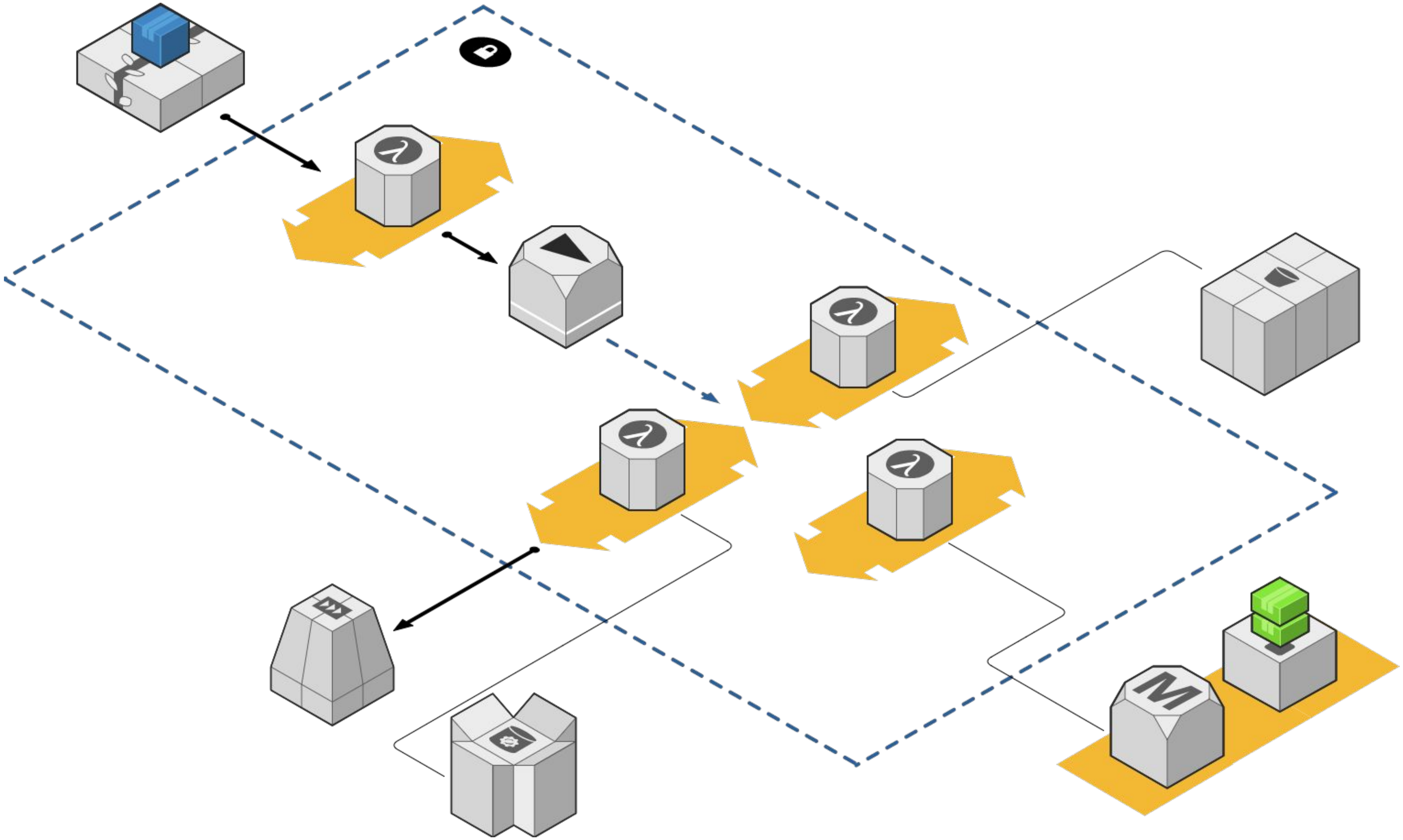
VERY FUNCTIONAL PYTHON

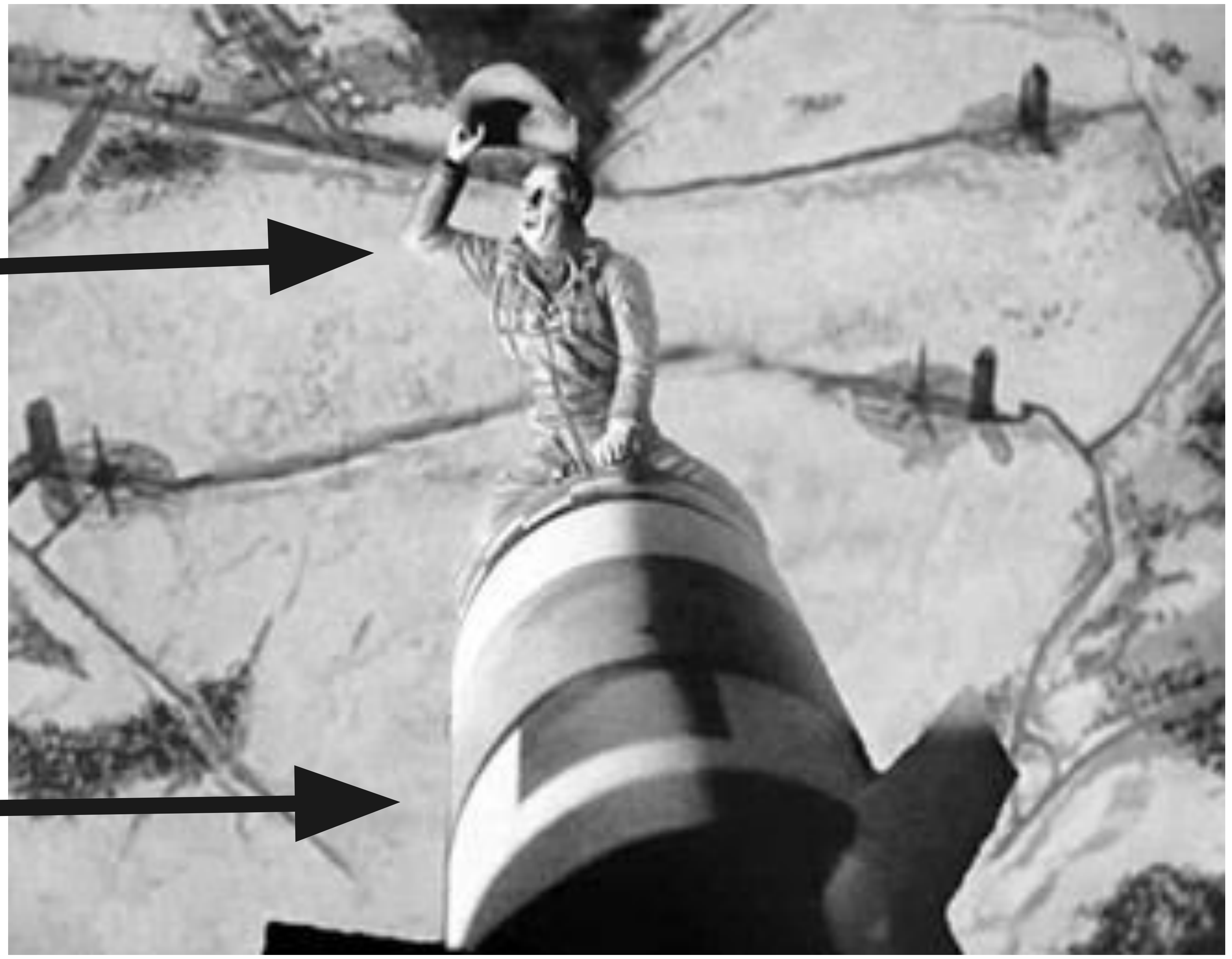
(IN FUNCTIONS (AS A SERVICE))

MY RELATIONSHIP WITH MUTABLE STATE



λ
λ
λ
λ
M

LEFT
EXIT 12
MAKING REASONED DECISIONS
SERVERLESS
ME
imgflip.com



ME

SERVERLESS

A CLOJURE EXAMPLE

```clojure
(->> (get-records-from event)
     transform-records
     assemble-records
     insert-records)
```

We get a list of records and then it’s just a set of transforms over a collection. Conceptually, this is a composition.

