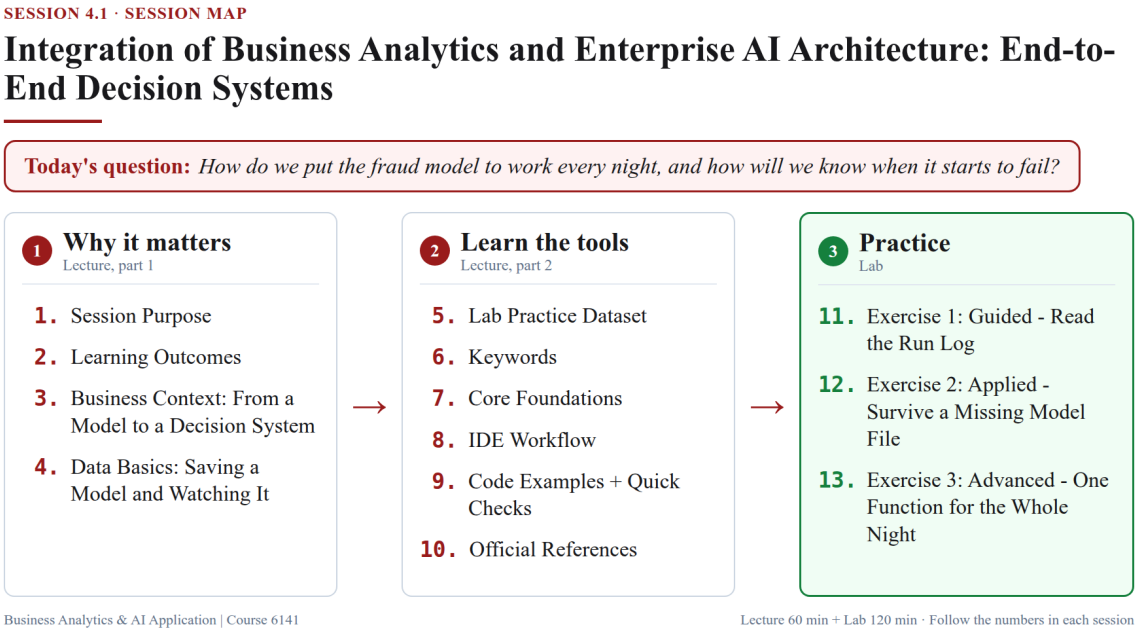
Session 4.1: Integration of Business Analytics and Enterprise AI Architecture: End-to-End Decision Systems

Course 6141: Business Analytics & AI Application | Reading for Week 13



Schedule: Week 13 | In-person classes (1:2 Delivery Ratio: 60 Min Lecture + 120 Min Lab)

1. Session Purpose

- A model that lives in an analyst's notebook creates no value; value comes only when its predictions trigger real actions, every day, without a person running the code
- The card issuer from Session 3.4 wants its fraud model to score each night's transactions automatically and hold the risky ones
- The head of operations asks: how do we put the fraud model to work every night, and how will we know when it starts to fail? Today, you save a trained model, reuse it on new data, and build the checks that warn you when the world changes

2. Expected Learning Outcomes (By the end of this session, you will be able to)

1. "How does a model get from training to daily use?": map an end-to-end decision system (data in, checks, model, action), and save and reload a model with `joblib`
2. "Is new data still like the training data?": detect data drift (a change in the incoming data that makes a model less reliable) by comparing averages
3. "What if something breaks at night?": monitor latency and accuracy from a run log, and handle a missing model file (`FileNotFoundError`) with a safe fallback

3. Business Context: From a Model to a Decision System

- A decision system has four stages, run every night as one batch (a group of records processed together):

Stage	What happens	Example
Ingestion	collect the day's data	120,000 card transactions
Validation	check it before use	no missing values, all columns present
Scoring	the model gives each record a probability	fraud probability from 0 to 1
Action	a business rule turns scores into actions	hold every transaction at 0.5 or above

