AWS re:Invent

ANT222-R

Analytics with Amazon Athena

Roy Hassan
Principal BD
Amazon Web Services

Anthony Virtuoso
Principal Engineer
Amazon Web Services

AWS re:Invent

© 2019, Amazon Web Services, Inc. or its affiliates. All rights reserved.

aws

Available today (preview)

- Connect Athena to 10 new data sources
- Build your own user-defined functions (UDFs)
- Build your own connectors for custom data sources
- Apply Amazon SageMaker inference as part of your queries
- 100% open source SDK + connector suite

https://github.com/awslabs/aws-athena-query-federation

Available today (preview)

- Create an Athena workgroup called:

AmazonAthenaPreviewFunctionality

https://github.com/awslabs/aws-athena-query-federation

Agenda

Intro to Athena Federated Query (preview)

Connect Athena to Amazon Cloudwatch Logs

Build a custom data source

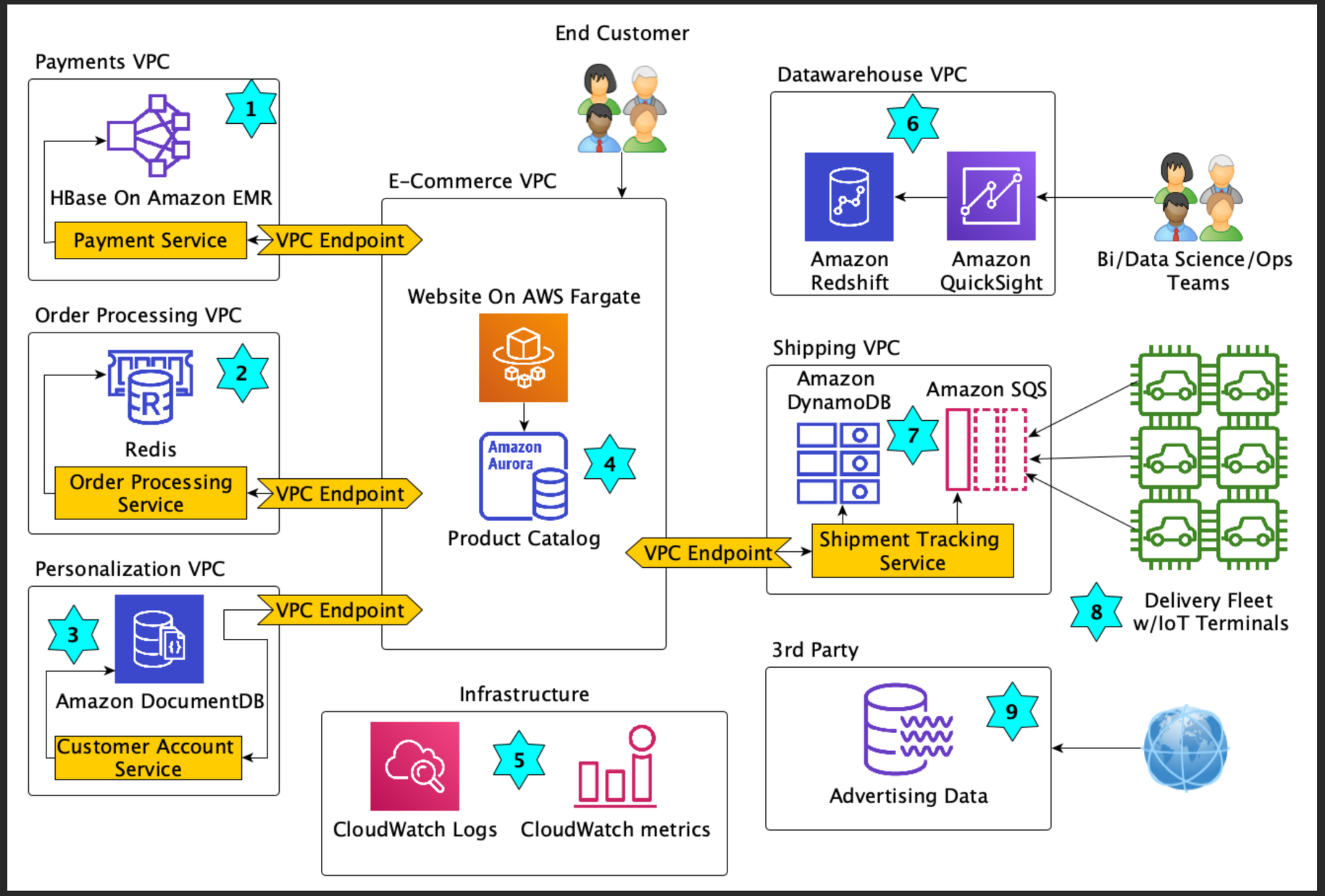
Build a user-defined function

Use Athena ML (preview) with Amazon SageMaker in a query

Data silos

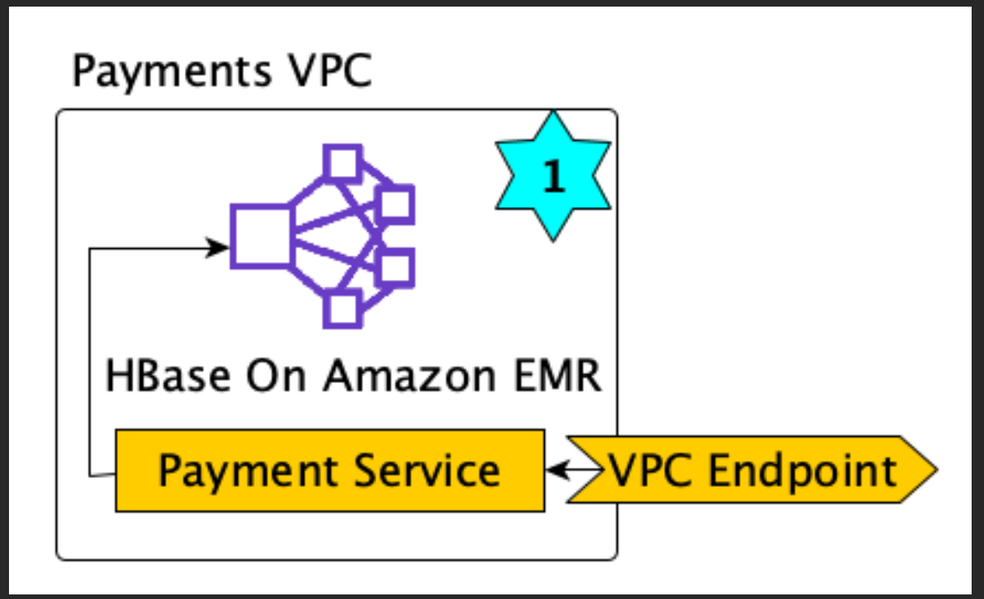
- Technology estate grows naturally over time
- Different use cases required different data storage or databases
- Diverse network topology (e.g., acquisitions, multiple sites, etc.)
- Preferred formats change over time
- Data lakes often miss:
  - Newly added fields
  - Frequently mutated data
  - Live system state
  - Transient data

Example architecture



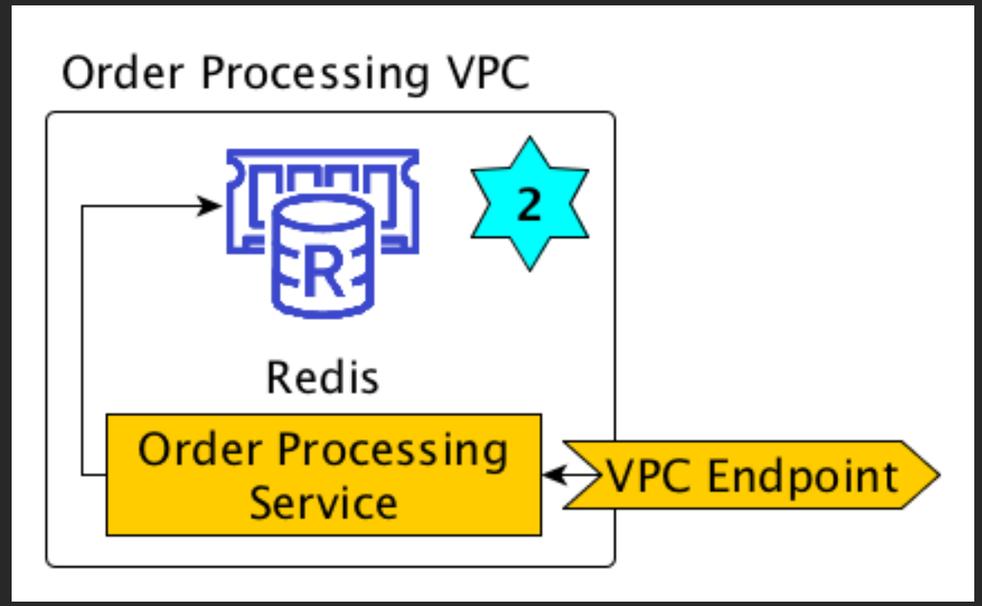
(1) Payments service

- Hbase on EMR, backed by Amazon S3
  - Required write-optimized data store
  - High transaction rate
  - Long-term durability
  - WebService on Amazon EC2
  - VPC endpoint exposed to e-comm platform



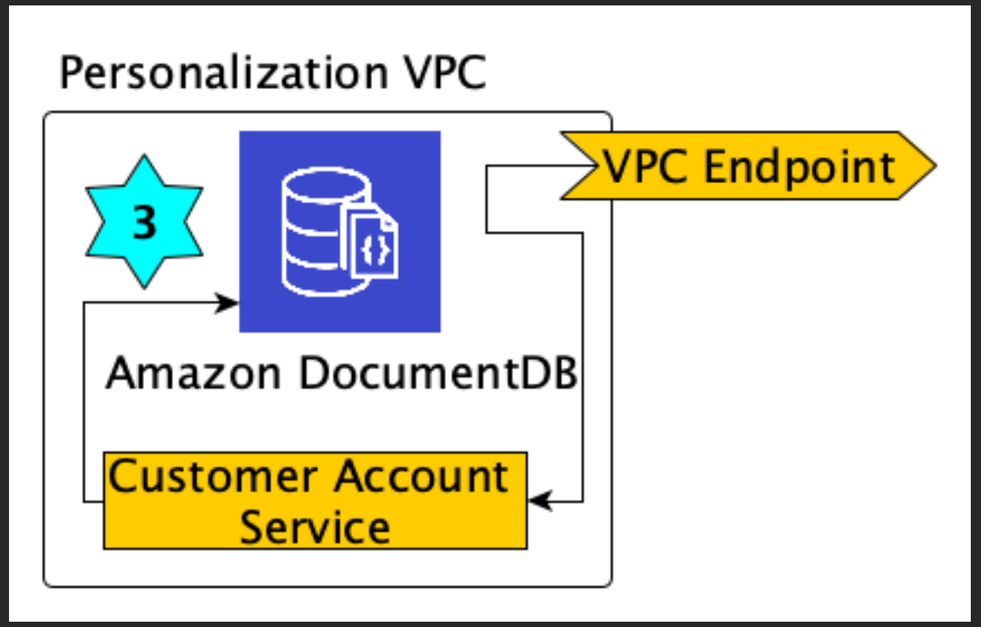
(2) Order processing service

- Redis on ElastiCache
  - Low-latency read-write
  - Minimal contextual info
  - In-memory w/rebuild capability
  - WebService on Amazon EC2
  - VPC endpoint exposed to e-comm platform