A CLOJURE EXAMPLE (THIS TIME IT’S PERSONAL)

```clojure
(def coll [1 2 3 4])

(defn byTwo [i]
 (* i 2))

(transduce (comp (map inc)
                 (map byTwo))
           conj
           coll)

;=> [4 6 8 10]
```

By changing our operation to be over a scalar item, we can compose transforms

A CLOJURE EXAMPLE (WITH A VENGEANCE)

```
(->> (get-records-from event)
     (map (comp insert-record
                assemble-record
                transform-record)))
```

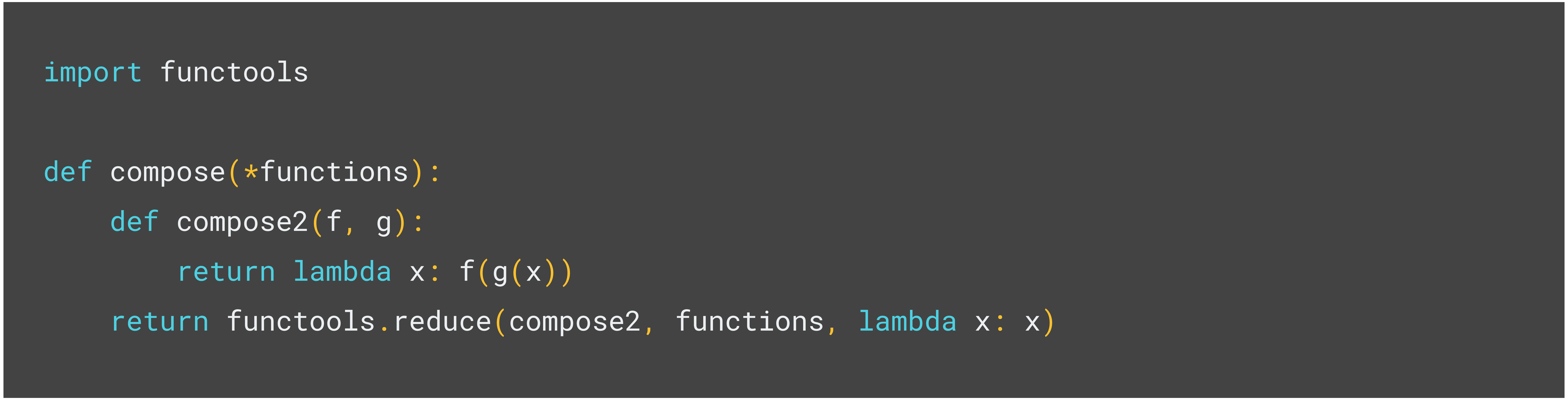
If we don’t need a fold operation, we can compose scalar operations within a map

HEY! HE USES LISP!
SEE? NOBODY CARES.
imgflip.com

A FIRST TRY

```
def error_sink(event, context):
  records = event['Records']
  conn, cur = connect_to_rds()
  json_records = tuple(map(unpack_record, records))
  try:
    cur.executemany("""INSERT INTO errorevents(id,path,status,event_created_at) VALUES (%(id)s, %(path)s,
    %(status)s, %(event_created_at)s)""", json_records)
    conn.commit()
  except Exception as e:
    logger.error("Lambda: EXCEPTION in error_sink: " + str(e))
  finally:
    cur.close()
```

