DATA AND AI

Advanced AIRFLOW Techniques

Shwetank Singh
GritSetGrow - GSGLearn.com

gsglearn.com

Data and AI

Dynamic Task Mapping

Dynamically generate tasks at runtime based on input data for scalable and flexible DAGs.

Processing a variable number of files or datasets.

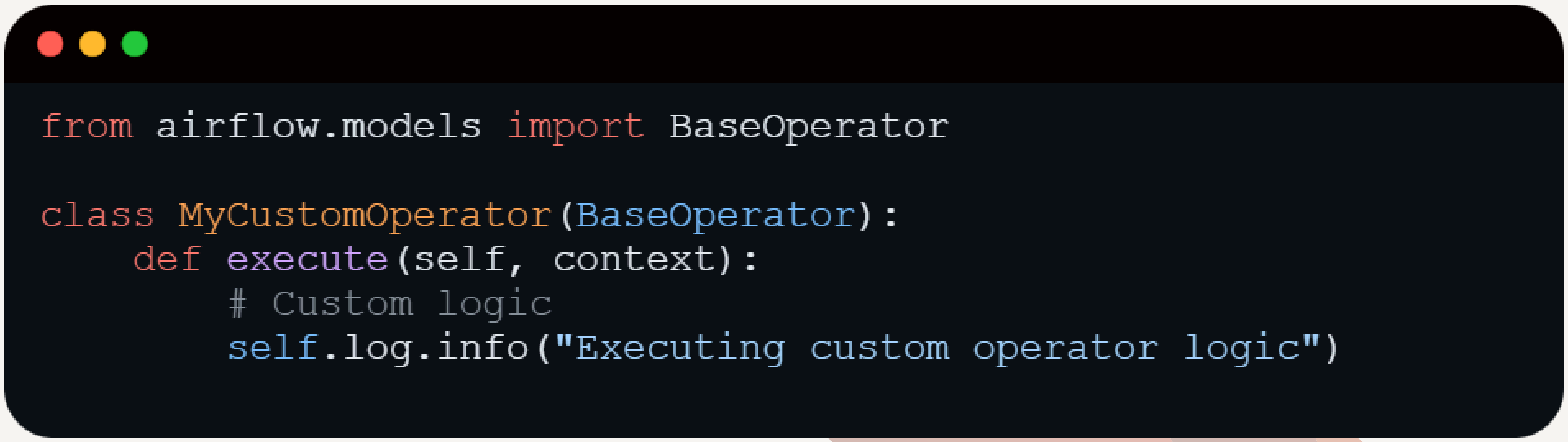
```
PythonOperator.partial(
    task_id="process_file",
    python_callable=process_file
).expand(
    op_args=files
)
```

Allows Airflow to create tasks on-the-fly based on the data, making DAGs adaptable to varying workloads without predefined task counts.

Custom Operators

Create tailored operators by extending Airflow's base classes to encapsulate specific functionality.

Integrating with a proprietary API or specialized tasks.



```python
from airflow.models import BaseOperator

class MyCustomOperator(BaseOperator):
    def execute(self, context):
        # Custom logic
        self.log.info("Executing custom operator logic")
```

Enables encapsulation of complex or unique operations within reusable components, promoting cleaner DAG definitions and reusability across workflows.