(3) Personalization service

- DocumentDB
  - Original WebService built on MongoDB
  - Customer account info varies by geography
  - VPC endpoint exposed to e-comm platform



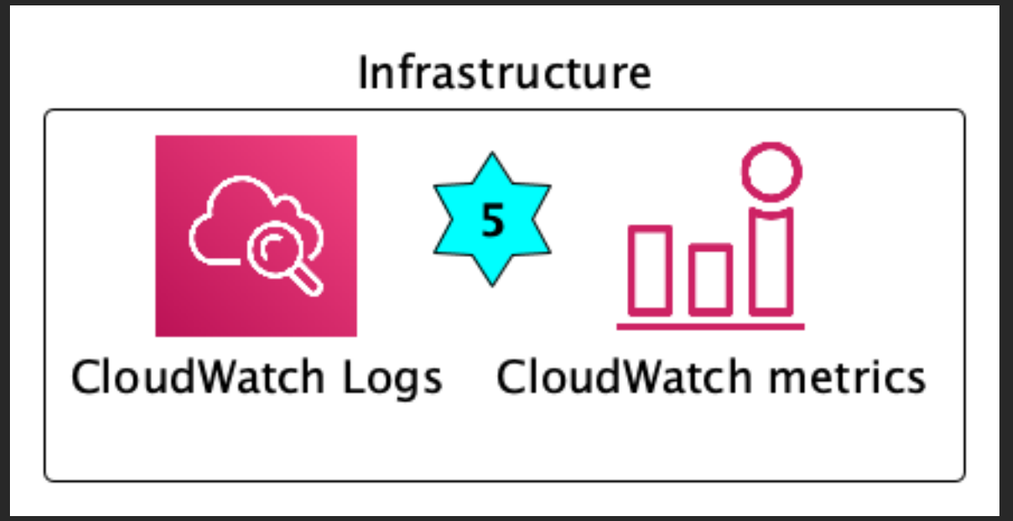
(4) E-commerce website

- Amazon Aurora
  - Scalable read-replicas
  - Low write-rate requirement
  - Fargate for auto-scaling front-end fleet
  - Requires VPC endpoint for all dependencies



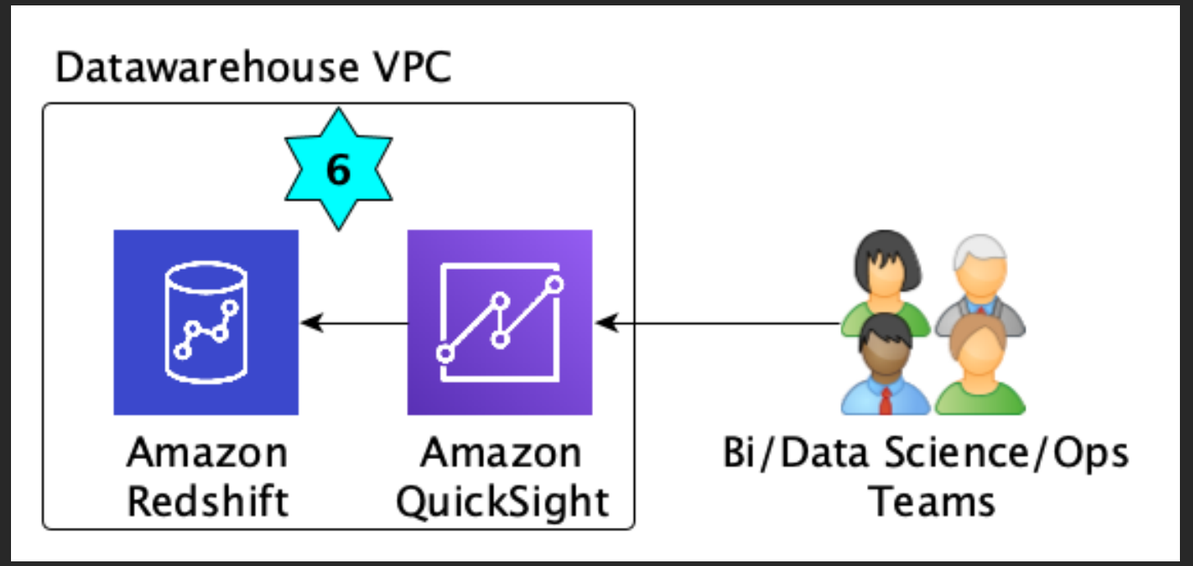
(5) Infra monitoring

- CloudWatch Logs & metrics
  - Integrated with all AWS services
  - Support custom app logs and metrics
  - Support for alarms



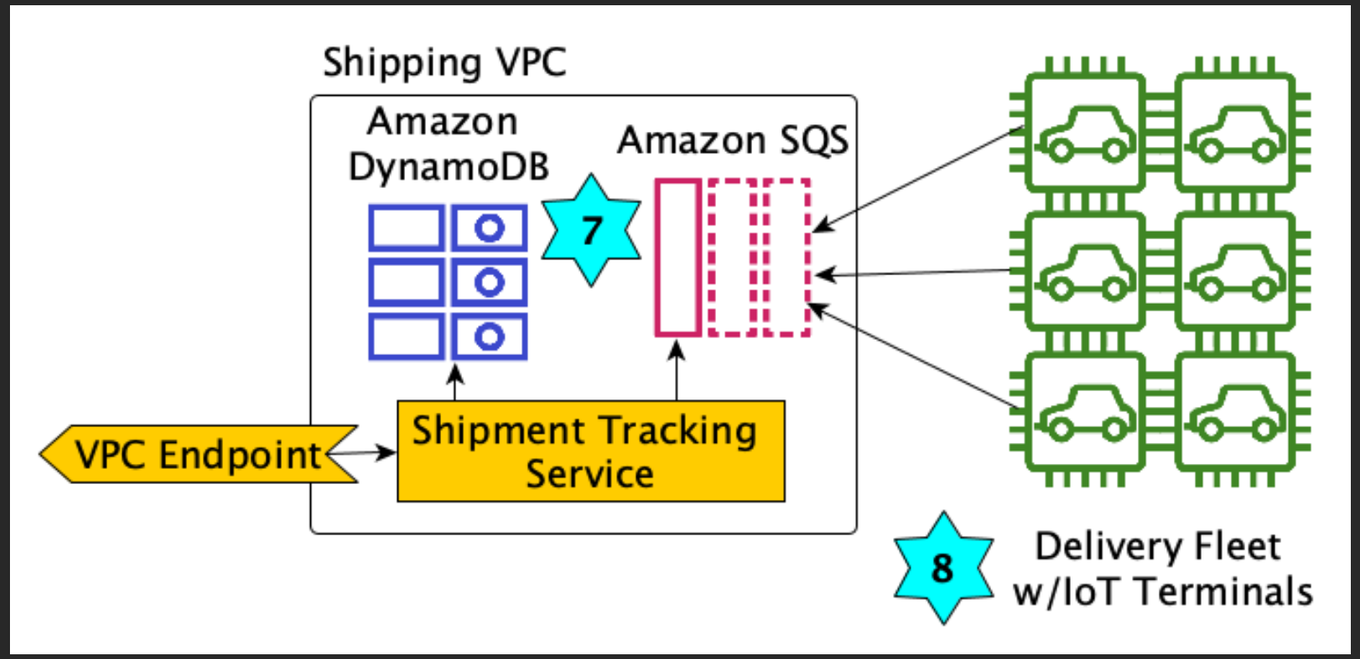
(6) Data warehouse

- Amazon Redshift
  - Write-once/read-many
  - Robust SQL support
  - Spectrum – long-term retention