- Batch vs real-time:
  - **Batch** scoring runs on a schedule, such as every night, for many records at once. It is cheaper, and fine when decisions can wait a few hours.
  - **Real-time** scoring answers one request within milliseconds, for example while a customer waits at the checkout. It is faster but costs more to build and run.
- Models get worse over time, because customers and markets change. Two warning signs to watch:
  - **Data drift:** the new data no longer looks like the training data (for example, purchases are much larger in the holiday season)
  - **Performance decay:** accuracy on recent, checked cases falls

4. Data Basics: Saving a Model and Watching It

- Training is slow and done rarely; scoring is fast and done often. So the trained model is **saved to a file** once, then **loaded** by the nightly job:
  - `joblib.dump(model, "models/fraud_model.joblib")` saves it
  - `joblib.load("models/fraud_model.joblib")` loads it back, ready to predict, with no retraining
- A simple drift check compares each feature's average in the new batch with its average in the training data:
  - change in percent = (new average - training average) / training average x 100

  - Our rule: flag a feature when the change is more than 20% in either direction

- A run log records every night's stages with their record counts and latency (how long a stage took, in milliseconds; 1,000 ms = 1 second). Rising latency or falling accuracy is an early warning.

5. Lab Practice Dataset

- Three files (all fictional teaching data):
  - Training data: `data/session_03_04_dataset.csv`, the 16 investigated transactions from Session 3.4. The model uses `Transaction_Amount_USD`, `Foreign_IP_Flag`, and `Failed_Login_Attempts_Count` to predict `Actual_Fraud_Binary`.
  - New batch: `data/session_04_01_new_batch.csv`, 8 new transactions from a holiday-season night, with the same three feature columns and no answer column.
  - Run log: `data/session_04_01_dataset.csv`, 20 rows: 5 nights x 4 stages.
- Run log columns:
  - `Run_Date`: The night of the run, `2024-12-01` to `2024-12-05`.
  - `Pipeline_Stage`: `Ingestion`, `Validation`, `Scoring`, or `Action`.
  - `Records_Count`: Records processed by the stage. For `Action`, the number of held transactions.
  - `Latency_Ms`: Time the stage took, in milliseconds.
  - `Model_Accuracy`: Accuracy on that night's investigated cases. Filled only for the `Scoring` stage; empty (`NaN`) for the others.

6. Keywords (Core Programming Concepts)

- `End-to-End Decision System`:
  - What: ingestion, validation, scoring, and action joined into one automatic run
  - Why: turns predictions into business actions without manual steps
- `Batch vs Real-Time Scoring`:
  - What: many records on a schedule vs one record on request
  - Why: the business deadline decides which one is worth its cost
- `Model Persistence` (`joblib`):
  - What: saving a trained model to a file and loading it later
  - Why: train once, score every night
- `Data Drift`:
  - What: new data that no longer looks like the training data
  - Why: a model is only reliable on data like the data it learned from
- `Monitoring`:
  - What: tracking counts, latency, and accuracy for every run
  - Why: catches a failing system before customers do
- `Fallback`:

- What: a simple, safe rule used when the model is not available
- Why: the business keeps running when something breaks

7. Core Programming & Computational Foundations

- **7.1 Train once, use many times: saving a model**
  - *The idea:* Training a model is slow and done rarely; using it is fast and done every night. So the trained model is saved to a file once, and the nightly job loads that file instead of training again. `joblib.dump(model, path)` saves it; `joblib.load(path)` loads it back, exactly as it was, ready to predict.
  - *See it on real data:*

Step	Code	Result
Save	`joblib.dump(model, "models/fraud_model.joblib")`	a small file, about 1.7 KB
Load	`loaded = joblib.load("models/fraud_model.joblib")`	the same trained model
Check	`(model.predict_proba(X) == loaded.predict_proba(X)).all()`	`True`: identical predictions

  - *Why save the whole `Pipeline`:* the scaler and the model are saved together, so the nightly job scales new data with the training averages (`1371.72`, `0.5`, `1.56`), exactly as during training. Saving only the model and re-fitting a scaler on tonight's data would be data leakage (Session 3.2).
  - *Common mistake:* a wrong path. `joblib.load("models/fraud_modle.joblib")` raises `FileNotFoundError: [Errno 2] No such file or directory`. Check first with `os.path.exists(path)`.
  - *More examples:*