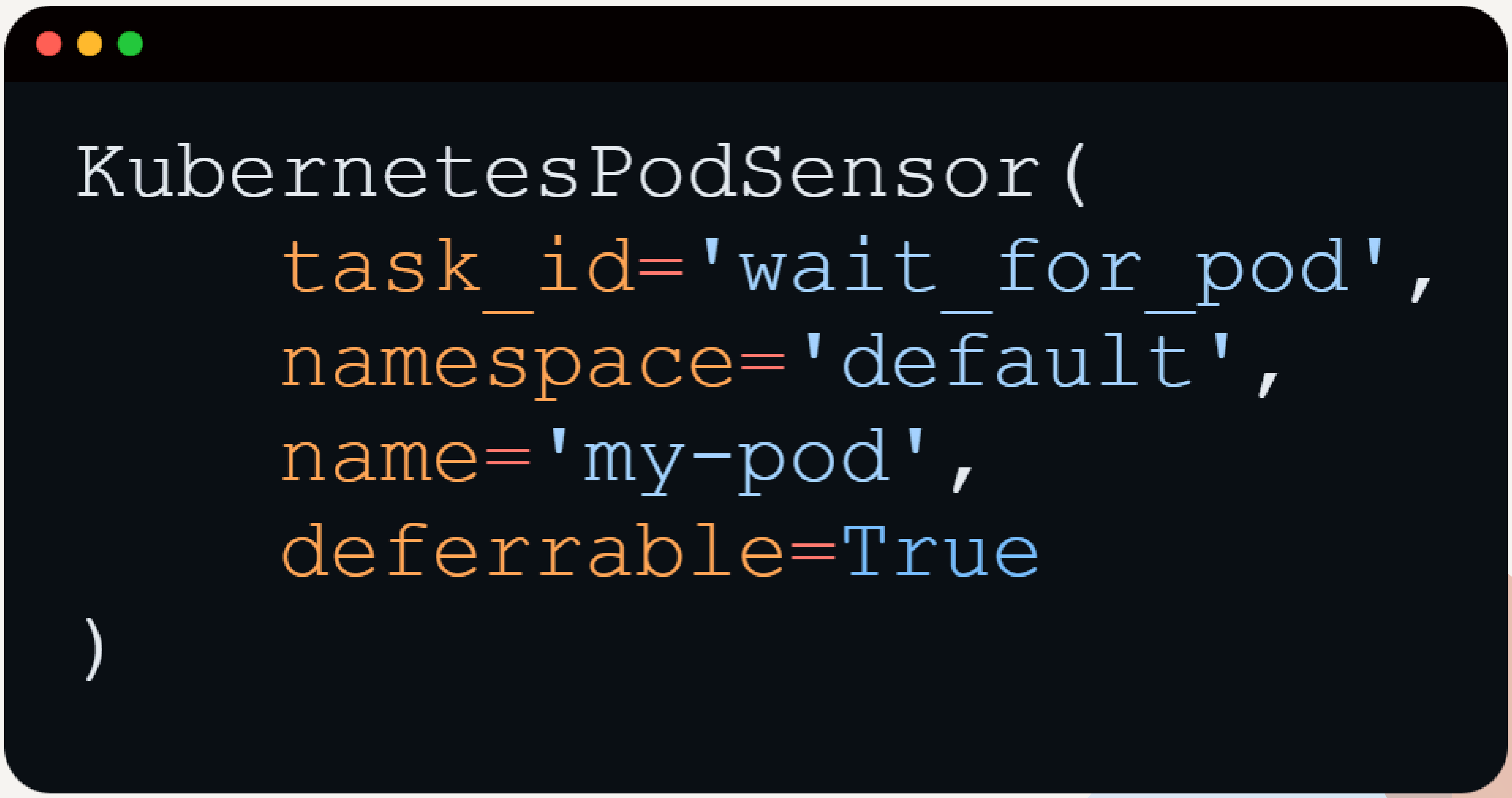
Shwetank Singh
GritSetGrow - GSGLearn.com

Data and AI

Deferrable Operators

Operators that defer execution and free up worker slots by leveraging asynchronous triggers.

Waiting for external events like file arrivals.



```
KubernetesPodSensor(
    task_id='wait_for_pod',
    namespace='default',
    name='my-pod',
    deferrable=True
)
```

Enhances scalability by allowing tasks to wait for events without occupying worker slots, optimizing resource usage and handling more concurrent workflows.

Shwetank Singh
GritSetGrow - GSGLearn.com

Data and AI

Task Groups

Organize tasks into logical groups within a DAG for better readability and manageability.

Structuring ETL pipelines with multiple transformation steps.

