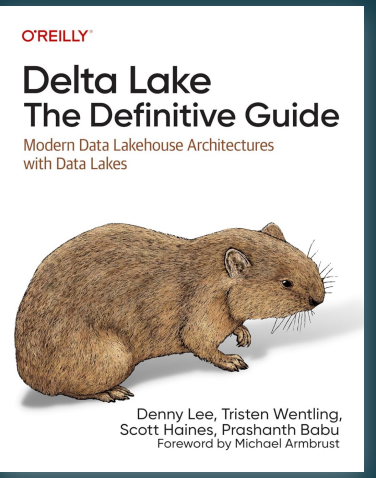

The Hitchhiker's Guide to Delta Lake Streaming
databricks
DATA+AI SUMMIT
In an Agentic Universe
Scott Haines
June 10th, 2025

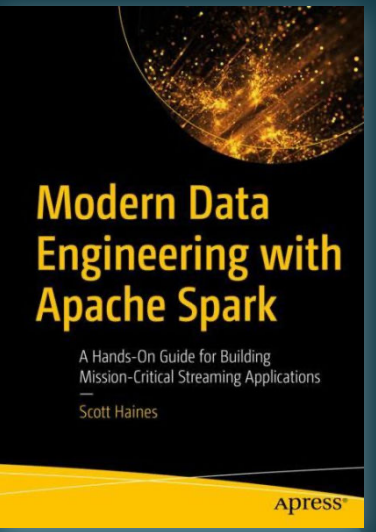
Piloting the Session:

Scott Haines

"A Little About Me"

- I'm a Distinguished Software Engineer at Nike.
- I enjoy teaching and mentoring, writing books, and learning about new and exciting things.
- I've been working on massive high-performance streaming systems for the better part of the last 15 years.