Code	Result	What it tells you
`os.makedirs("models", exist_ok=True)`	creates the folder	`exist_ok=True`: no error if it is already there
`os.path.exists("models/fraud_model.joblib")`	`True`	the file is in place
`os.path.exists("models/nope.joblib")`	`False`	check before loading
`list(loaded.named_steps)`	`['scale', 'classify']`	the loaded pipeline still has both steps

  - *Remember:* save the whole pipeline once; load it every night.
- **7.2 Probabilities, not just yes or no**

- *The idea:* `model.predict(X)` returns a decision, `1` or `0`, using a cut-off of 0.5. `model.predict_proba(X)` returns the probabilities behind it: two numbers per row, the probability of class `0` and of class `1`, which always add up to 1. With probabilities, the business can choose its own cut-off and rank cases by risk.
- *See it on real data:* the first two new transactions.

Transaction	`predict_proba`	`[:, 1]` (fraud probability)	`predict`
`N-TXN-9001`	`[0.138, 0.862]`	`0.862`	`1`
`N-TXN-9002`	`[0.930, 0.070]`	`0.070`	`0`

`[:, 1]` means "all rows (`:`), second column (`1`)".

- *More examples:*

Code (`proba = model.predict_proba(X)`)	Result	What it tells you
`proba.sum(axis=1)`	`1.0` for every row	the two probabilities always add up to 1
`(proba[:, 1] >= 0.9).sum()`	`2`	a strict cut-off holds only the 2 clearest cases
`(proba[:, 1] >= 0.5).sum()`	`4`	the default cut-off
`(proba[:, 1] >= 0.1).sum()`	`5`	a cautious cut-off also holds `N-TXN-9006` (`0.163`)
`proba[:, 1].mean()`	`0.499`	the expected share of fraud in tonight's batch

- *Remember:* use probabilities when the cut-off is a business decision.

- **7.3 Data drift: is tonight's data like the training data?**
  - *The idea:* A model learned its patterns from past data. If new data looks very different, for example much larger purchases in the holiday season, the model is being asked about situations it rarely saw, and its scores become less reliable. A simple check compares each feature's average in the new batch with its average in the training data.
  - *See it on real data:*

Feature	Training average	Tonight's average	Change
`Transaction_Amount_USD`	`1371.72`	`2555.00`	`+86.3%` (drift)
`Foreign_IP_Flag`	`0.50`	`0.50`	`0.0%`
`Failed_Login_Attempts_Count`	`1.56`	`1.75`	`+12.2%`

2 of tonight's 8 purchases are larger than the largest training purchase (`4200.0`): the biggest is `5200.0`.

  - *Common mistake:* checking only for increases. A feature that falls by 30% has drifted too; use the absolute change, `.abs()`.

  - *More examples:*

Measure	Training	Tonight	What it tells you
median amount	`684.7`	`2150.0`	`+214.0%`: the median shows the shift even more clearly
largest amount	`4200.0`	`5200.0`	tonight goes beyond anything seen in training
purchases above the training maximum		`2 of 8`	scores for these two need extra caution

  - *Remember:* compare new averages with training averages, in both directions.

- **7.4 Monitoring: reading the run log**
  - *The idea:* A system that runs every night must report on itself. A run log records each stage of each night: how many records, how long it took (latency), and, where checked cases exist, how accurate the model was. Trends in the log warn you before customers notice a problem.
  - *See it on real data:* the Scoring stage over 5 nights.