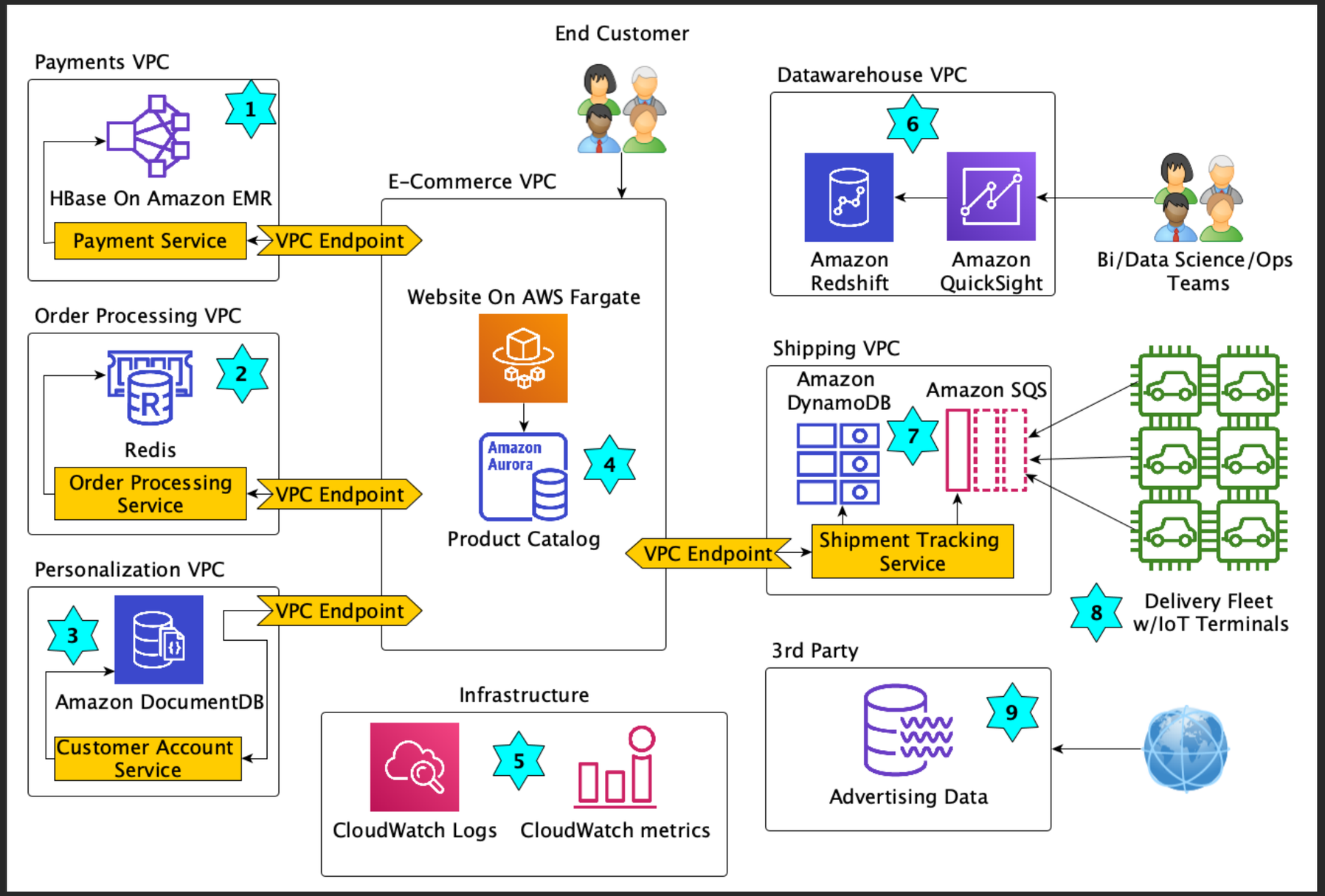


(7,8) Shipping service

- DynamoDB
  - Predictable performance
  - Schema varies by shipping method
- IoT devices
  - Live package status



Why are customer orders getting stuck?

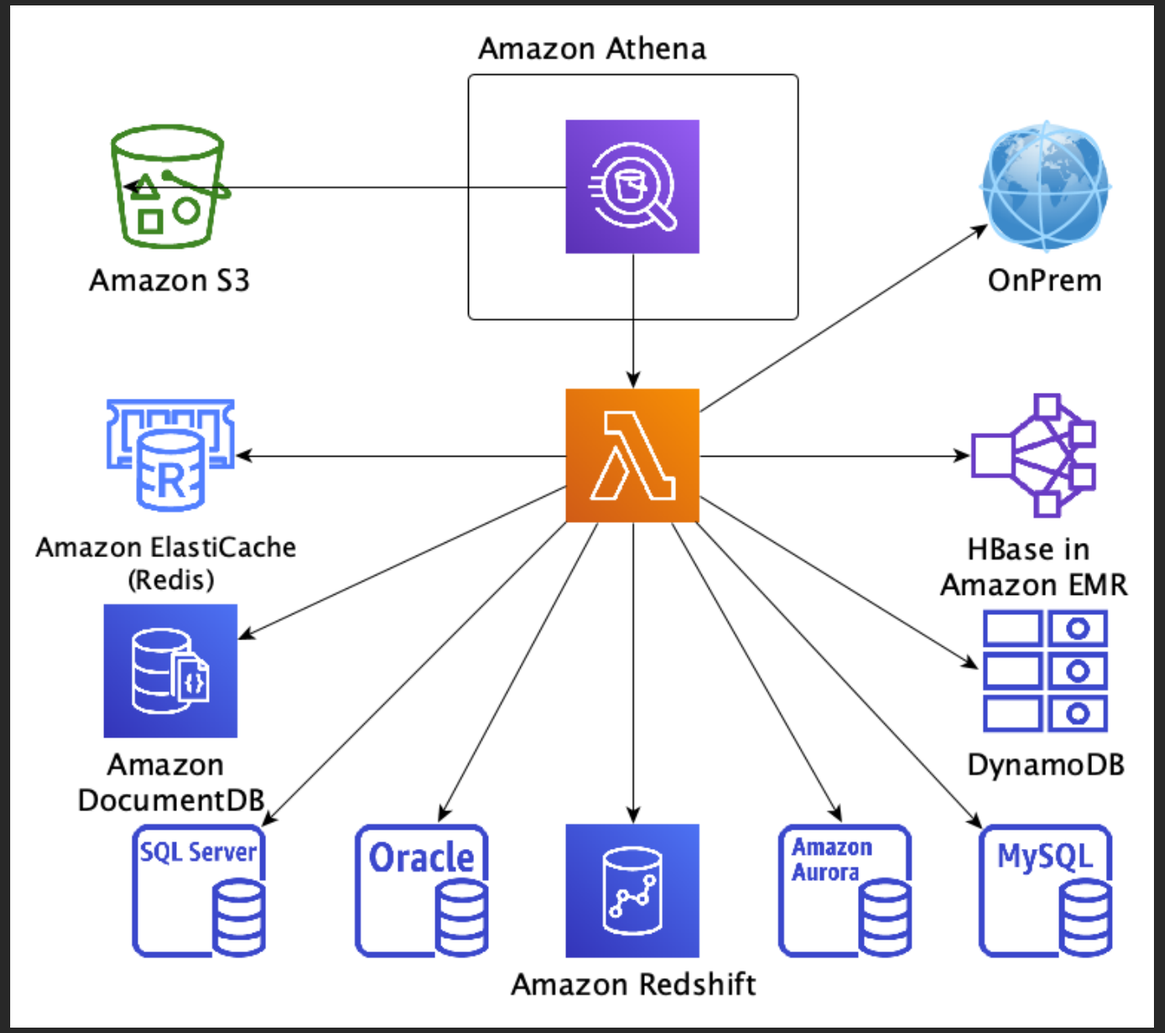


Federation demo

© 2019, Amazon Web Services, Inc. or its affiliates. All rights reserved.

Athena Federated Query (preview)

- 10 new sources available today
- Support for your own custom sources
- No ETLs. No servers. No problems.
- 100% open source
  - SDK & connectors on GitHub
- 1-click deployment



https://github.com/awslabs/aws-athena-query-federation

Available today (preview)

- Hbase
  - Parallelizes by region server and supports predicate pushdown
- Amazon DocumentDB
  - On-the-fly schema inference or configure explicit schema using the AWS Glue Data Catalog
  - Supports predicate pushdown
- DynamoDB
  - On-the-fly schema inference or configure explicit schema using the AWS Glue Data Catalog
  - Supports parallel scan and predicate pushdown
- JDBC
  - Works with Aurora, MySQL, Postgres, and Amazon Redshift
  - Supports parallel scans and predicate pushdown
- Redis
  - Use your Redis z-sets, hmaps, or key prefixes to define tables in the AWS Glue Data Catalog and then query them from Athena

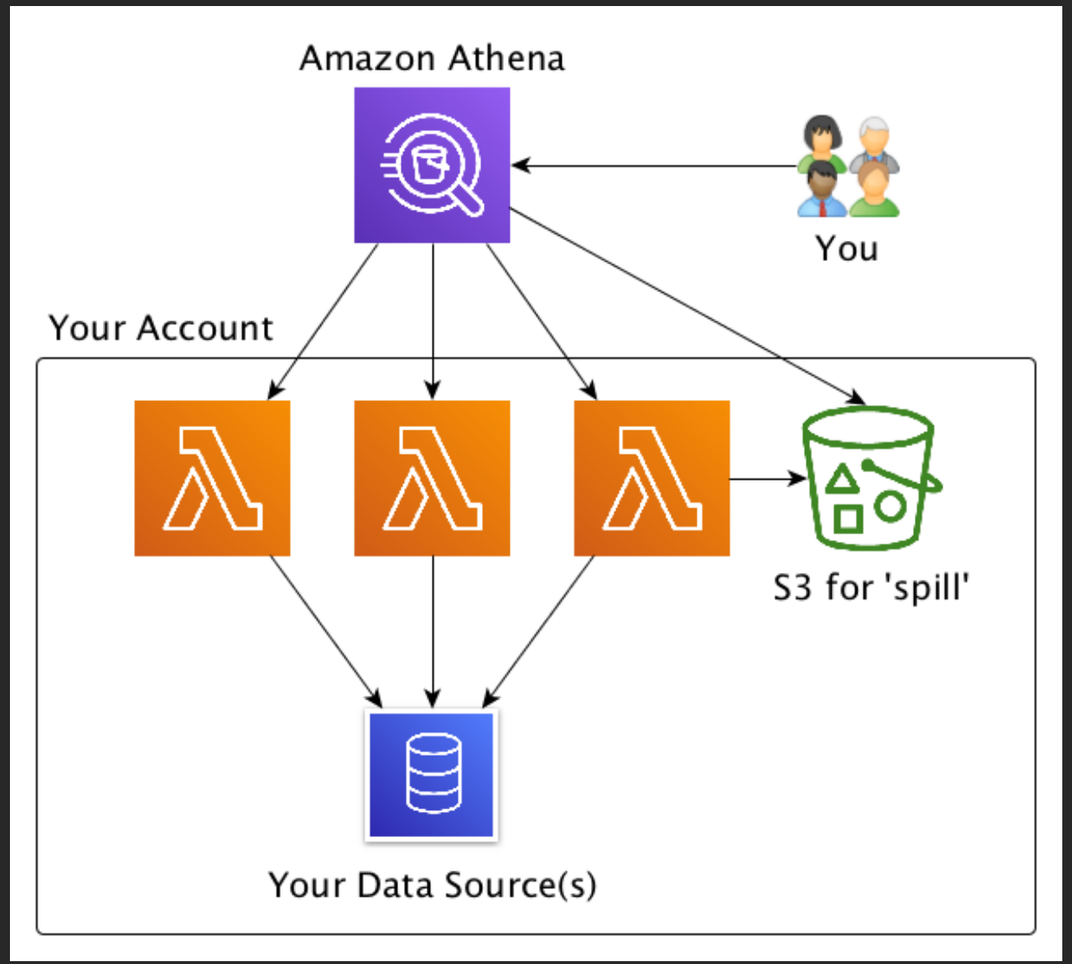
Available today (preview)

- CloudWatch Logs
  - Support parallel scan of log streams, predicate pushdown support, and rich regular expressions
- CloudWatch metrics
  - Support parallel scan of metric namespaces and dimension as well a predicate pushdown
- CMDB
  - Query resources as a Configuration Management Database (e.g., select id, hostname from ec2_instances;)
- TPCDS data generator
  - Supports parallel scans and predicate pushdown as a reference implementation for building your own connector
- Commonly requested UDFs
  - e.g., Ability to decrypt columns using encryption keys from AWS Key Management Service (AWS KMS) or AWS Secrets Manager