Preparing for the Journey

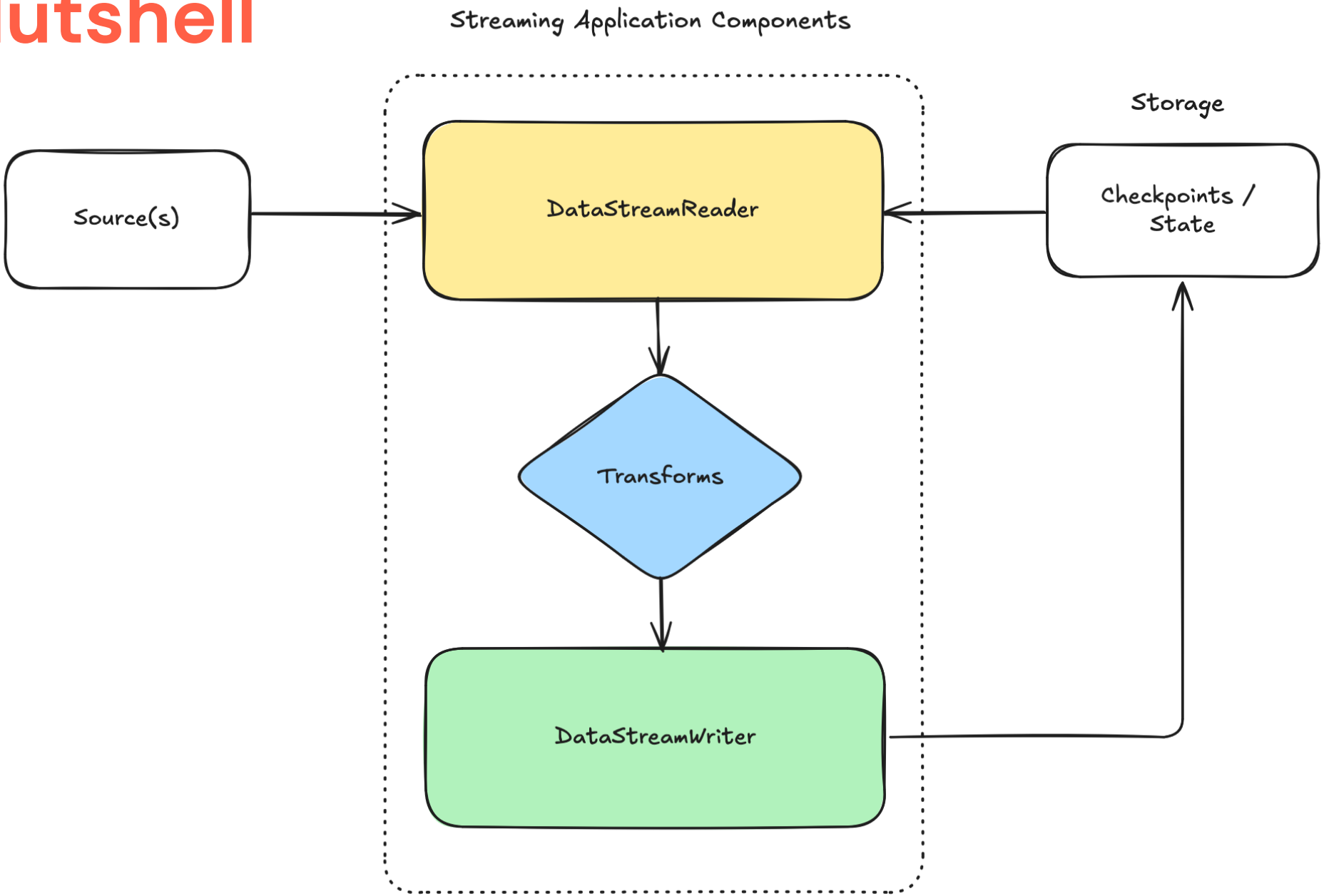
Mission Prep

1. Writing “Generic Data Applications” as “Components”.
2. Using Agentic Workflow Augmentation.
3. Q/A or High Fives

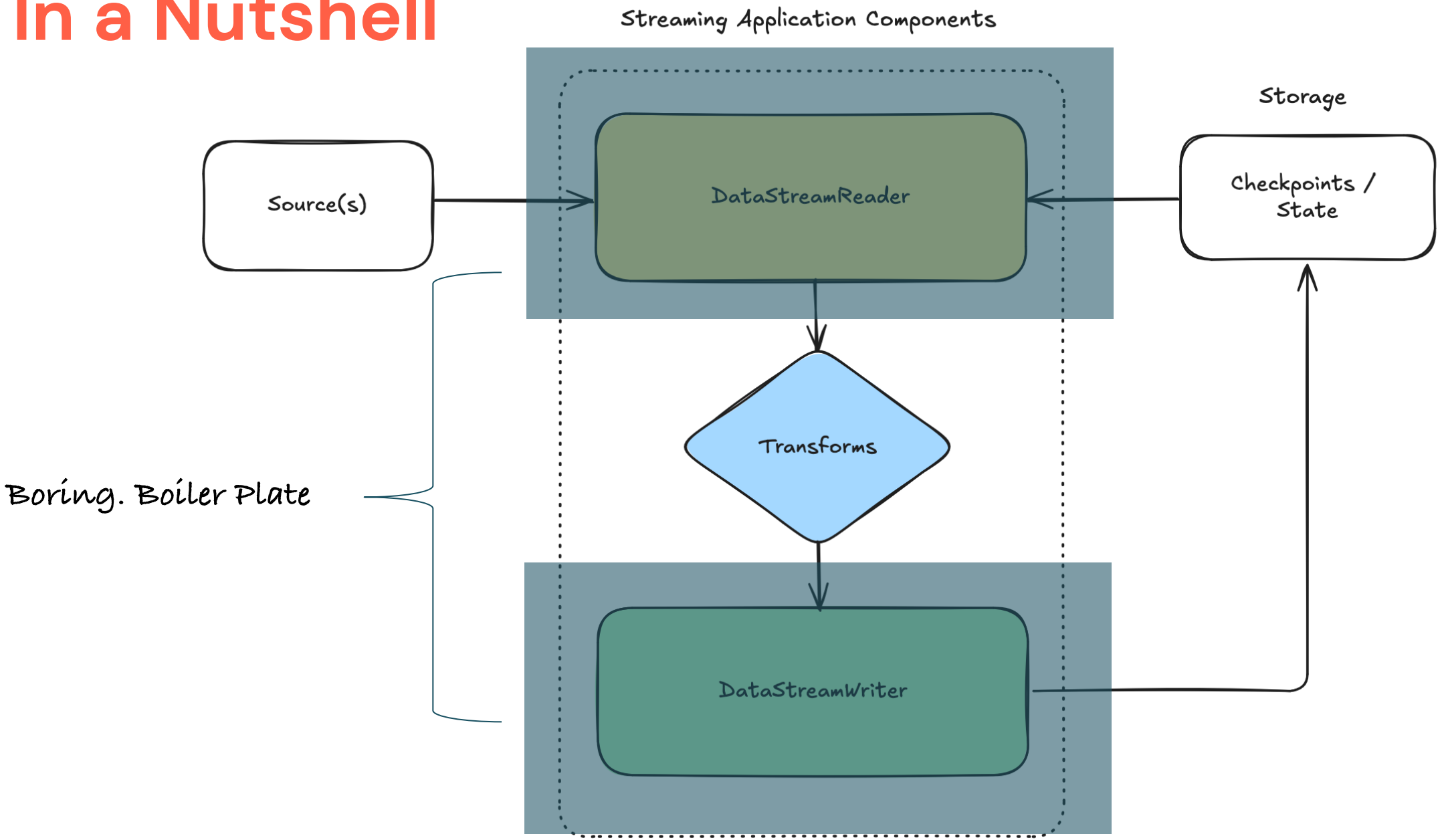
On Writing Generic Data Applications as Components

