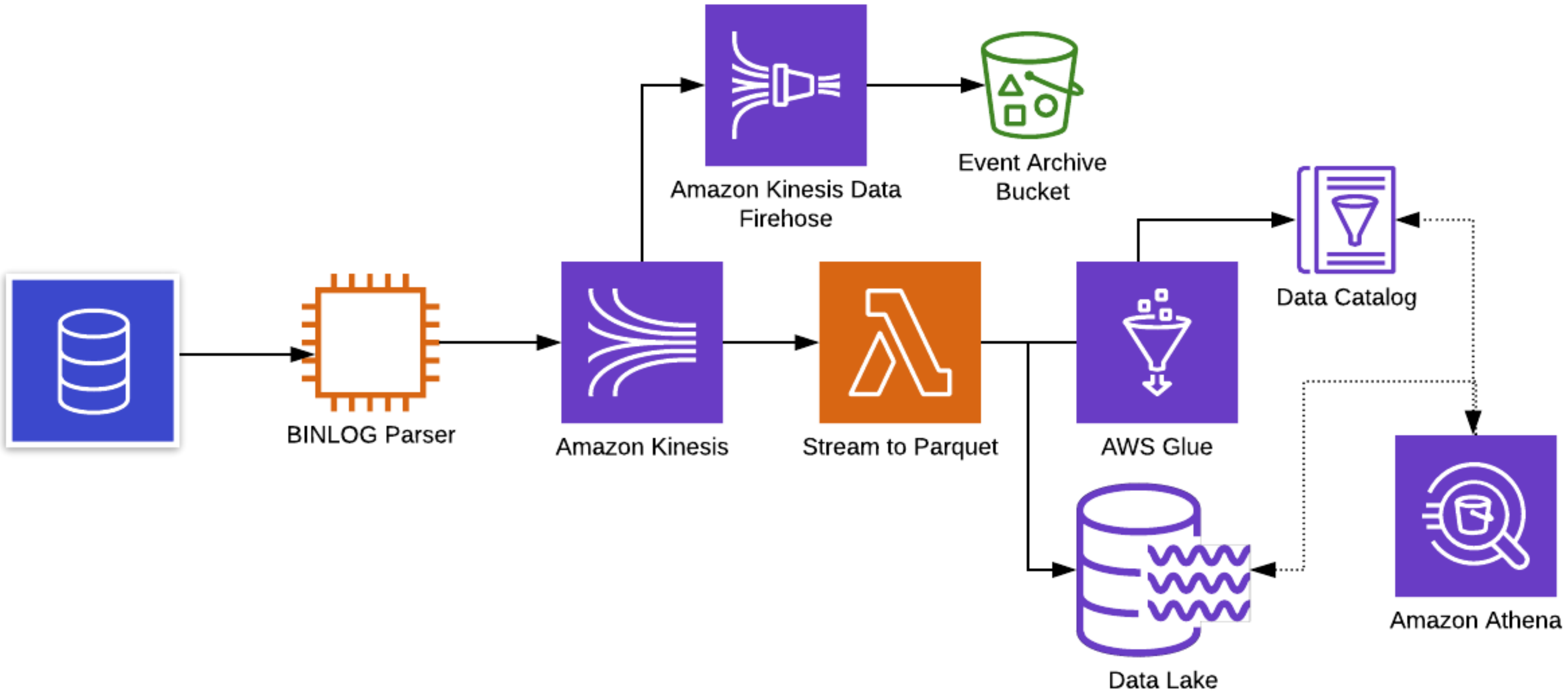
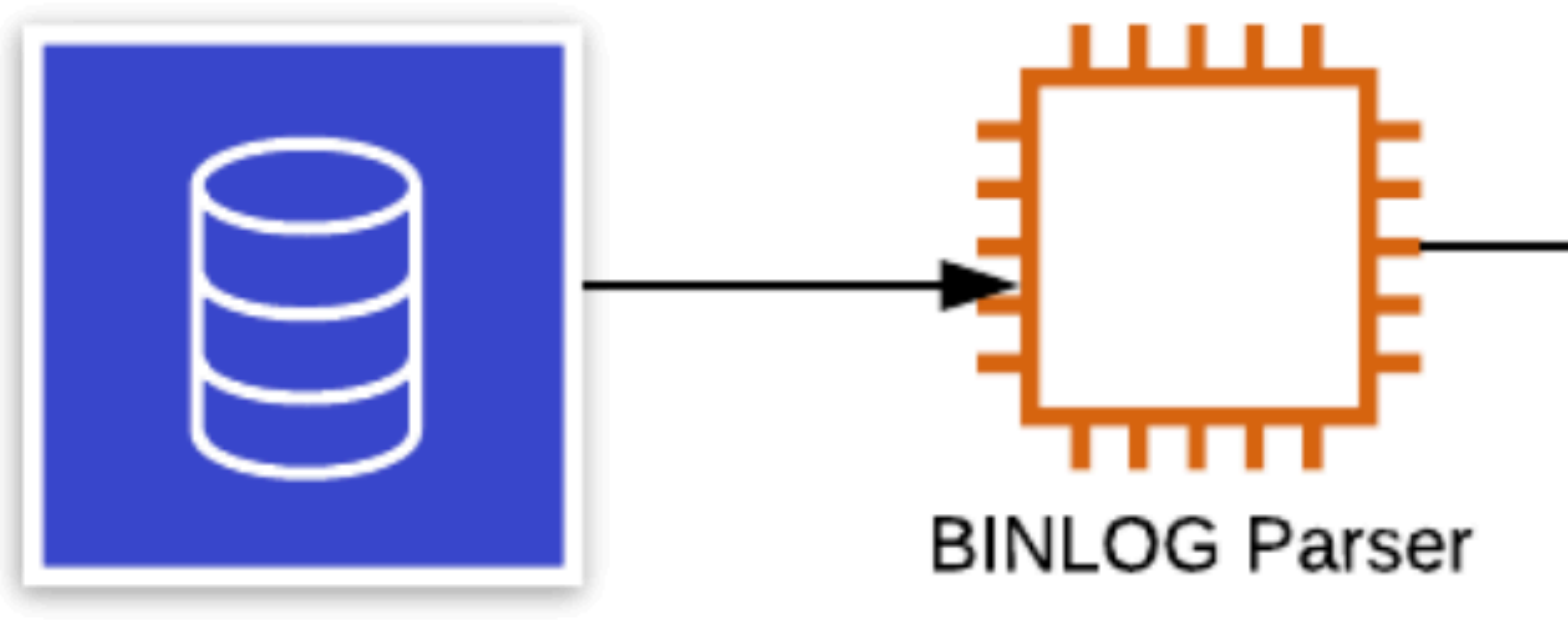
Building History from MySQL

