running job is complete. It uses the REST API call post\_to\_connection.
6. The client receives the S3 results data over their WebSocket connection using the IssueCallback Lambda function through the callback URL from the API Gateway WebSocket API.

**The Alternate basic flow of the import process is as follows:**

1. The user makes an API, which is served by API Gateway and backed by a Lambda function. The Lambda function computes a signed URL granting upload access to an S3 bucket and returns that to API Gateway, and API Gateway forwards the signed URL back to the user.
2. At this point, the user can use the existing S3 API to upload files larger than 10MB.

Timeline

Description automatically generated

pass HTTP headers returned by the createPresignedPost request.

**Gotchas**

1. file must be the last field added to FormData.
2. Make sure to use the POST method, not PUT.
3. Do not set the Content-Type header in fetch. If you do, the browser won’t set the correct boundary and you’ll get a MalformedPOSTRequest error.
4. Again, be careful with small Expires values.
5. Again, ensure your function’s IAM role has the s3:PutObject permission.
6. Don’t forget CORS! (See above)

**Conclusion**

1. When application consumers poll for long-running tasks, it can be a wasteful, detrimental, and costly use of resources.
2. This post outlined multiple ways to refactor the polling method.
3. Use API Gateway to host a RESTful interface,
4. Step Functions to orchestrate your workflow,
5. Lambda to perform backend processing,
6. API Gateway WebSocket API to push results to your clients.
7. TAGS: [Amazon API Gateway](https://aws.amazon.com/blogs/compute/tag/amazon-api-gateway/), [AWS Lambda](https://aws.amazon.com/blogs/compute/tag/aws-lambda/), [AWS Step Functions](https://aws.amazon.com/blogs/compute/tag/aws-step-functions/), [serverless](https://aws.amazon.com/blogs/compute/tag/serverless/)