Don't Panic. This part is simple ☺

In a Nutshell



In a Nutshell



Base Application Pattern

Provides Sane Defaults and Common Ways of Working

Roles and Responsibilities

- Provide *Sane Defaults* and Common, Standardized Functionality.
- Establish a Common Way of Building and Tearing down applications.
- **Standardize where "checkpoints" and "state" live** — which comes in handy with greater automation.

```python
class StreamingApp(SparkLoggingProvider):
    spark_session: SparkSession

    app_name: str = "generic-streaming-app"
    app_version: str = "0.0.1"
    app_logging_prefix: str = "StreamingApp::genAI"
    app_checkpoint_path: Optional[str] = None
    app_checkpoint_version: Optional[str] = None

    def __init__(self, spark: Optional[SparkSession] = None):
        pass

    def generate_checkpoint_path(self) -> Path:
        pass

    def generate_read_stream(self, s_options: Dict[str, str]) -> DataStreamReader:
        pass

    @staticmethod
    def generate_write_stream(
            df: DataFrame,
            s_options: Dict[str, str]) -> DataStreamWriter:
        pass
```

Pass: used to fit on slide. Imagine what is needed ^^

Base Application Pattern

Provides Sane Defaults and Common Ways of Working

Roles and Responsibilities

- Provide *Sane Defaults* and Common, Standardized Functionality.
- Establish a Common Way of Building and Tearing down applications.
- **Standardize where "checkpoints" and "state" live** — which comes in handy with greater automation.
- *Yes. This is config-driven development…*

```
fresh_session = self.spark_session.newSession()
fresh_session.conf.set('spark.app.name',
                       'delta_to_delta_app')
fresh_session.conf.set("spark.app.version", "3.0.0")
fresh_session.conf.set("spark.app.checkpoints.path",
                       "/Volumes/dev/app/fs/d2d/")
fresh_session.conf.set('spark.app.checkpoints.version',
                       'canary')

app = StreamingApp(spark=fresh_session).
```

```
app_name                                  StreamingApp
app_version                               StreamingApp
spark_session                             StreamingApp
app_checkpoints_path                      StreamingApp
app_logging_prefix                        StreamingApp
app_checkpoint_version                    StreamingApp
generate_checkpoint_location(self)        StreamingApp
generate_read_stream(self, s_options)     StreamingApp
generate_spark_session                    StreamingApp
generate_write_stream                     StreamingApp
logger                                    StreamingApp
Press ↵ to insert, ⇥ to replace  Next Tip
```

Data Stream Reading

Using Config-Driven Design Patterns

Roles and Responsibilities

- Must provide a mechanism to populate the correct configuration for a given streaming data source.
- This is a piece of cake for Delta Lake.
- *We'll look at the available options next.*

```
def generate_read_stream(self, s_options: Dict[str, str])
    """
    Generate the DataStreamReader. Note: This method can
    :param s_options: The dictionary of key/value pairs
    :return: The DataStreamReader object for composition
    """

    return self.spark_session.readStream.options(**s_optio
```