IT'S BEEN A ROUGH WEEK...
HERE'S A BABY SLOTH IN PJ'S.
imgflip.com



Amazon Kinesis Data Firehose
Event Archive Bucket
Data Catalog
BINLOG Parser
Amazon Kinesis
Stream to Parquet
AWS Glue
Amazon Athena
Data Lake


BINLOG Parser

BIN LOG

- `pip install mysql-replication`

-
  ```
  GRANT REPLICATION SLAVE, REPLICATION CLIENT, SELECT
  ON *.* TO 'username'@'host'
  ```

```
MYSQL_SETTINGS = {
    "host": "mysql-server.some-domain.com",
    "port": 3306,
    "user": 'username',
    "passwd": 'password'
}
```

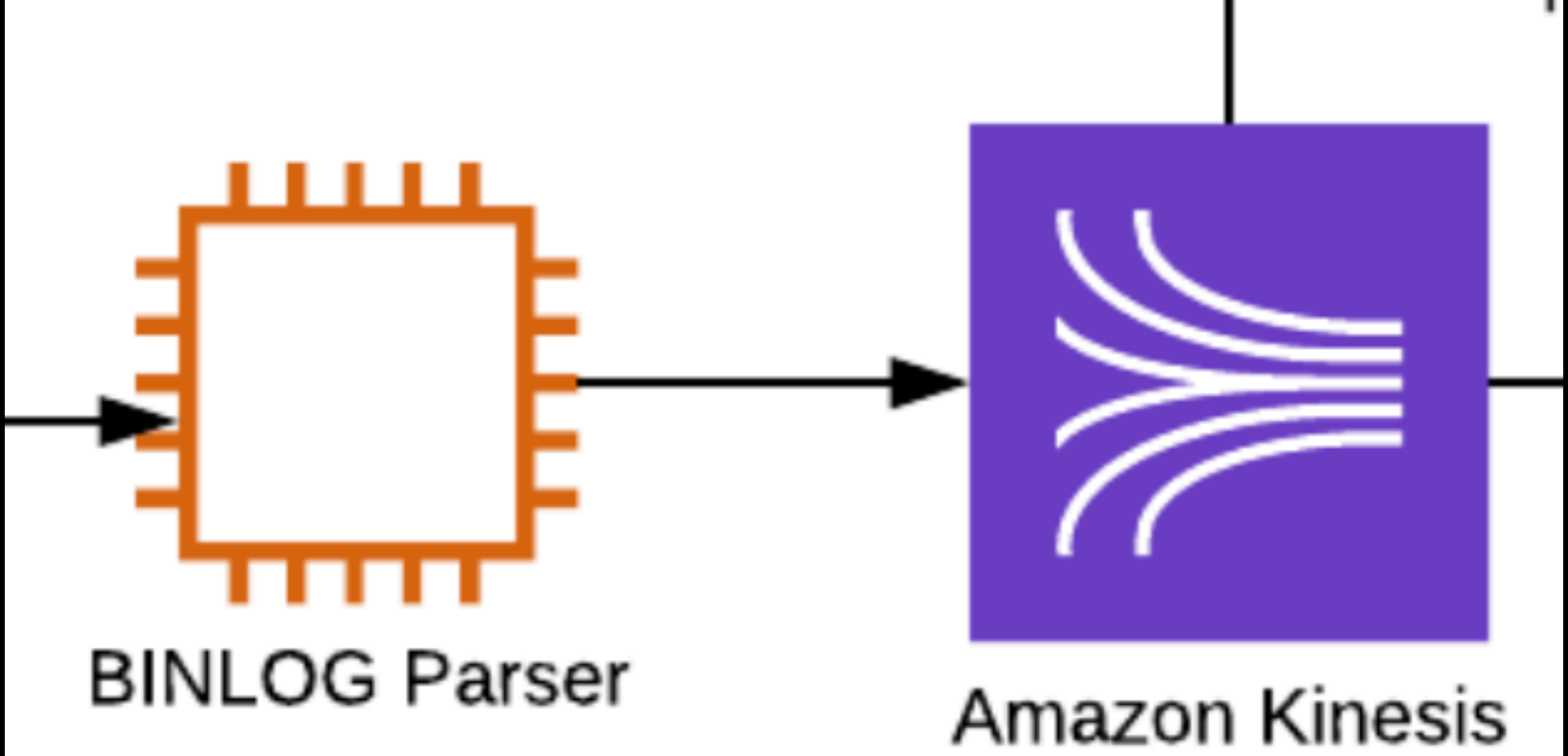
```python
def main():
    # server_id is your slave identifier, it should be unique.
    # set blocking to True if you want to block and wait for the next event at
    # the end of the stream
    stream = BinLogStreamReader(
        connection_settings=MYSQL_SETTINGS,
        server_id=1337,
        blocking=True,
        only_events=[
            DeleteRowsEvent,
            WriteRowsEvent,
            UpdateRowsEvent
        ]
    )

    for binlogevent in stream:
        binlogevent.dump()

    stream.close()
```

```
=== TableMapEvent ===
Date: 2020-02-18T14:27:58
Log position: 28649
Event size: 65
Read bytes: 63
Table id: 91221
Schema: staging_api
Table: activity_feed
Columns: 11
```

```
=== WriteRowsEvent ===
Date: 2020-02-18T14:27:58
Log position: 28761
Event size: 89
Read bytes: 13
Table: staging_api.activity_feed
Affected columns: 11
Changed rows: 1
Values:
--
* activity_feed_id : 2630701
* loan_id : None
* crud : read
* model : Loans/LoanDraft
* primary_key : 44206
* user_role : branchAdministrator
* is_deleted : 0
* created_by : 15016
* created_at : 2020-02-18 20:27:58.992000
* updated_by : 15016
* updated_at : 2020-02-18 20:27:58.992000
```

```
=== QueryEvent ===
Date: 2020-02-18T14:28:13
Log position: 28947
Event size: 67
Read bytes: 67
Schema: b'staging_api'
Execution time: 0
Query: BEGIN
```