Serverless big data using Lambda

Tested beyond 1.5B rows/sec and 100Gbps in a single query when using parallel scans

- Partition pruning
- Parallel scans
- Portable columnar memory-format (Apache Arrow)
- Predicate pushdown
- Congestion control/avoidance



Lab 1:
Connect Athena to CloudWatch

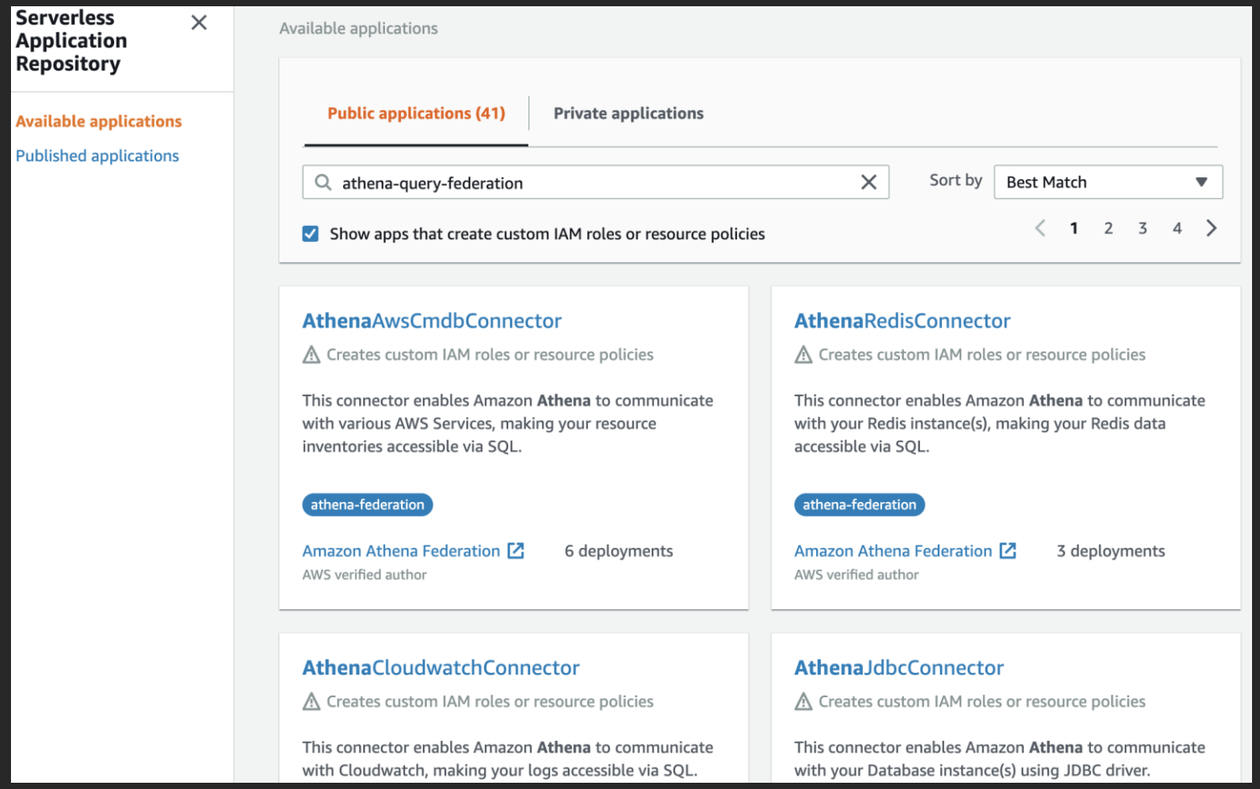
AWS re:Invent

© 2019, Amazon Web Services, Inc. or its affiliates. All rights reserved.

aws

Step 1: Connect to CloudWatch

- Open the AWS console
- Create an S3 bucket our connector will use
- Navigate to serverless application repository
- Search for ‘athena-query-federation’
- Click AthenaCloudWatchConnector
- Complete the form
- Click deploy



Step 2: Run an Athena query

```
SELECT *
FROM
“lambda:cloudwatch”.”/aws/lambda/cloudwatch”.all_log_streams
LIMIT 10;
```

Step 2: Run an Athena query

Lets see how we did

CloudWatch Logs connector

- Log Groups modeled as schemas (aka databases)
- Log Streams modeled as tables
- Special ‘all_log_streams’ table
- Supports parallel scans of log streams
- Supports predicate pushdown of time ranges
- Supports ‘limited’ scans based on LIMIT clause

Lab 2: Custom data source

AWS re:Invent

© 2019, Amazon Web Services, Inc. or its affiliates. All rights reserved.

aws

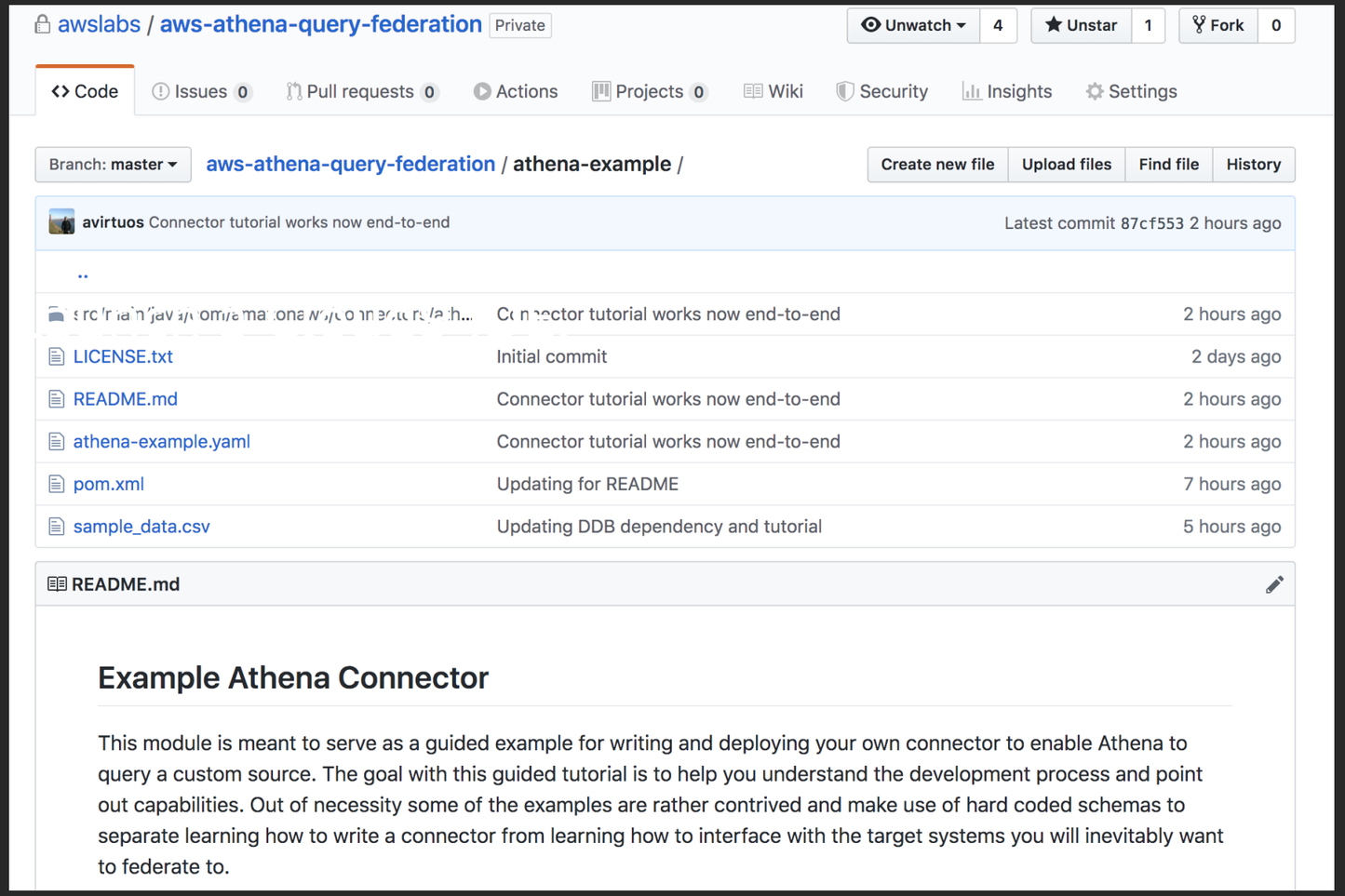
Lab 2: Custom data source

- Launch a AWS Cloud9 IDE
- Install prerequisites (maven, AWS CLI, Athena SDK)
- Modify the example connector project
- Modify the example UDF project
- Package the example connector
- Deploy the example connector via Serverless App Repository
- Run queries