Delta Streaming Source

Using Config-Driven Design Patterns

Source Component's Must:

- *Auto-Wire themselves using Spark's Runtime Configuration*
- *Provide "sane" defaults*
- *Fail Fast and Correct the "user"*

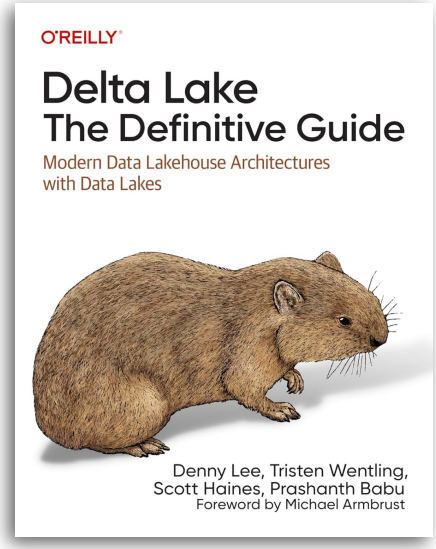
```python
class DeltaSourceGenerator:
    app: StreamingApp
    config_prefix: str = 'spark.app.source.delta'
    table_name: str

    options_default: Dict[str, Optional[str]] = {}

    def generate_options(self) -> Dict[str, str]:
        """
        Filters and expands options for a delta streaming source
        :return: the filtered and expanded options
        """
        return dict(
            {(opt[0], opt[1]) for opt in self.options.items() if opt[1] is not None}
        )

    def generate_stream_source(self) -> DataStreamReader:
        """
        Returns the generated stream. Requires calling `load` or `table`
        method (toDF) to generate a `DataFrame`
        :return: The DataStreamReader
        """
        return self.app.generate_read_stream(
            self.generate_options()
        ).format('delta')

    def toDF(self) -> DataFrame:
        table_name = self.table_name
        is_managed = "/" not in table_name

        reader: DataStreamReader = self.generate_stream_source()
        return reader.table(table_name) if is_managed else reader.load(table_name)
```

Data Stream Reading

Common Delta Lake Streaming Source Options

Getting a Handle on Delta's Streaming Source Options

- **Rate Limiting** *maxFilesPerTrigger / maxBytesPerTrigger*
- **Runtime Ignorance** *ignoreDeletes / ignoreChanges*
- **Time Travel and Pointers for Stream Triage** *startingVersion / startingTimestamp*
- **Stream Correctness** *withEventTimeOrder*
- **Powering through with Column Mapping Enablement** *schemaTrackingLocation*



Data Stream Reading

Generic Delta Lake Source Options Reading

Delegate and Be Lazy Where Possible

- Read the configuration via the Spark runtime config. With this pattern, the standard app (StreamingApp) can delegate config parsing and supply defaults.
- **Remember**: We are trying to do as little as possible here. The less we do, the easier our application is to manage.

```python
class DeltaSourceGenerator:
    app: StreamingApp
    config_prefix: str = 'spark.app.source.delta'
    table_name: str

    options_defaults: Dict[str, Optional[str]] = {...}

    def __init__(self, app: StreamingApp):
        """
        Lazy auto-wired source
        """
        self.app = app
        self.table_name = self._table_name()
        self.options = Functions.generate_options_dict(
            self.config_prefix, self.options_defaults, self.app.spark_session
        )
```

*Use `spark.conf` for config which helps with notebooks or vanilla data apps

Next Up.
Transformations...

Don't Panic. Still simple ☺

Data Stream Transformations

The Heart of the Application

Transformations.

- This is where the real work happens.
- Imagine a world where you no longer need to think about the “source(s)”, where you can **focus** on the output of your data applications, and simply config your “source(s) and sink(s)”.
- In other words, **focus on doing** vs the boiler plate.

```
spark.conf.set("spark.app.source.delta.table.catalog", catalog_name)
spark.conf.set("spark.app.source.delta.table.schema", schema_name)
spark.conf.set("spark.app.source.delta.table.name", table_name)
spark.conf.set("spark.app.checkpoints.path", checkpoints_path)
spark.conf.set("spark.app.sink.delta.table.catalog", sink_catalog_name)
spark.conf.set("spark.app.sink.delta.table.schema", sink_schema_name)
spark.conf.set("spark.app.sink.delta.table.name", sink_table_name)

app = StreamingApp(spark=spark)
delta_source_df = DeltaSourceGenerator(app).toDF()
transformed = delta_source_df.transform(...)
streaming_query = DeltaSinkGenerator(app, df=transformed).toTable()

# inject monitoring, awaiting termination
```

We can handle this ^^. Job Well Done

Next Up. Finishing the Writer

Don't Panic. Still simple ☺

Data Stream Writing

Take a DataFrame. Return a Stream.