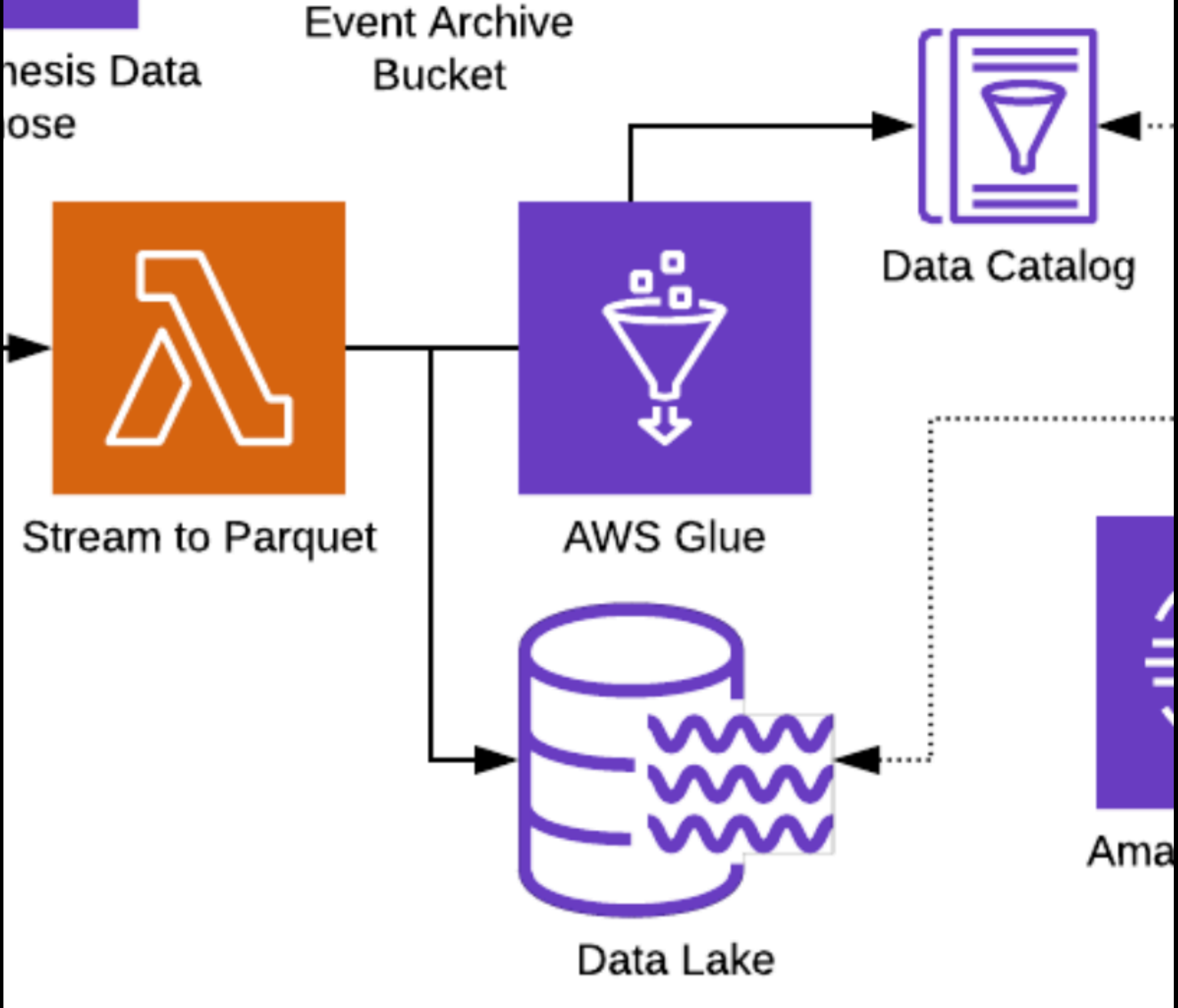
```
=== UpdateRowsEvent ===
Date: 2020-02-18T14:28:42
Log position: 30127
Event size: 293
Read bytes: 23
Table: staging_api.inspections
Affected columns: 41
Changed rows: 1
Affected columns: 41
Values:
--
*inspection_id:23151=>23151
*is_deleted:0=>0
*created_by:12396=>12396
*created_at:2020-02-17 20:29:32.225000=>2020-02-17 20:29:32.225000
*updated_by:12396=>12396
*updated_at:2020-02-17 20:29:32.277000=>2020-02-18 20:28:42.760000
*field_inspector_id:None=>None
```


BINLOG Parser
Amazon Kinesis

```python
@dataclass
class ServiceEvent:
    source: str
    type: str
    data: dict
    aws_request_id: str
    version: int = field(default=1)
    event_date: str = field(default_factory=lambda: datetime.utcnow().isoformat())
    uuid: str = field(default_factory=lambda: str(uuid4()))
```

```python
event_list.append(
    ServiceEvent(
        source=type(elem).__name__.lower(),
        type='create',
        data=row2dict(elem),
        aws_request_id='test_id',
        user_id=user_id,
        version=2,
    )
)
```