Open the GitHub repository

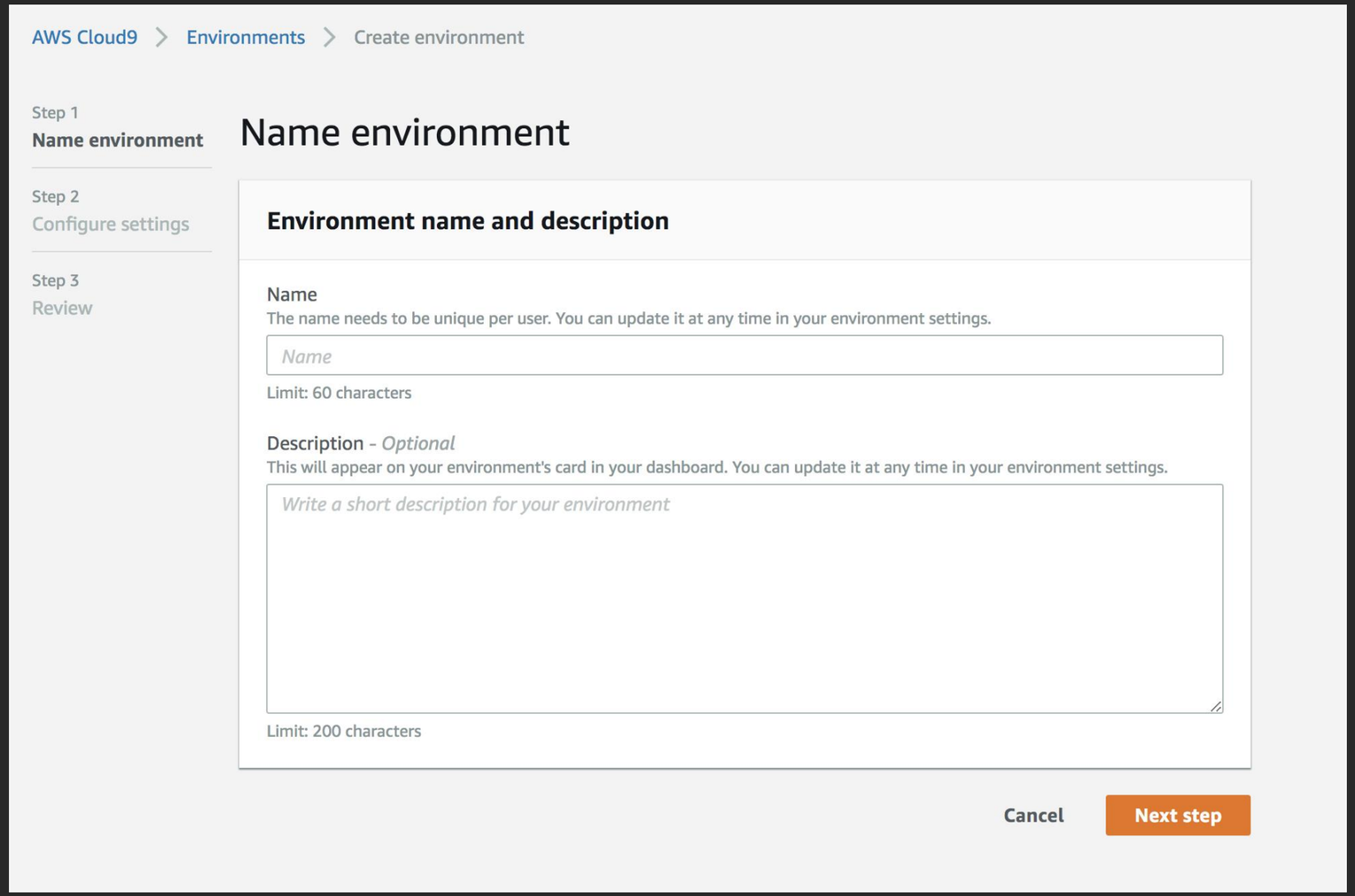
https://github.com/awslabs/aws-athena-query-federation

- Click on athena-example folder
- Step-by-step instructions
- Copy-paste commands
- Detailed context



Step 1: AWS Cloud9 setup

- Name it
- Select M4 instance type
- Launch it



Step 2: Download the SDK

- At the AWS Cloud9 terminal, run git clone ‘REPO URL’

https://github.com/awslabs/aws-athena-query-federation

Step 3: Prepare development environment

- ./tools/prepare_dev_env.sh
- Install maven
- Update JAVA_HOME
- Install HomeBrew
  - Update aws-cli
  - Update sam-cli
- Compile the SDK

```
#Install Maven and Java 8
sudo wget https://repos.fedorapeople.org/repos/dchen/apache-maven/epel-apache-maven
sudo sed -i s/\$releasever/6/g /etc/yum.repos.d/epel-apache-maven.repo
sudo yum install -y apache-maven
sudo yum -y install java-1.8.0-openjdk-devel
sudo update-alternatives --set java /usr/lib/jvm/jre-1.8.0-openjdk.x86_64/bin/java
sudo update-alternatives --set javac /usr/lib/jvm/java-1.8.0-openjdk.x86_64/bin/jav

# If the above update-alternatives doesn't work and you don't know your path try
#  sudo update-alternatives --config java
#  sudo update-alternatives --config javac


#Install HomeBrew so we can install aws cli and aws-sam
sh -c "$(curl -fsSL https://raw.githubusercontent.com/Linuxbrew/install/master/inst

#Prepare the Brew environment
test -d ~/.linuxbrew && eval $(~/.linuxbrew/bin/brew shellenv)
test -d /home/linuxbrew/.linuxbrew && eval $(/home/linuxbrew/.linuxbrew/bin/brew sh
test -r ~/.bash_profile && echo "eval \$($(brew --prefix)/bin/brew shellenv)" >>~/.
echo "eval \$($(brew --prefix)/bin/brew shellenv)" >>~/.profile

#Install the latest aws cli and sam tools (run these one at a time)
brew tap aws/tap
brew install awscli
brew install aws-sam-cli
```

Step 4: Write the code

- Create our S3 bucket used for sample data and spill
  - aws s3 mb s3://BUCKET_NAME
- Athena calls your Lambda function for:
  - List schemas
  - List tables
  - Table definition (columns and data types)
  - Partitioning information
  - Row data (the actual data in your tables)
  - Scalar UDFs

Step 4: Write the code – ExampleMetadataHandler

```java
    /**
     * Used to get the list of schemas (aka databases) that this source contains.
     * @param allocator Tool for creating and managing Apache Arrow Blocks.
     * @param request Provides details on who made the request and which Athena catalog they are querying.
     * @return A ListSchemasResponse which primarily contains a Set<String> of schema names and a catalog name
     *          corresponding the Athena catalog that was queried.
     */
    @Override
    protected ListSchemasResponse doListSchemaNames(BlockAllocator allocator, ListSchemasRequest request){}

    /**
     * Used to get the list of tables that this source contains.
     * @param allocator Tool for creating and managing Apache Arrow Blocks.
     * @param request Provides details on who made the request and which Athena catalog and database they are querying.
     * @return A ListTablesResponse which primarily contains a List<TableName> enumerating the tables in this
     *          catalog, database tuple. It also contains the catalog name corresponding the Athena catalog that was queried.
     */
    @Override
    protected ListTablesResponse doListTables(BlockAllocator allocator, ListTablesRequest request){}

    /**
     * Used to get definition (field names, types, descriptions, etc...) of a Table.
     * @param allocator Tool for creating and managing Apache Arrow Blocks.
     * @param request Provides details on who made the request and which Athena catalog, database, and table they are querying.
     * @return A GetTableResponse which primarily contains:
     *          1. An Apache Arrow Schema object describing the table's columns, types, and descriptions.
     *          2. A Set<String> of partition column names (or empty if the table isn't partitioned).
     */
    @Override
    protected GetTableResponse doGetTable(BlockAllocator allocator, GetTableRequest request){}
```

Step 4: Write the code

Lets see how we did

Step 4: Write the code – ExampleMetadataHandler

```
204     /**
205      * Used to get the partitions that must be read from the request table in order to satisfy the requested predicate.
206      *
207      * @param blockWriter Used to write rows (partitions) into the Apache Arrow response.
208      * @param request Provides details of the catalog, database, and table being queried as well as any filter predicate.
209      * @param queryStatusChecker A QueryStatusChecker that you can use to stop doing work for a query that has already terminated
210      * @note Partitions are partially opaque to Amazon Athena in that it only understands your partition columns and
211      * how to filter out partitions that do not meet the query's constraints. Any additional columns you add to the
212      * partition data are ignored by Athena but passed on to calls on GetSplits.
213      */
214     @Override
215     public void getPartitions(BlockWriter blockWriter, GetTableLayoutRequest request, QueryStatusChecker queryStatusChecker)
```

Step 4: Write the code

Lets see how we did

Step 4: Write the code – ExampleMetadataHandler

```
63      /**
64       * Used to split-up the reads required to scan the requested batch of partition(s).
65       * @param allocator Tool for creating and managing Apache Arrow Blocks.
66       * @param request Provides details of the catalog, database, table, andpartition(s) being queried as well as
67       *                any filter predicate.
68       * @return A GetSplitsResponse which primarily contains:
69       *          1. A Set<Split> which represent read operations Amazon Athena must perform by calling your read function.
70       *          2. (Optional) A continuation token which allows you to paginate the generation of splits for large queries.
71       * @note A Split is a mostly opaque object to Amazon Athena. Amazon Athena will use the optional SpillLocation and
72       *       optional EncryptionKey for pipelined reads but all properties you set on the Split are passed to your read
73       *       function to help you perform the read.
74       */
75      @Override
76      protected GetSplitsResponse doGetSplits(BlockAllocator allocator, GetSplitsRequest request)
```

Step 4: Write the code

Lets see how we did

Step 4: Write the code – ExampleRecordHandler

```
84      /**
85       * Used to read the row data associated with the provided Split.
86       *
87       * @param spiller A BlockSpiller that should be used to write the row data associated with this Split.
88       * The BlockSpiller automatically handles chunking the response, encrypting, and spilling to S3.
89       * @param recordsRequest Details of the read request, including:
90       * 1. The Split
91       * 2. The Catalog, Database, and Table the read request is for.
92       * 3. The filtering predicate (if any)
93       * 4. The columns required for projection.
94       * @param queryStatusChecker A QueryStatusChecker that you can use to stop doing work for a query that has already terminated
95       * @throws IOException
96       * @note Avoid writing >10 rows per-call to BlockSpiller.writeRow(...) because this will limit the BlockSpiller's
97       * ability to control Block size. The resulting increase in Block size may cause failures and reduced performance.
98       */
99      @Override
100 @    protected void readWithConstraint(BlockSpiller spiller, ReadRecordsRequest recordsRequest, QueryStatusChecker queryStatusChecker)
```

Step 4: Write the code

Lets see how we did

Step 4: Write the code – ExampleUserDefinedFuncHandler

```
52      /**
53       * This UDF extracts an 'Account' from the input STRUCT (provided as a Map). In this case 'Account' is
54       * an application specific concept and very custom to our test dataset's schema.
55       *
56       * @param transaction The transaction from which to extract the id field.
57       * @return An Integer containing the Transaction ID or -1 if the id couldn't be extracted.
58       *
59       * @note The UserDefinedFunctionHandler class that this class extends handles mapping the UDFs we use in our
60       * SQL queries into calls to this function.
61       */
62      public Integer extract_tx_id(Map<String, Object> transaction) {}
63
64      /**
65       * Decrypts the provided value using our application's secret key and encryption Algo.
66       *
67       * @param payload The cipher text to decrypt.
68       * @return ClearText version if the input payload, null if the decrypt failed.
69       *
70       * @note The UserDefinedFunctionHandler class that this class extends handles mapping the UDFs we use in our
71       * SQL queries into calls to this function.
72       */
73      public String decrypt(String payload){}
```

Step 4: Write the code

Lets see how we did

Step 4: Write the code

- Upload our sample data to Amazon S3
  - aws s3 cp ./sample_data.csv s3://BUCKET_NAME/2017/12/1/sample_data.csv

