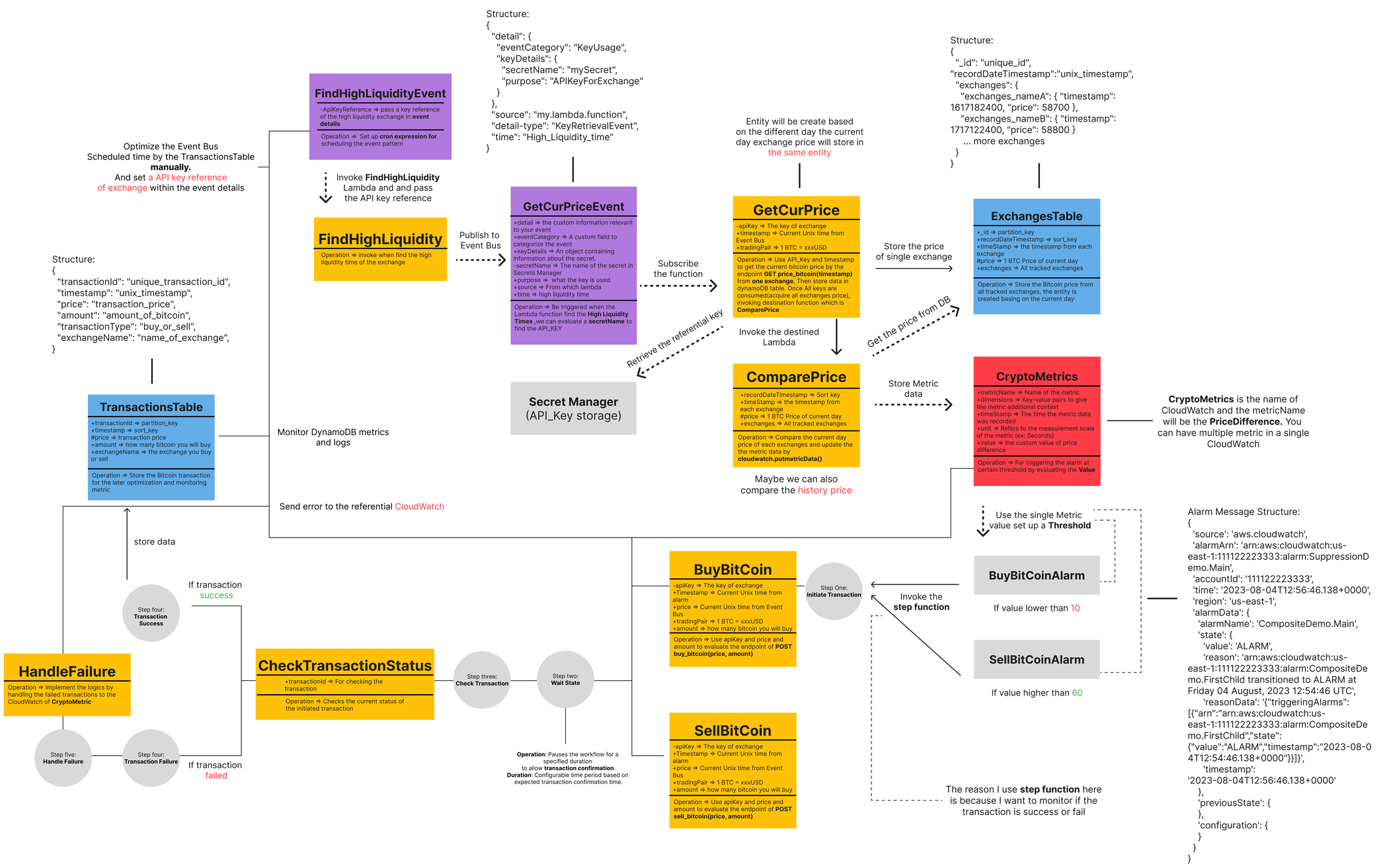
A Serverless event-drive architecture

EventBridge | Lambda function | CloudWatch | DynamoDB | Other

Reference: https://en.wikipedia.org/wiki/Class_diagram



The Benefit of system:

The benefits of implementing a serverless, event-driven architecture include not having to manage servers, with the added advantage of monitoring logs through a single CloudWatch interface. Although CloudWatch metadata might not be easily readable, third-party monitoring tools like Datadog or Serverless.com can be utilized for more in-depth analysis. This approach is also cost-effective; for instance, AWS Lambda offers 1 million free requests per month and 3.2 million seconds of compute time per month. Any additional costs incurred are generally minimal. Regarding security, each service can be interconnected using resource policies, and IAM roles can be employed to securely manage access permissions to various services. Moreover, The deployment and removal of our infrastructure can be efficiently managed as code, utilizing commands like `serverless deploy` and `serverless remove`, offering a robust and powerful system control

Service I choose:

1. Lambda function (Handle the business logic execution)
2. EventBridge (Sends a signal for execution)
3. CloudWatch(Monitoring, Alarm, Metric)
4. Step Function (like a state machine but for controlling the function execution)
5. DynamoDB (NoSQL database for storing the data)
6. Secret Manager (Store a private data)

System Process:

① Invoke the lambda(FindHighLiquidity) function by eventBridge which is scheduled manually by referring the TransactionsTable of dynamoDB and set a API_KEY reference of exchange within the event details.

② Invoke the lambda(GetCurPrice) function by eventBridge and retrieve the referential key from Secret Manager by event details. Then we get the price from this endpoint GET price_bitcoin(timestamp) and store the data to the ExchangesTable of dynamoDB from each exchange (one exchange / per invocation).

③ Once consume all the API_KEY from Secret Manager, we can then Invoke the destination lambda(ComparePrice) function, And this function will evaluate the ExchangesTable of dynamoDB storing the compared custom metric value to the CryptoMetrics of CloudWatch.

④ If the custom metric value is lower than 10, triggering the BuyBitCoinAlarm, or the value is higher than 60, triggering the SellBitCoinAlarm.

⑤ The alarm will sends a signal to initiate the execution of AWS Step Functions:
   a. Initiate Transaction: Start with either a 'BuyBitCoin' or 'SellBitCoin' Lambda function.
   b. Wait State: Add a pause to wait for a potential transaction confirmation.
   c. Check Transaction: Invoke a 'CheckTransactionStatus' Lambda function.
   d. Transaction Success: If successful, proceed to a 'Success' state and store the data to the TransactionsTable of dynamoDB for the later monitoring purpose in the CryptoMetrics of CloudWatch
   e. Transaction Failure: If failed or unconfirmed, proceed to a 'Failure' state.
   f. Handle Failure: In the 'Failure' state, choose to either retry the transaction or Send error to the CryptoMetrics of CloudWatch

⑥ Then we can start manually set up the event bus that from first step