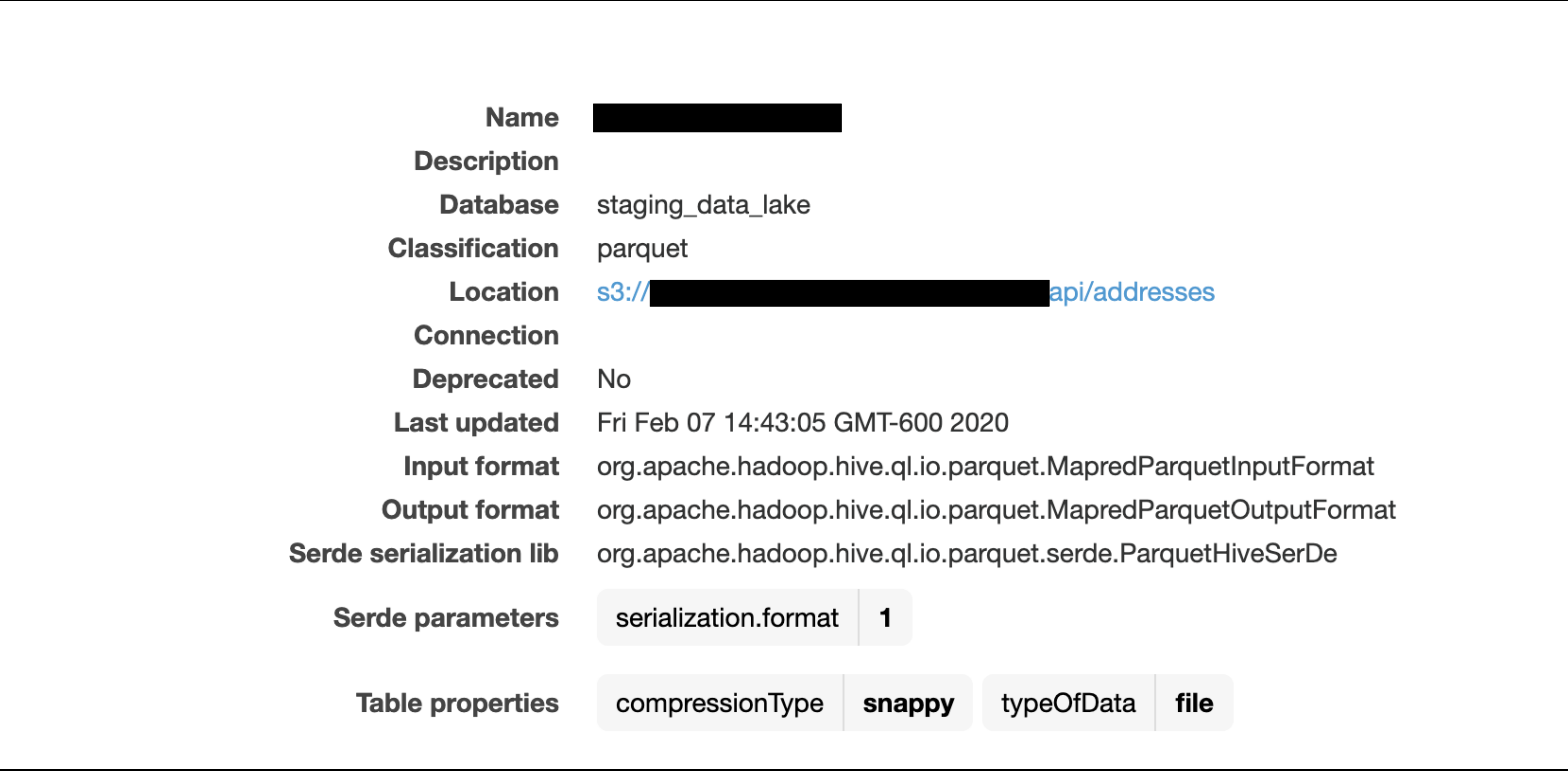
```python
for event in event_list:
    records_list.append(
        {"Data": event.to_json(),
        "PartitionKey": partition_key
        }
    )

for records_list in chunk_records(records, 500):
    try:
        kinesis_client.put_records(
            Records=records_list,
            StreamName=environ.get("KINESIS_STREAM")
        )
```


nesis Data
ose
Event Archive
Bucket
Data Catalog
Stream to Parquet
AWS Glue
Data Lake
Ama