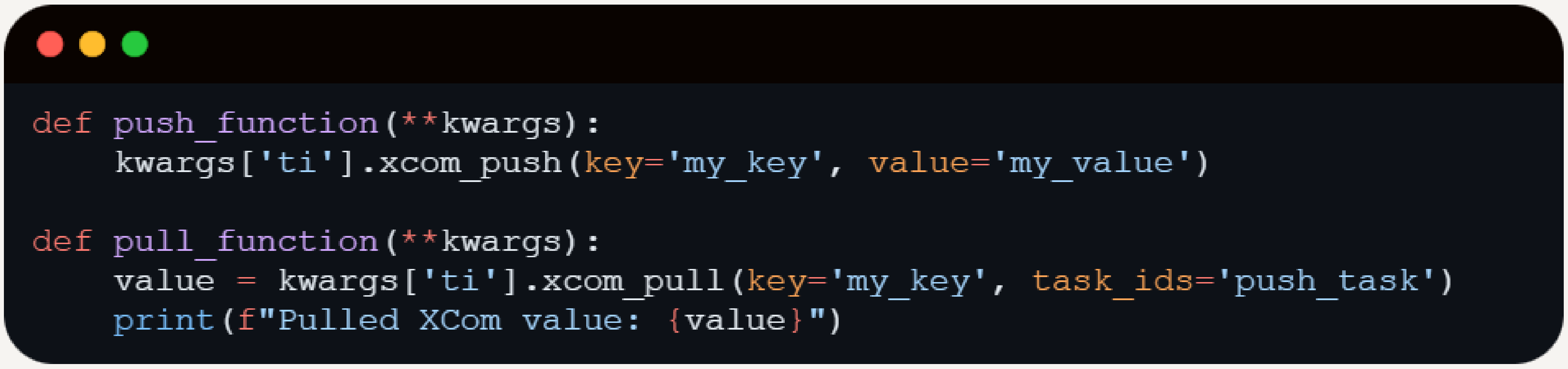
```
from airflow.operators.dummy import DummyOperator
from airflow.utils.task_group import TaskGroup

with TaskGroup('processing_group') as processing:
    task1 = DummyOperator(task_id='task1')
    task2 = DummyOperator(task_id='task2')
```

Helps manage large DAGs by encapsulating related tasks, making them easier to navigate and maintain with a hierarchical structure.

XComs (Cross-Communication)

Mechanism for tasks to exchange small amounts of data during DAG execution.

Passing file paths or statuses between tasks.



```python
def push_function(**kwargs):
    kwargs['ti'].xcom_push(key='my_key', value='my_value')

def pull_function(**kwargs):
    value = kwargs['ti'].xcom_pull(key='my_key', task_ids='push_task')
    print(f"Pulled XCom value: {value}")
```

Allows tasks to share metadata or small data snippets, facilitating inter-task communication without external storage.

Data and AI

SubDAGs

Embed a DAG within another DAG to modularize complex workflows.

Reusing a sequence of tasks across multiple DAGs.

```python
from airflow.operators.subdag import SubDagOperator

subdag = SubDagOperator(
    task_id='subdag_task',
    subdag=child_dag(),
    dag=parent_dag
)
```

Enables reusability and modularization by allowing a group of tasks to be reused as a single task within multiple DAGs.