USING HIGHER-ORDER COMPOSITION

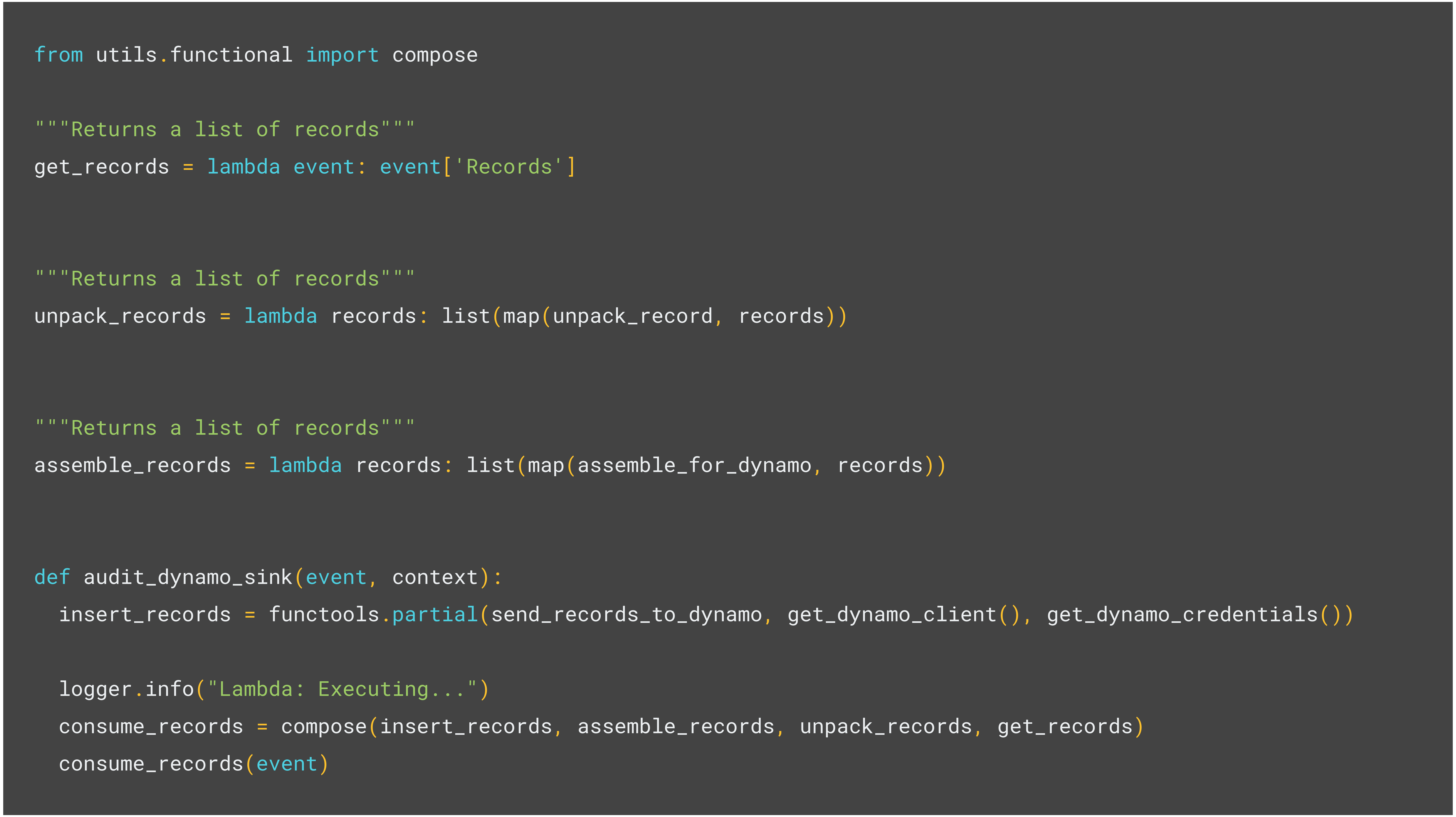


```
import functools

def compose(*functions):
    def compose2(f, g):
        return lambda x: f(g(x))
    return functools.reduce(compose2, functions, lambda x: x)
```

Simple composition of multiple functions

me: *slaps roof
of aws lambda function*
this bad boy can fit so
much functional composition in it
imgflip.com



```python
from utils.functional import compose

"""Returns a list of records"""
get_records = lambda event: event['Records']


"""Returns a list of records"""
unpack_records = lambda records: list(map(unpack_record, records))


"""Returns a list of records"""
assemble_records = lambda records: list(map(assemble_for_dynamo, records))


def audit_dynamo_sink(event, context):
  insert_records = functools.partial(send_records_to_dynamo, get_dynamo_client(), get_dynamo_credentials())

  logger.info("Lambda: Executing...")
  consume_records = compose(insert_records, assemble_records, unpack_records, get_records)
  consume_records(event)
```

ALL OF OUR LOGIC IS COMPOSED
WE NEED TO GO DEEPER
imgflip.com

JSON SCHEMA

```
{
  "payload": {
    "field1": 1,
    "field2": "foo"
    "optional_properties": {}
  },
  "errors": []
}
```

Now we have an implicit type we can reason about throughout our system.

Sorry, maths

```
pipeline :: a -> a
```

Where arrows are morphisms

Sorry, maths II

```
pipeline :: a -> a -> b
pipeline :: a -> a -> c
            ● -> ● -> ●
                   -> ●
```

*A pipeline is a directed graph, with one **producer** per **entity***

Sorry, maths III

```
pipeline :: a -> a -> b
pipeline :: a -> a -> c
            ● -> ● -> ●
                   -> ●
```

In this example ***here*** *is the service (Kinesis, SQS, DynamoDB, Kafka, Cassandra), and subsequent morphisms could be in separate lambdas or transformations within one - the guiding rule is now how they compose, and how/where they will be (re)used*

Meta composability

```
pipeline :: a -> a -> a -> a

pipeline :: a -> a -> a -> b

pipeline :: a -> a -> b -> c
```

This means that business logic is composable between functions (and partial application is possible)

LINKS

OFFICIAL DOCS

https://docs.python.org/3.1/howto/functional.html

ROLL YOUR OWN

https://mathieularose.com/function-composition-in-python/

THANKS!

@hipsters_unite