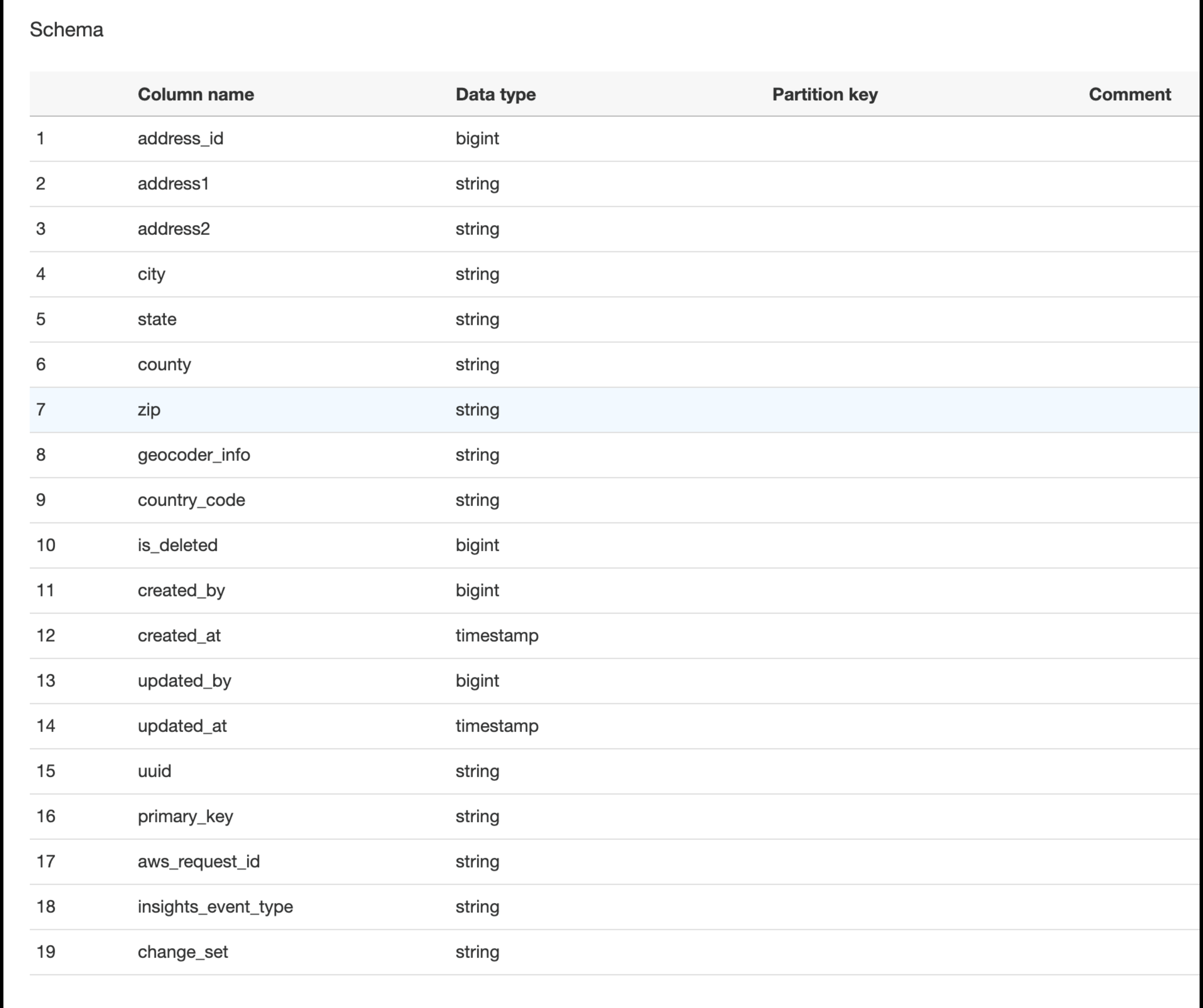
AWS Data Wrangler

- `pip install awswrangler`
- `pip install fastparquet`

```python
database_name = f'data_lake'
table_name = f'{source}_{data_model}'
session = awswrangler.Session(region_name='us-east-1')
session.pandas.to_parquet(
    dataframe=event_df,
    database=database_name,
    table=table_name,
    cast_columns=cast_dict_parquet,
    path=s3_path,
    preserve_index=False,
    inplace=True,
)
```

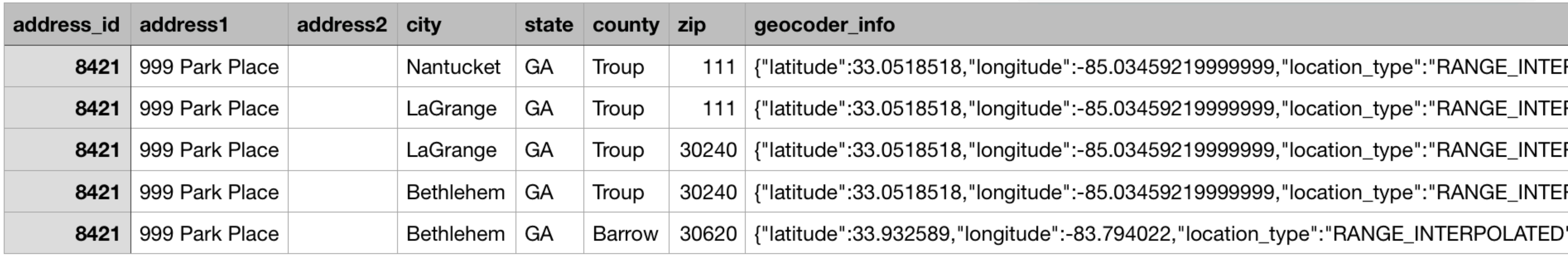


Name	[redacted]
Description	
Database	staging_data_lake
Classification	parquet
Location	s3://[redacted]api/addresses
Connection	
Deprecated	No
Last updated	Fri Feb 07 14:43:05 GMT-600 2020
Input format	org.apache.hadoop.hive.ql.io.parquet.MapredParquetInputFormat
Output format	org.apache.hadoop.hive.ql.io.parquet.MapredParquetOutputFormat
Serde serialization lib	org.apache.hadoop.hive.ql.io.parquet.serde.ParquetHiveSerDe
Serde parameters	serialization.format **1**
Table properties	compressionType **snappy** typeOfData **file**