Step 5: Deploy your new connector

- Publish to SAR
  - ../tools/publish.sh S3_BUCKET_NAME athena-example AWS_REGION
- Go to SAR and click deploy
  - The publish command will give you a link (or you can go to SAR in the AWS console)

https://console.aws.amazon.com/serverlessrepo/

Step 5: Deploy your new connector

Lets see how we did

Step 6: Validate our connector

- Run the validation tool

```
../tools/validate_connector.sh --lambda-func <function_name> \
  --schema schema1 \
  --table table1\
  --constraints year=2017,month=11,day=1
```

Simulates an Athena query against your connector and quickly identifies common integration issues

Step 6: Validate our connector

Lets see how we did

Step 7: Create workgroup & run a query

From the Athena console, run the below query from the **AmazonAthenaPreviewFunctionality** workgroup

```
USING FUNCTION extract_tx_id(value ROW(id INT, completed boolean) )
        RETURNS INT TYPE LAMBDA_INVOKE
WITH (lambda_name = '<function_name>'), FUNCTION decrypt(payload VARCHAR )
        RETURNS VARCHAR TYPE LAMBDA_INVOKE
WITH (lambda_name = '<function_name>')
SELECT year,
          month,
          day,
          account_id,
          decrypt(encrypted_payload) AS decrypted_payload,
          extract_tx_id(transaction) AS tx_id
FROM "lambda:<function_name>".schema1.table1
WHERE year=2017
        AND month=11
        AND day=1;
```

Step 7: Create workgroup & run a query

Lets see how we did

Recap: Custom source details

- Implement just a few methods in your Lambda function:
  - Get Table – returns the schema of the request function
  - Get Partitions – (Optional) Perform partitioning pruning
  - Get Splits – (Optional) Tell Athena how to ‘split’ (aka parallelize) reads against your source
  - Read Records – Given a ‘split’ read the rows for Athena
- SDK automatically encrypts and spills large response to Amazon S3
  - Lambda has a 6MB response size limit
- Use multiple ’splits’ to get around the Lambda 15-minute max

Recap: User-defined functions

- Powered by AWS Lambda
- Simple to write
- Scalar functions
- Multiple UDFs per Lambda

Athena query

```
USING FUNCTION totalprice(quantity int, unitprice DOUBLE)
                RETURN DOUBLE TYPE lambda_udf
        WITH (lambda_udf='ecommerselambdaudf'),
USING FUNCTION isInternational(fullAddress VARCHAR) RETURN BOOLEAN
        TYPE LAMBDA_UDF WITH (lambda_udf='ECommerseLambdaUdf')
SELECT productname,
        productid,
        totalprice(productquantity, unitprice)
FROM    productcatalog
WHERE   isInternational(product.vendor.addr)
```

UDF Lambda code

```
public class ECommerceLambdaUdfHandler extends ScalarUdfHandler {

    public double totalPrice(int quantity, double unitPrice) {
        return quantity * unitPrice;
    }

    public boolean isInternational(String encryptedAddress) {
        String customerAddr = cipher.decrypt(encryptedAddress);
        return isInternational(customerAddr);
    }
}
```

Lab 3: Athena ML

AWS re:Invent

© 2019, Amazon Web Services, Inc. or its affiliates. All rights reserved.

aws

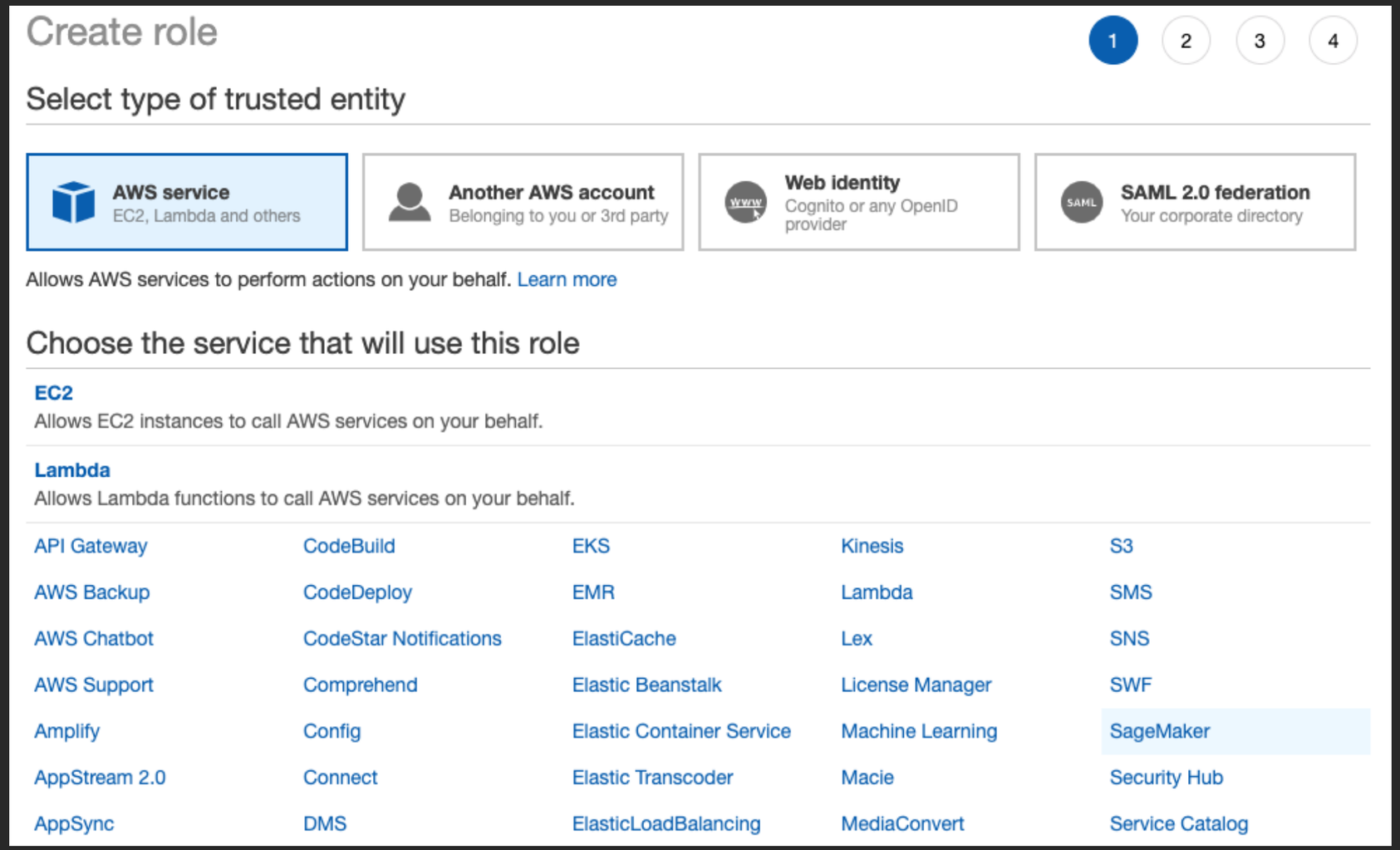
Amazon Athena ML (preview)

- Leverage Amazon SageMaker models for inference
- Supported algorithms: Linear Learner , XGBoost, Random Cut Forest, AutoML
- Supports CSV data with more formats in the future

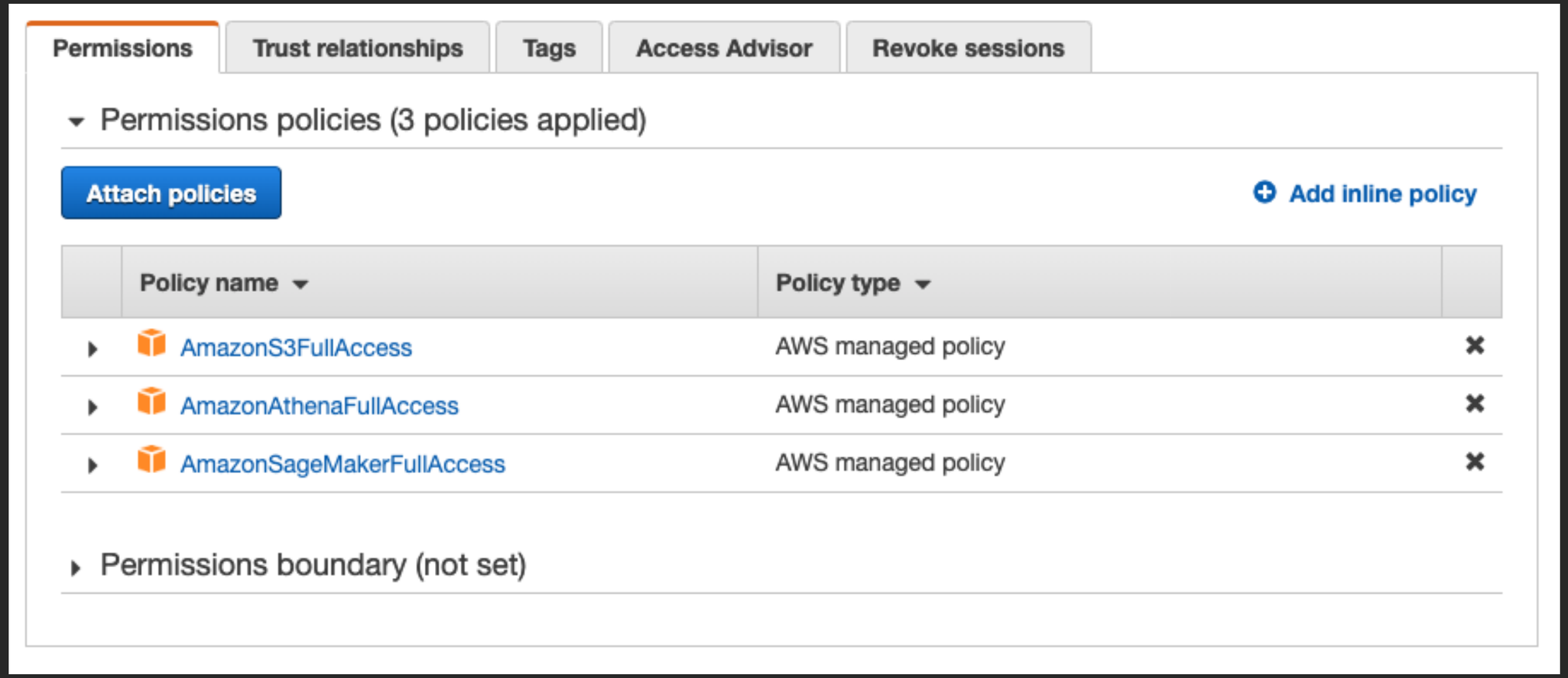
Open the tutorial on GitHub, link below, or at the end of athena-example.