Rinse and Repeat.

- Just like with the StreamingApp's generate_read_stream method.
- We can utilize composition in order to abstract away complexity.
- Let's look at that next.

```
@staticmethod  1 usage  Scott Haines
def generate_write_stream(
    df: DataFrame, s_options: Dict[str, str]
) -> DataStreamWriter:
    """
    Generate the DataStreamWriter. Note: This method can be further composed b
    Calling start or toTable on the DataStreamWriter creates a StreamingQuery
    :param df: The DataFrame to compose the pipeline
    :param s_options: The dictionary of key/value pairs
    :return: The DataStreamReader object for composition
    """

    return df.writeStream.options(**s_options)
```

StreamingApp.generate_write_stream

Data Stream Writing

Generating the Delta Lake Streaming Sink

Abstraction Away.

- Utilize Composition to drastically simplify writing boiler plate Delta Lake Streaming applications
- Lean on simple configuration.
- Enabling us to then simply focus on transformations, which we can get into in the next section.

```
class DeltaSinkGenerator:
    app: StreamingApp
    config_prefix: str = 'spark.app.sink.delta'
    table_name: str
    read_df: Optional[DataFrame] = None

    write_config_defaults: Dict[str, str] = {
        'outputMode': 'append',
        'timestampColumn': 'timestamp'
    }

    options_default: Dict[str, Optional[str]] = {
        'overwriteSchema': 'false',
        'mergeSchema': 'false'
    }

    def __init__(self, app: StreamingApp, read_df: Optional[DataFrame] = None):
        self.app = app
        self.read_df = read_df
        self.table_name = self._table_name()

        self.options = {
            'checkpointLocation': str(self.app.generate_checkpoint_location()),
            ...
        }

        def fromDF(self, read_df: Optional[DataFrame]=None) -> DataStreamWriter:
            """
            Like toDF, will generate a stream by receiving a DataFrame
            and auto-wiring the streaming sink options
            """
            df = read_df or self.read_df
            if df is None:
                raise RuntimeException("be kind...")
            builder: DataStreamWriter = self.app.generate_write_stream(
                df, self.generate_options()
            )
            writer = Functions.auto_trigger(
                Functions.auto_partition_or_clusterBy(builder, spark), spark)

            return writer.outputMode(self.write_config['outputMode']).format('delta')

        def toTable(self, read_df: Optional[DataFrame]=None) -> StreamingQuery:
            """
            Auto-Call and Auto-Wire the StreamingQuery using composition
            """
            writer = self.fromDF(read_df)
            return writer.toTable(tableName=self.table_name)
```

Generic Data Apps

Towards Application Composition

- Data Sources can be configured in a generic way allowing our applications to be controlled via configuration.
- The same is true for our Data Sink's.
- This leaves *us to focus on the core task of data "transformation".
- Luckily. We can get help there if we choose…