Schema

	Column name	Data type	Partition key	Comment
1	address_id	bigint		
2	address1	string		
3	address2	string		
4	city	string		
5	state	string		
6	county	string		
7	zip	string		
8	geocoder_info	string		
9	country_code	string		
10	is_deleted	bigint		
11	created_by	bigint		
12	created_at	timestamp		
13	updated_by	bigint		
14	updated_at	timestamp		
15	uuid	string		
16	primary_key	string		
17	aws_request_id	string		
18	insights_event_type	string		
19	change_set	string		



address_id	address1	address2	city	state	county	zip	geocoder_info
8421	999 Park Place		Nantucket	GA	Troup	111	{"latitude":33.0518518,"longitude":-85.03459219999999,"location_type":"RANGE_INTER
8421	999 Park Place		LaGrange	GA	Troup	111	{"latitude":33.0518518,"longitude":-85.03459219999999,"location_type":"RANGE_INTER
8421	999 Park Place		LaGrange	GA	Troup	30240	{"latitude":33.0518518,"longitude":-85.03459219999999,"location_type":"RANGE_INTER
8421	999 Park Place		Bethlehem	GA	Troup	30240	{"latitude":33.0518518,"longitude":-85.03459219999999,"location_type":"RANGE_INTER
8421	999 Park Place		Bethlehem	GA	Barrow	30620	{"latitude":33.932589,"longitude":-83.794022,"location_type":"RANGE_INTERPOLATED"

Oh yeah… 💩 just got real

Schema Migrations

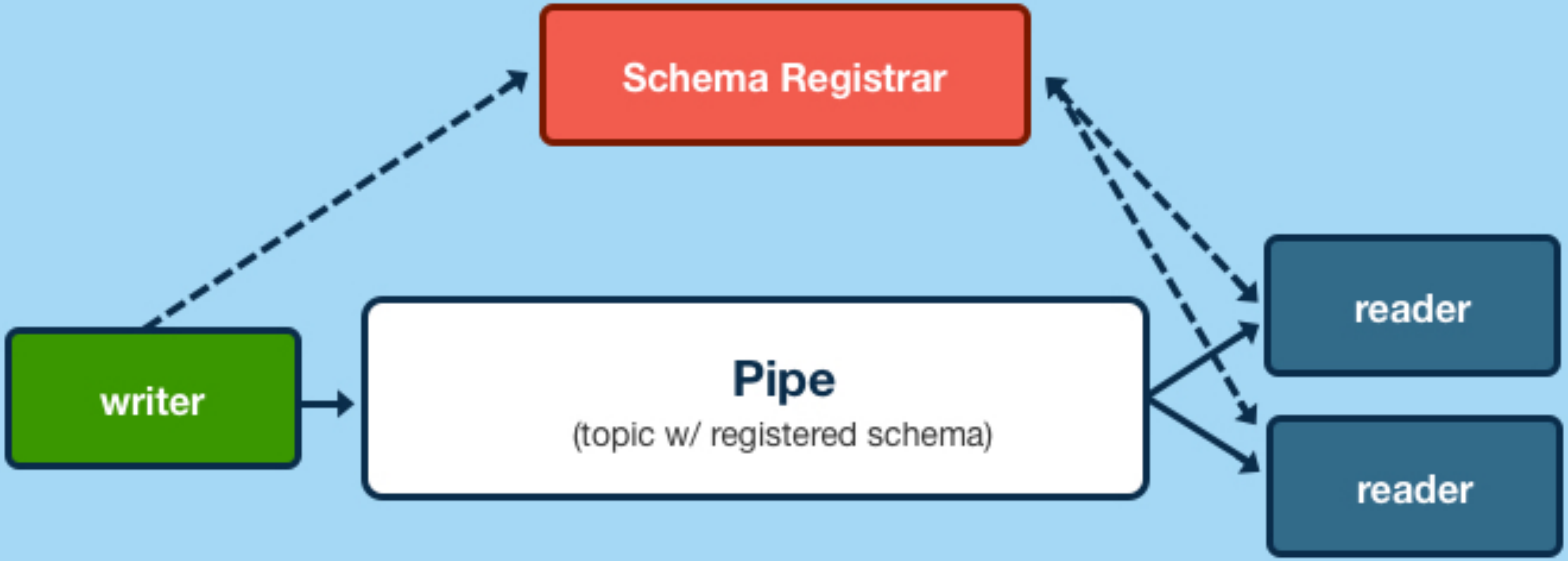
- Default all new data to strings, and gradually fix types
- Write the schema to Kinesis (Apache AVRO, ProtoBufs)

```python
def build_data_dict(tables: List, db_meta) -> Dict:
    data_dict: Dict = {"athena": {}, "python": {}}
    for table in tables:
        for column in db_meta.tables[table].c:
            if "TINYINT" in str(column.type):
                data_dict["athena"][column.name] = "bigint"
                data_dict["python"][column.name] = "int"
                continue
            if column.name == "primary_key":
                data_dict["athena"][column.name] = "string"
                data_dict["python"][column.name] = "str"
                continue
            try:
                data_dict["athena"][column.name] = python2athena(
                    column.type.python_type