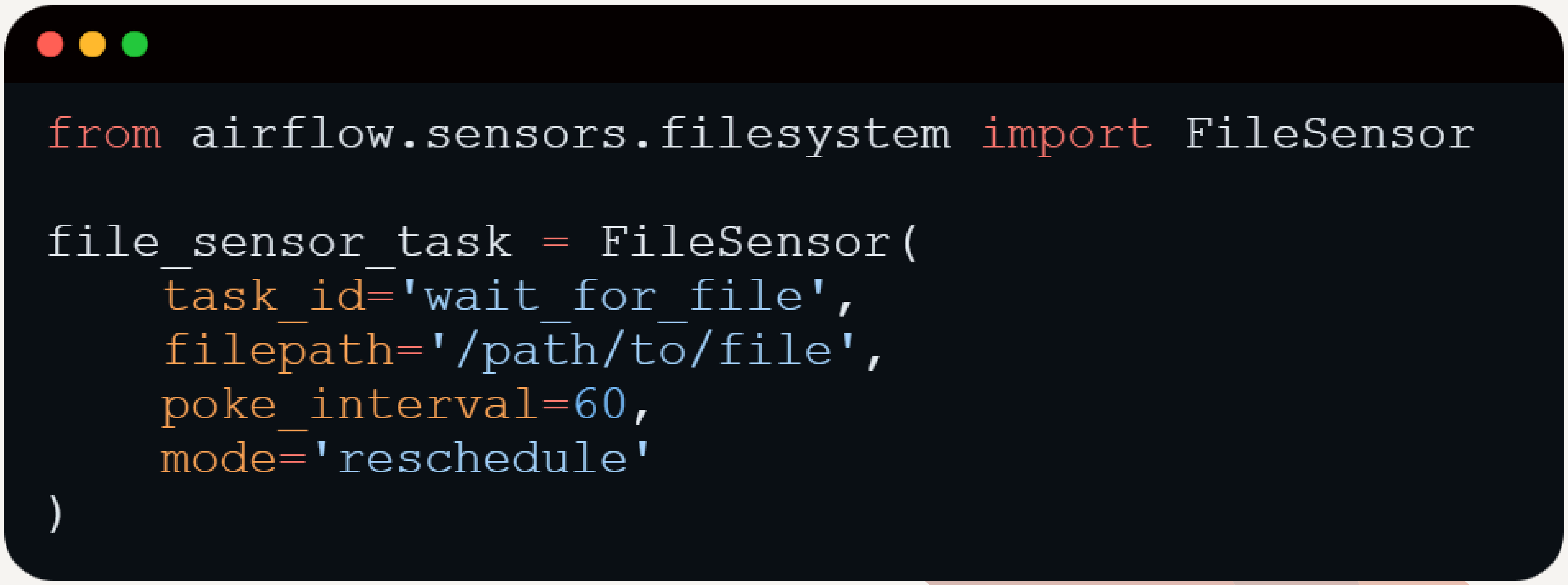
Shwetank Singh
GritSetGrow - GSGLearn.com

Data and AI

Sensor Improvements

Use advanced sensors like AsyncSensor to improve efficiency and reduce resource usage.

Monitoring for file existence without occupying slots.



```python
from airflow.sensors.filesystem import FileSensor

file_sensor_task = FileSensor(
    task_id='wait_for_file',
    filepath='/path/to/file',
    poke_interval=60,
    mode='reschedule'
)
```

Enhances sensor performance by allowing them to reschedule instead of continuously occupying worker slots, leading to better resource management.

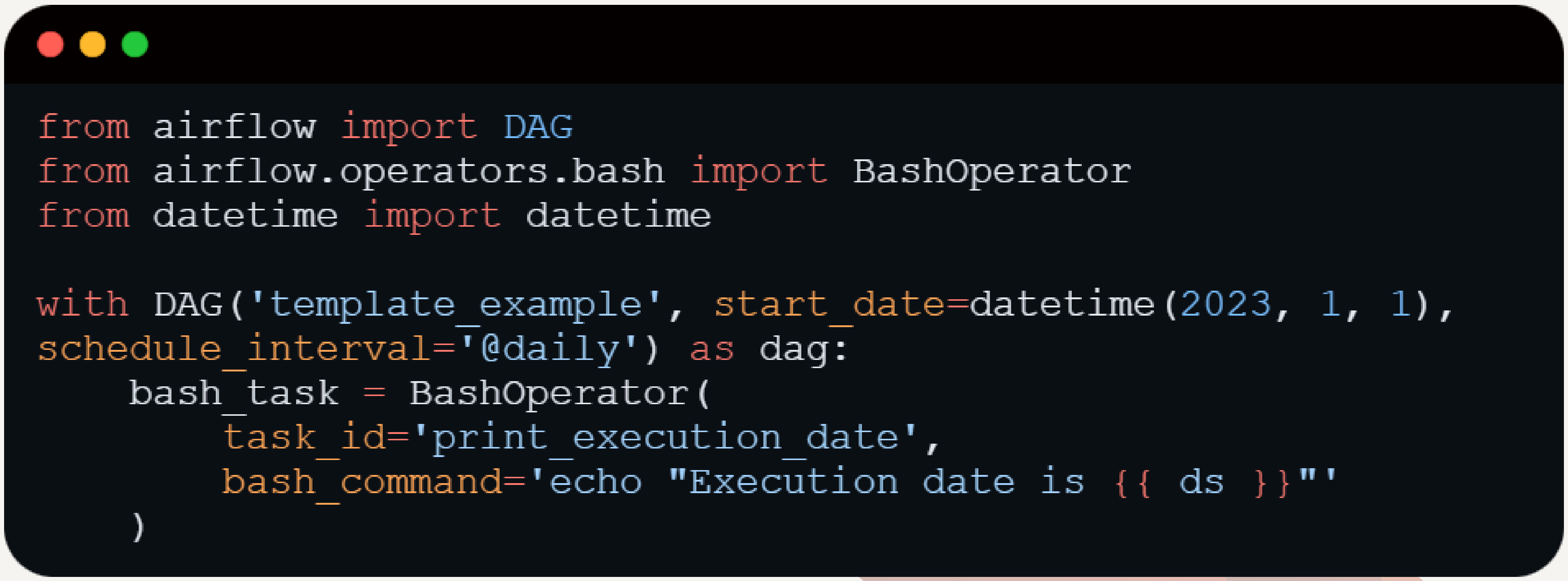
Shwetank Singh
GritSetGrow - GSGLearn.com