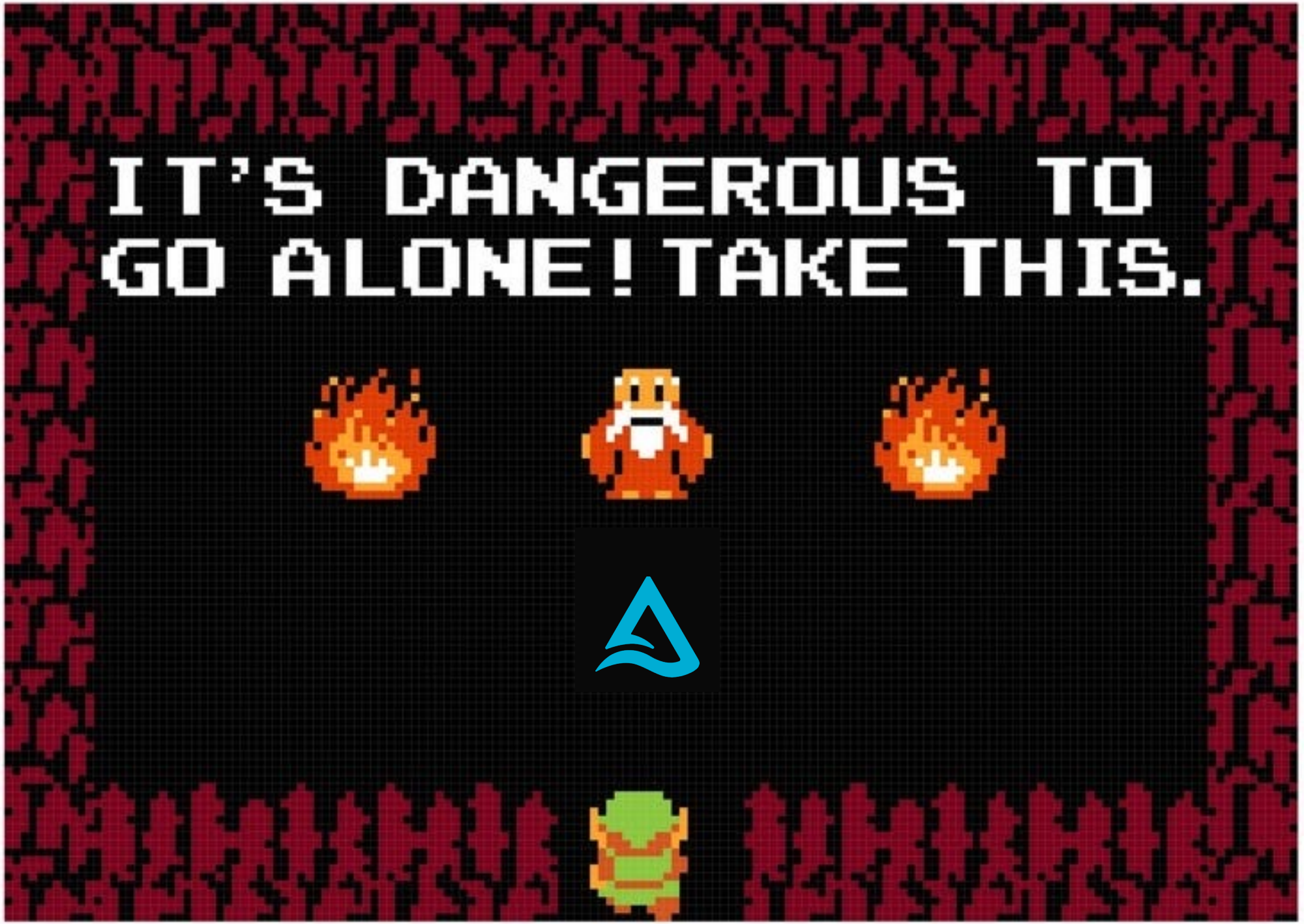
```
spark.conf.set("spark.app.source.delta.table.catalog", catalog_name)
spark.conf.set("spark.app.source.delta.table.schema", schema_name)
spark.conf.set("spark.app.source.delta.table.name", table_name)
spark.conf.set("spark.app.checkpoints.path", checkpoints_path)
spark.conf.set("spark.app.sink.delta.table.catalog", sink_catalog_name)
spark.conf.set("spark.app.sink.delta.table.schema", sink_schema_name)
spark.conf.set("spark.app.sink.delta.table.name", sink_table_name)

app = StreamingApp(spark=spark)
delta_source_df = DeltaSourceGenerator(app).toDF()
transformed = delta_source_df.transform(...)
streaming_query = DeltaSinkGenerator(app, df=transformed).toTable()

# inject monitoring, awaiting termination
```

Assembling the Pieces

Where Exactly are we Headed?

AI Agents can help us along our way by:

- Assisting us to find important information about our Tables
- Generating data for Tests (from our table information)
- Figuring out the right syntax for an Operation (joins, etc)
- And oh! so much more...


IT'S DANGEROUS TO
GO ALONE! TAKE THIS.

Workflow Automation with Agents

Don't Panic. This Part is Fun!

The Answer to the Great Question... Of Life, the Universe and Everything... Is...

Tools and the Number 42!