                )
                data_dict["python"][column.name] = sql2python(column.type.python_type)
            except TypeError as exc_info:
                LOGGER.error(
                    "Failed to determine type for %s with message: %s",
                    column.name,
                    str(exc_info),
TYPE_MAP = {
    "<class 'int'>": {"athena": "bigint", "python": "int"},
    "<class 'float'>": {"athena": "double", "python": "float"},
    "<class 'decimal.Decimal'>": {"athena": "double", "python": "float"},
    "<class 'bool'>": {"athena": "bigint", "python": "int"},
    "<class 'str'>": {"athena": "string", "python": "str"},
    "<class 'datetime.datetime'>": {
        "athena": "timestamp",
        "python": "datetime.datetime",
    },
    "<class 'datetime.date'>": {"athena": "date", "python": "datetime.datetime"},
    "<class 'bytes'>": {"athena": "bytes", "python": "bytes"},
    "<class 'dict'>": {"athena": "string", "python": "str"},
}
```

```python
def python2athena(python_type: type) -> str:
    python_type_str: str = str(python_type)
    if python_type_str in TYPE_MAP:
        return TYPE_MAP[python_type_str]["athena"]

    raise TypeError(f"Unsupported Athena type: {python_type_str}")
```

```python
s3_fs = s3fs.S3FileSystem()
file_accum = []
for file_name in files:
    table = parquet.read_table(f"s3://{bucket_name}/{file_name}", filesystem=s3_fs)
    table_data_frame = table.to_pandas()
    file_accum.append(table_data_frame)
data_frame = pandas.concat(
    file_accum, keys=range(1, len(file_accum) + 1), sort=False
).reset_index(level=1, drop=True)
return data_frame
```

```python
wrangler = awswrangler.Session()
group_files = wrangler.pandas.to_parquet(
    dataframe=data_frame,
    database=data_lake,
    table=f"{table_source.replace('-', '_')}_{table_name.replace('-', '_')}",
    preserve_index=False,
    mode='append',
    cast_columns=cast_types,
    procs_cpu_bound=1,
    procs_io_bound=1,
    inplace=True,
    path=f"s3://{bucket_name}/{table_source}/{table_name}",
)
```