https://github.com/awslabs/aws-athena-query-federation/wiki/AthenaML-Tutorial

Step 1: Create Amazon SageMaker role



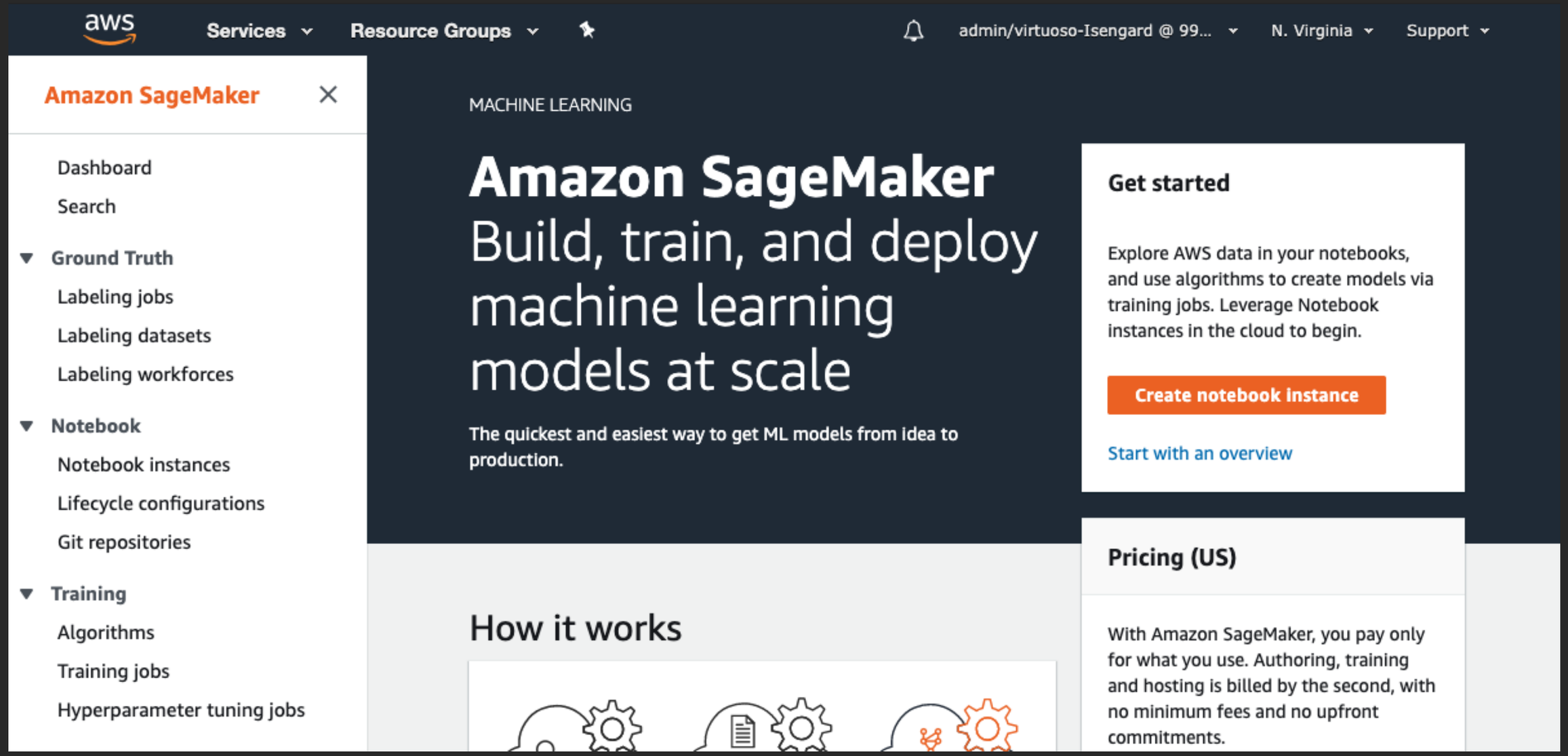
Step 1: Create Amazon SageMaker role



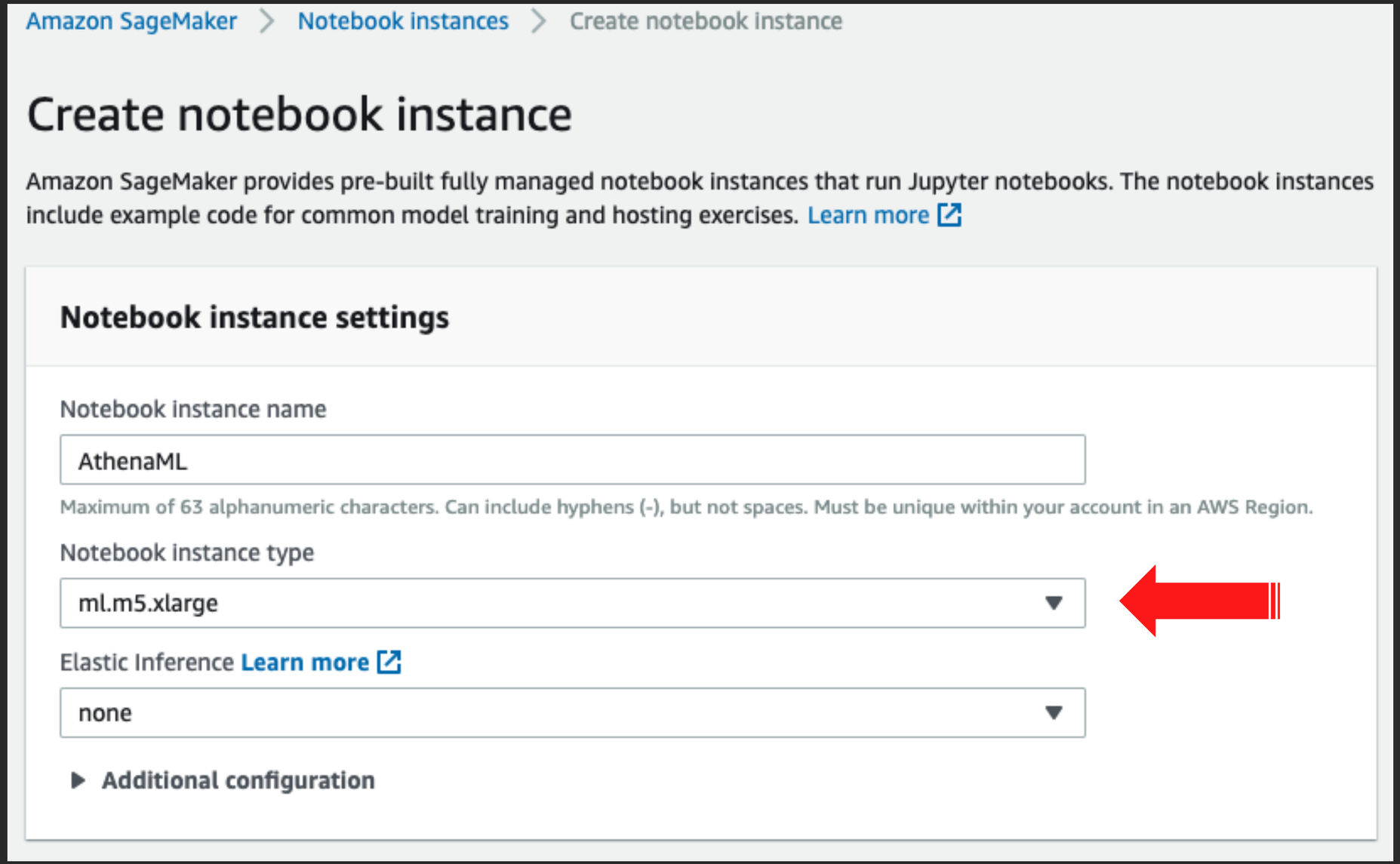
Step 1: Create Amazon SageMaker role

Lets see how we did

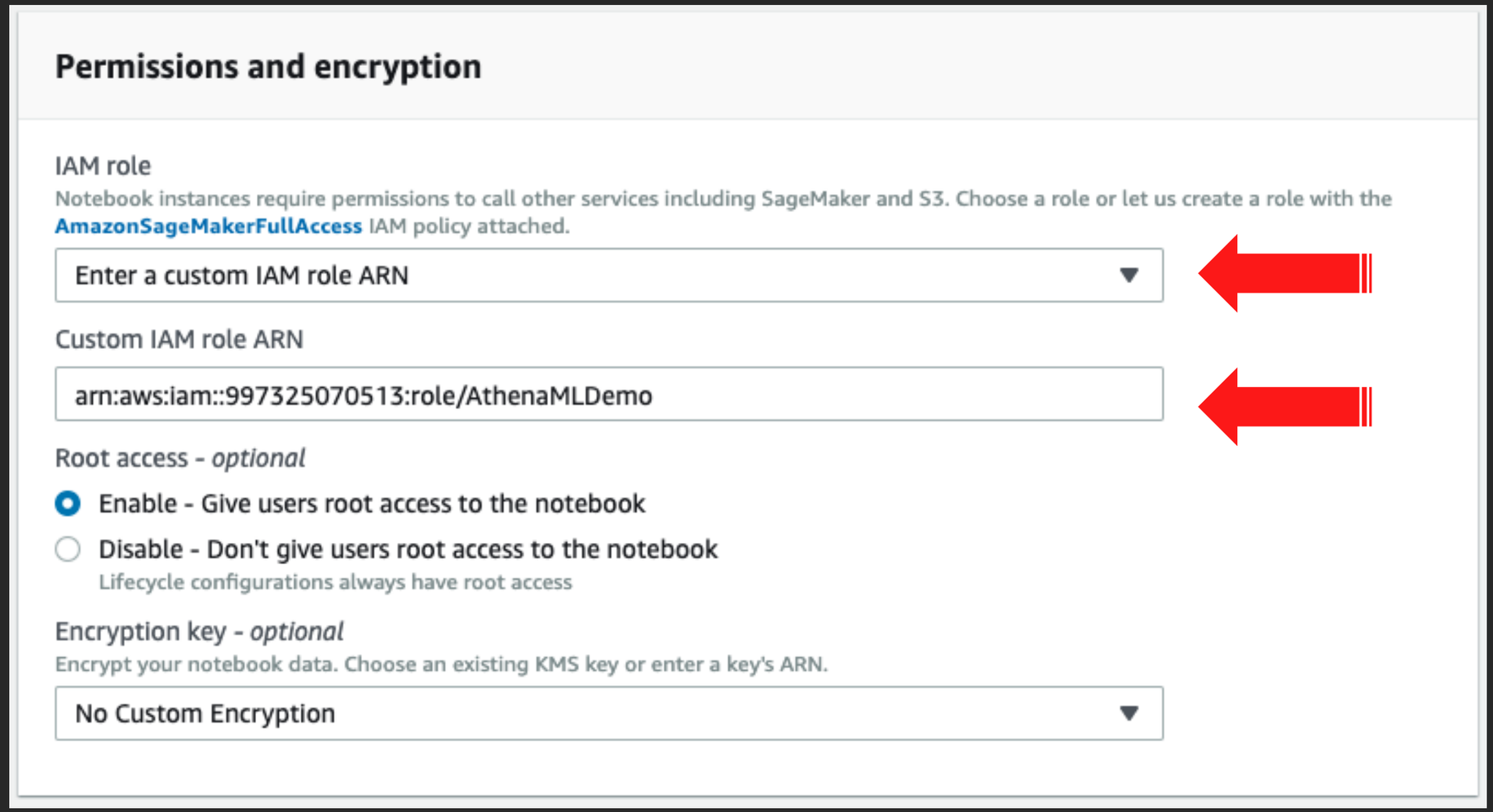
Step 2: Create Amazon SageMaker notebook