Data and AI

Macros and Jinja Templating

Utilize macros and Jinja templating for dynamic DAG and task parameterization.

Dynamic file paths based on execution date.



```python
from airflow import DAG
from airflow.operators.bash import BashOperator
from datetime import datetime

with DAG('template_example', start_date=datetime(2023, 1, 1),
schedule_interval='@daily') as dag:
    bash_task = BashOperator(
        task_id='print_execution_date',
        bash_command='echo "Execution date is {{ ds }}"'
    )
```

Enables dynamic generation of task parameters and DAG configurations based on runtime variables, enhancing flexibility and customization.

Shwetank Singh
GritSetGrow - GSGLearn.com

Plugins and Extensions

Extend Airflow functionality by developing custom plugins.

Adding new UI components or operators.

```
from airflow.plugins_manager import AirflowPlugin

class MyPlugin(AirflowPlugin):
    name = "my_plugin"
    operators = [MyCustomOperator]
```

Allows customization and extension of Airflow's core features, enabling the addition of new operators, sensors, hooks, or UI elements tailored to specific needs.

Data and AI

Airflow REST API

Interact with Airflow programmatically using its REST API for automation and integration.

Triggering DAG runs from external systems.

```
curl -X POST "http://localhost:8080/api/v1/dags/my_dag/dagRuns" \
     -H "Content-Type: application/json" \
     -d '{"conf": {"key": "value"}}'
```

Facilitates integration with other tools and systems by allowing programmatic control over DAG executions, monitoring, and management via HTTP requests.

Data and AI

Versioned DAGs

Manage and deploy multiple versions of DAGs to ensure stability and backward compatibility.

Rolling out new DAG versions without disrupting ongoing runs.

Implement versioning in DAG file names and use Git for version control.

Ensures that updates to DAGs do not interfere with existing workflows by maintaining multiple versions, allowing for safe testing and gradual rollout of changes.

Airflow Variables and Connections

Use Variables and Connections to manage dynamic configurations and sensitive information securely.

Storing API keys and dynamic parameters for tasks.

```python
from airflow.models import Variable, Connection

api_key = Variable.get("api_key")
```

Provides a secure and centralized way to manage configuration parameters and credentials, enhancing security and flexibility in DAG configurations and task executions.

Data and AI

Retry and Alerting Strategies

Define sophisticated retry mechanisms and alerting for task failures to improve reliability.

Automatically retrying failed tasks and notifying stakeholders.

```python
from airflow.operators.python import PythonOperator
from datetime import timedelta

task = PythonOperator(
    task_id='task',
    python_callable=my_func,
    retries=3,
    retry_delay=timedelta(minutes=5)
)
```

Enhances the robustness of workflows by automatically handling transient failures and providing timely notifications for persistent issues, ensuring higher reliability.

Thank you!