Schema Registrar
writer
Pipe
(topic w/ registered schema)
reader
reader

- It is an error if the two schemas do not *match*.
  To match, one of the following must hold:
  - both schemas are arrays whose item types match
  - both schemas are maps whose value types match
  - both schemas are enums whose (unqualified) names match
  - both schemas are fixed whose sizes and (unqualified) names match
  - both schemas are records with the same (unqualified) name
  - either schema is a union
  - both schemas have same primitive type
  - the writer's schema may be *promoted* to the reader's as follows:
    - int is promotable to long, float, or double
    - long is promotable to float or double
    - float is promotable to double
    - string is promotable to bytes
    - bytes is promotable to string
- **if both are records:**
  - the ordering of fields may be different: fields are matched by name.
  - schemas for fields with the same name in both records are resolved recursively.
  - if the writer's record contains a field with a name not present in the reader's record, the writer's value for that field is ignored.
  - if the reader's record schema has a field that contains a default value, and writer's schema does not have a field with the same name, th
    value from its field.
  - if the reader's record schema has a field with no default value, and writer's schema does not have a field with the same name, an error i
- **if both are enums:**
  if the writer's symbol is not present in the reader's enum and the reader has a `default` value, then that value is used, otherwise an error is
- **if both are arrays:**
  This resolution algorithm is applied recursively to the reader's and writer's array item schemas.
- **if both are maps:**
  This resolution algorithm is applied recursively to the reader's and writer's value schemas.
- **if both are unions:**
  The first schema in the reader's union that matches the selected writer's union schema is recursively resolved against it. if none match, an er
- **if reader's is a union, but writer's is not**
  The first schema in the reader's union that matches the writer's schema is recursively resolved against it. If none match, an error is signalled.
- **if writer's is a union, but reader's is not**
  If the reader's schema matches the selected writer's schema, it is recursively resolved against it. If they do not match, an error is signalled.

Short Version of AVRO + JAM RULE

- Don't rename columns
- Don't change column types
- Don't repurpose a column or a feature flag (Knight Capital Group)

Thank you