Crash Course: Agents and Tools

Tools are Functions that Provide Context

Why does Context Matter?

- LLMs know a “ton”. But sometimes struggle to distinguish between “a ton” – being a unit of measurement or an expression.
- Context provides a Guiding Light to a system in the Dark.
- LLMs understand language and patterns really well, so tools must also act as a part of the “memory” of the “system at large”

Figure 42: Tools act like “parts” that can be combined to engineer much more complex things.

LangGraph

Tools 101

What is an Agentic Tool?

- It is a simple function or “api-like” interface that enables an LLM to “touch” the outside world.
- We provide the utility to bridge the gap between “the LLM void” and our ”world”

```python
class ListCatalogsRequest(BaseModel):
    max_results: Optional[int] = Field(default=1,
                                       description="The number of schema's to return from the request",
                                       gt=0, lt=100)

class ListCatalogsResponse(BaseModel):
    catalogs: List[str] = Field(description="The name of the base catalog")

@tool("list_catalogs",
      description="returns a list of base catalogs. Individually, " \
      "each catalog name can be used as the parent catalog reference for schema or table discovery",
      args_schema=ListCatalogsRequest,
      return_direct=True)
def list_catalogs(max_results: Optional[int] = 1) -> ListCatalogsResponse:
    # returns an Iterator[CatalogInfo]
    print(f"limit to max_results:{max_results}")
    try:
        # browse_only, catalog_type, comment, metastore_id, securable_type, properties
        return ListCatalogsResponse(
            catalogs=[clean_string(catalog.name) for catalog in workspace_client.catalogs.list()])
    except Exception as e:
        raise ToolException(f"Failed to List the Catalogs: {e}")

```

Speeding Up Now. Let’s automate a data pipeline

LangGraph

Let's Build a Pipeline

What Will We Need?

- Access to Unity Catalog
- Tools to do the following:
  - List Catalogs [check]
  - List Schemas of a Catalog
  - List Tables of a Schema in a Catalog
  - Get Table Details and all Known Metadata

LangGraph

Let's Build a Pipeline

```python
class ListSchemaRequest(BaseModel):
    catalog_name: str = Field(description="The name of the parent catalog (Unity Catalog) containing this schema")
    max_results: Optional[int] = Field(default=1, description="The number of schema's to return from the request", gt=0, lt=100)

class ListSchemaResponse(BaseModel):
    catalog_name: str = Field(description="The name of the parent catalog (Unity Catalog) containing this schema")
    schemas: Optional[List[str]] = Field(default=None, description="The name of the schemas. Schemas are also referred to as databases in older systems")

@tool(
    "list_schemas",
    description="returns a list of schema names, if they exist, from the associated unity catalog name",
    args_schema=ListSchemaRequest,
    return_direct=True)
def list_schemas(catalog_name: str, max_results: Optional[int] = 1) -> ListSchemaResponse:
    # returns an Iterator[SchemaInfo]
    print(f"limit to max_results:{max_results}")
    try:
        schemas = workspace_client.schemas.list(
            catalog_name=clean_string(catalog_name),
            include_browse=False,
            max_results=max_results
        )
        return ListSchemaResponse(
            catalog_name=clean_string(catalog_name),
            schemas=[schema.name for schema in schemas]
        )
    except Exception as e:
        raise ToolException(f"Failed to List the Schemas: {e}")
```

LangGraph

LangChain

Let's Build a Pipeline

```python
class ListTablesRequest(BaseModel):
    catalog_name: str = Field(description="The name of the parent catalog (Unity Catalog) containing this schema")
    schema_name: str = Field(description="The name of the schema containing the tables to list. The schema is a child of the parent catalog named catalog_name")
    max_results: Optional[int] = Field(default=10, description="The number of table's to list in this request", gt=0, lt=100)

class ListTablesResponse(BaseModel):
    catalog_name: str = Field(description="The name of the parent catalog (Unity Catalog) containing this schema")
    schema_name: str = Field(description="The name of the schema containing the tables to list. The schema is a child of the parent catalog named catalog_name")
    tables: Optional[List[str]] = Field(default=None, description="The list of full table names. Each table's full name includes the catalog.schema.table_name.")

@tool(
    "list_tables",
    description="returns a list of table names, if any exist, from the associated catalog_name.schema_name provided in the request",
    args_schema=ListTablesRequest,
    return_direct=True)
def list_tables(catalog_name: str, schema_name: str, max_results: Optional[int] = 10) -> ListTablesResponse:
    # returns an Iterator[TableInfo]
    print(f"limit to max_results:{max_results}")
    try:
        tables = workspace_client.tables.list(
            catalog_name=clean_string(catalog_name),
            schema_name=clean_string(schema_name),
            max_results=max_results,
            include_delta_metadata=False,
            include_manifest_capabilities=False,
            omit_username=True,
            omit_columns=True,
            omit_properties=True,
        )
        return ListTablesResponse(
            catalog_name=clean_string(catalog_name),
            schema_name=clean_string(schema_name),
            tables=[table.full_name for table in tables if table.full_name is not None]
        )
    except Exception as e:
        raise ToolException(f"Failed to List the Tables under the {catalog_name}.{schema_name}")
```