Step 2: Create Amazon SageMaker notebook



Step 2: Create Amazon SageMaker notebook

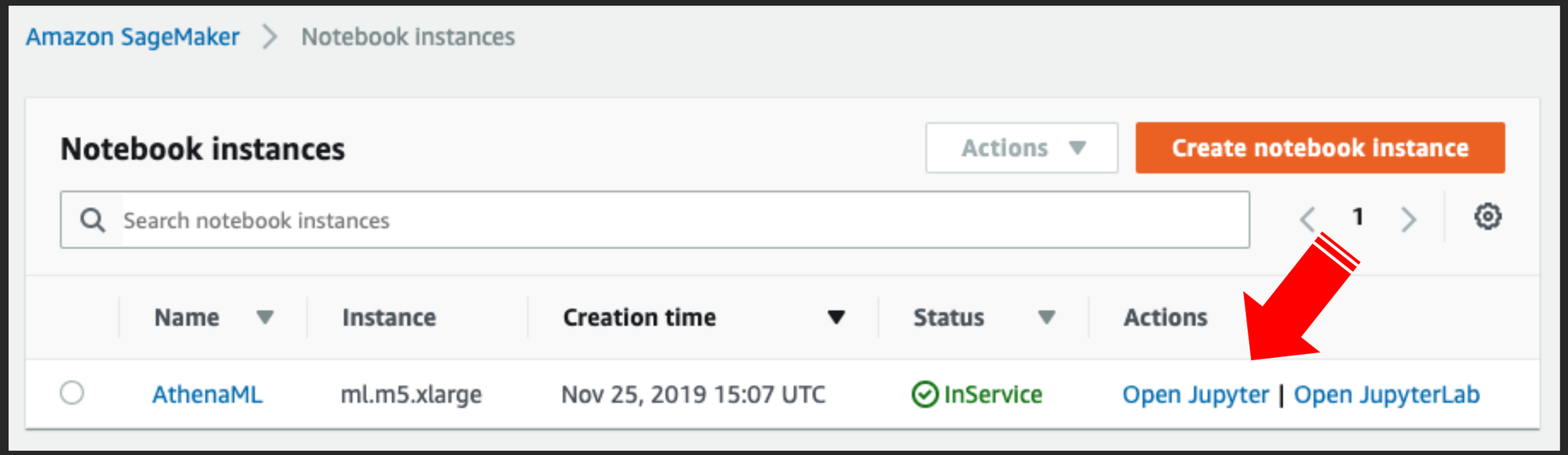


Step 2: Create Amazon SageMaker notebook

Lets see how we did

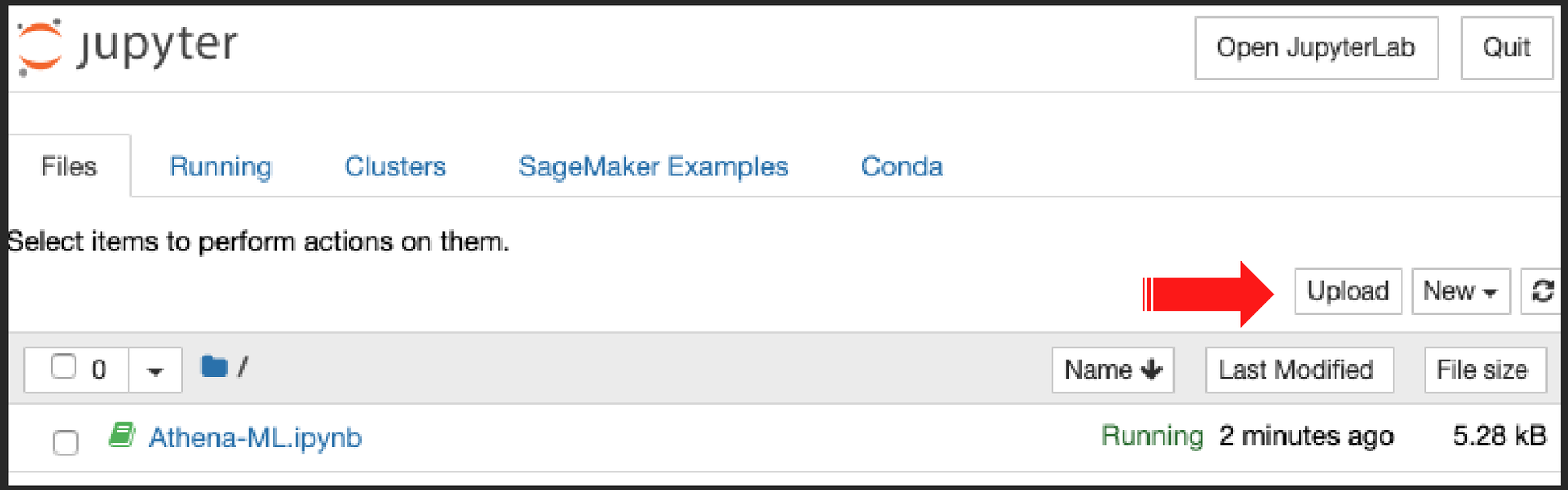
Step 3: Preparing an Amazon SageMaker model

- From the Amazon S3 console, create an S3 bucket for our data
- From the Amazon SageMaker console, open our Amazon SageMaker Jupyter notebook instance



Step 3: Preparing an Amazon SageMaker model

- Upload the example notebook file from the GitHub Wiki



Step 3: Preparing an Amazon SageMaker model

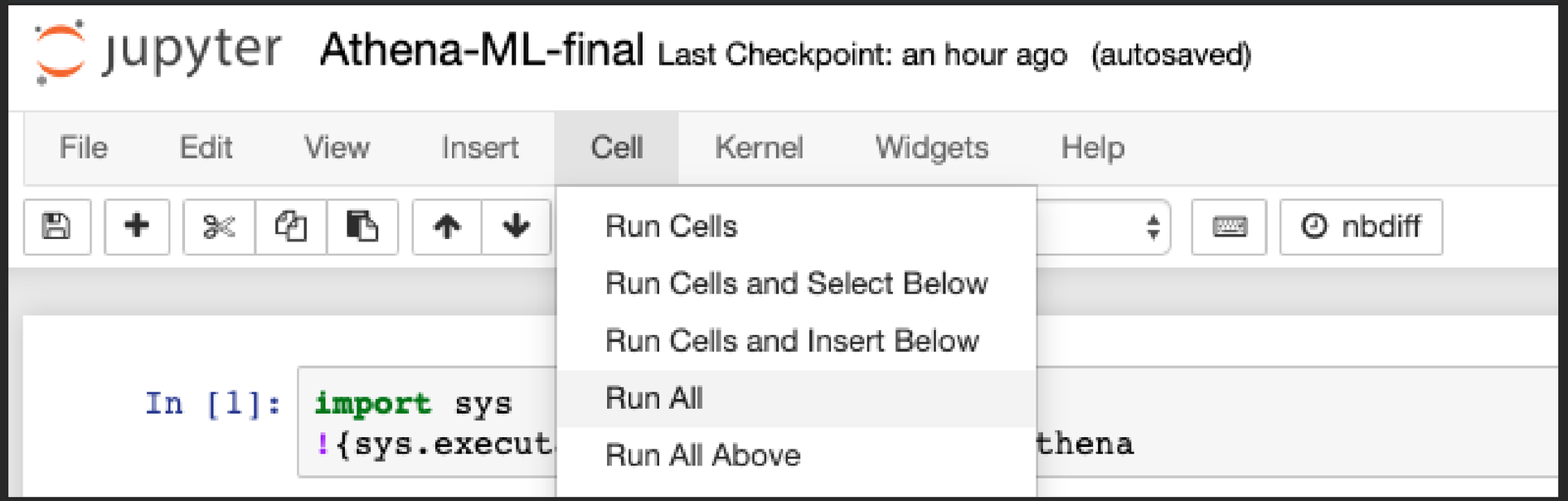
- Update the Amazon S3 location used by the notebook

```
from pyathena import connect
import pandas as pd
import sagemaker
import matplotlib.pyplot as plt

## TODO: change the bucket and prefix variable to point to an S3 bucket and path to use for the model output
bucket = 'athena-federation-test'
output_location = 's3://' + bucket + '/athena-ml/'
connection = connect(s3_staging_dir=output_location, region_name='us-east-1')
```

Step 3: Preparing an Amazon SageMaker model

- Run the notebook



Step 3: Preparing an Amazon SageMaker model

Let’s see how we did

Step 4: Run a query!

From the Athena console, run the below query from the **AmazonAthenaPreviewFunctionality** workgroup

```
USING FUNCTION detect_anomaly(b INT) RETURNS DOUBLE
                        TYPE SAGEMAKER_INVOKE_ENDPOINT
WITH (sagemaker_endpoint = '<ENDPOINT_NAME>')
SELECT detect_anomaly(number),
        time,
        number
FROM taxi_ridership_data limit 10;
```

Step 4: Run a query!

Let’s see how we did

Recap

- 1-click deploy-then-query 14 sources today in preview!
- Add custom data sources by writing 4 methods
- Add custom UDFs by writing 1 method
- Use Amazon SageMaker inference from an Athena query

Thank you!

AWS re:Invent

© 2019, Amazon Web Services, Inc. or its affiliates. All rights reserved.

aws



Please complete the session survey in the mobile app.

© 2019, Amazon Web Services, Inc. or its affiliates. All rights reserved.