LangGraph

Let's Build a Pipeline

```python
# The TableColumn class is a lighter weight version of "TableInfo". This allows us to tune the amount of data we provide
# to the LLM. This can be extended if you wanted to "query" system tables to understand what "tags" are associated with a given table
# or even to learn from the lineage to see what other tables are "commonly" constructed from "this" table.
class TableColumn(BaseModel):
    name: str = Field(description="The name of the column. This is used for querying the table")
    sql_type: str = Field(description="Full data type specification as SQL/catalogString text")
    comment: Optional[str] = Field(default=None, description="Describes what the column represents. The comment provides a semantic understanding for better queries")
    nullable: Optional[bool] = Field(default=True, description="If the value is True, then this field isn't required and we can't count on it to always be present")
```

Python

```python
class TableDetailsRequest(BaseModel):
    catalog_name: str = Field(description="The name of the parent catalog (Unity Catalog) containing this schema")
    schema_name: str = Field(description="The name of the schema containing the tables to list. The schema is a child of the parent catalog named catalog_name")
    table_name: str = Field(description="The name of the table to inspect and discover. This is a child of the associated schema")

# This is fed from the ColumnInfo class via the Databricks SDK
class TableDetailsResponse(BaseModel):
    full_table_name: str = Field(description="The full name of the table. The full name includes the catalog.schema.table_name")
    comment: Optional[str] = Field(default=None, description="The table description if it has been provided")
    columns: Optional[List[TableColumn]] = Field(default=None, description="The columns of the table. These are important for querying the table")

@tool(
    "get_table_details",
    description="Provides metadata about a table within Unity Catalog. The details include the table name, comment, description, columns, cluster or partition information, as w
    args_schema=TableDetailsRequest,
    return_direct=True)
def get_table_details(catalog_name: str, schema_name: str, table_name: str) -> TableDetailsResponse:
    full_name = f"{clean_string(catalog_name)}.{clean_string(schema_name)}.{clean_string(table_name)}"
    print(f"getting table details: for `{full_name}`")
    try:
        table_details = workspace_client.tables.get(full_name=full_name, include_delta_metadata=True, include_browse=True, include_manifest_capabilities=True)
        #print(f"{table_details}")
        return TableDetailsResponse(
            full_table_name=full_name,
            comment=table_details.comment,
            columns=[
                TableColumn(
                    name=column.name,
                    sql_type=column.type_text,
                    comment=column.comment,
                    nullable=column.nullable
                ) for column in table_details.columns]
        )
    except Exception as e:
        raise ToolException(f"Failed to collect the Table details and metadata for {full_name}")
```

Fruits of “Our” Labor

Remember. This is only using Metadata

LangGraph

Let's Run "Scott" 2.0

Full Video will be on the YouTube recording from the presentation. It is too big for upload.

Closing Thoughts

Towards Agentic Augmented Application Composition

- Data Pipelines are not that hard to make
- Using Config-Driven software engineering design patterns can take us 90% of the way.
- Getting agentic assistance is amazing, especially when it saves hours of effort, or actual meetings ☺

Closing Thoughts

Towards Agentic Augmented Application Composition

- Context with respect to “short term” memory is a tough nut to crack.
- Providing all the “right tools” can take a long time, especially when figuring out the right ”level of focus”.

Cheers

So Long and Thanks for all the Fish!

Now do we still have time for Questions? (future self>..)