Night	Latency (ms)	Accuracy	Held as share of incoming
2024-12-01	`820`	`0.945`	`4.02%`
2024-12-02	`840`	`0.942`	`4.04%`
2024-12-03	`860`	`0.938`	`4.08%`
2024-12-04	`910`	`0.921`	`4.48%`
2024-12-05	`980`	`0.904`	`4.85%`

    Scoring is slower, less accurate, and holds more transactions every night: three signals pointing the same way.
  - *Common mistake:* treating an empty value as a problem. `Model_Accuracy` is empty (`NaN`) for 15 of 20 rows by design, because only the Scoring stage has an accuracy.
  - *More examples:*

Code	Result	What it tells you
`log.groupby("Pipeline_Stage")["Latency_Ms"].mean()`	Scoring `882.0`, Ingestion `365.0`, Validation `193.0`, Action `126.0`	scoring is the slowest stage
`log["Model_Accuracy"].isnull().sum()`	`15`	empty by design outside Scoring
`log[log["Pipeline_Stage"] == "Scoring"]["Model_Accuracy"].min()`	`0.904`	the worst night

  - *Remember:* log every run, then watch the trends, not just single nights.

- **7.5 When something breaks: a safe fallback**
  - *The idea:* A nightly system must keep the business safe even when a part fails. A **fallback** is a simple rule used when the model is not available, for example "hold foreign purchases with 2 or more failed logins". It is weaker than the model, but much better than holding nothing or stopping everything.

```
if os.path.exists(model_path):
    use the model
else:
    use the fallback rule, and report that it was used
```

  - *See it on real data:* the fallback holds 3 of tonight's transactions; the model holds 4. It misses `N-TXN-9005`, a foreign purchase of $3,900 with only 1 failed login.
  - *More examples:*

Transaction	Facts	Model	Fallback rule
`N-TXN-9001`	foreign, 3 failed logins	Hold	Hold
`N-TXN-9003`	foreign, 5 failed logins	Hold	Hold
`N-TXN-9005`	foreign, 1 failed login, $3,900	Hold	Pass
`N-TXN-9007`	foreign, 4 failed logins	Hold	Hold
the 4 domestic purchases	0 or 1 failed logins	Pass	Pass

  - *Remember:* plan for failure, and always report which method made the decisions.

8. Development Workflow in Google Antigravity IDE

- `[1]` File Explorer: create your script with your name and the session code (for example, `Reacher_S41_demo_1.py`), and check the three files are in `data/`. Micro-Example 1 creates a new `models/` folder; it appears here.
- `[2]` Code Editor: write a few lines, run, repeat. Small steps keep errors small.
- `[3]` AI Agent Chat: on an error, capture the traceback (Python's error report) with `Win + Shift + S` (macOS: `Cmd + Shift + 4`), paste it with `Ctrl + V`, and ask why it failed.
- `[4]` Terminal: run `python` followed by your script name. Return code `0` means the script finished with no errors. First time only: `pip install scikit-learn joblib`.

9. Hands-On Coding Micro-Examples

- Every example has three parts: **1. Input**, **2. Process**, and **3. Output**. Name each file with your own name and the session code, for example `Reacher_S41_demo_1.py` (pattern: `YourName_S41_demo_1.py`). Run Micro-Example 1 first: the other examples load the model it saves.
- *Micro-Example 1 (Train Once, Save to a File):* `YourName_S41_demo_1.py`

```
# 1. Input: the investigated transactions from Session 3.4
import os
import joblib
import pandas as pd
```

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
train = pd.read_csv("data/session_03_04_dataset.csv")
features = ["Transaction_Amount_USD", "Foreign_IP_Flag",
            "Failed_Login_Attempts_Count"]

# 2. Process: scale + logistic regression, trained in one Pipeline
model = Pipeline([
    ("scale", StandardScaler()),
    ("classify", LogisticRegression()),
])
model.fit(train[features], train["Actual_Fraud_Binary"])

# 3. Output: save the trained model as a file
os.makedirs("models", exist_ok=True)
joblib.dump(model, "models/fraud_model.joblib")
print("Saved: models/fraud_model.joblib")
print("Training accuracy:", model.score(train[features],
 ↪  train["Actual_Fraud_Binary"]))
```

  - *How it works, line by line:*
    - The training data is the 16 investigated transactions from Session 3.4; the model uses 3 features.
    - `Pipeline([("scale", StandardScaler()), ("classify", LogisticRegression())])` joins two steps into one object: first standardize, then classify. Each step has a name and a tool.
    - `model.fit(...)` fits both steps in order.
    - `os.makedirs("models", exist_ok=True)` creates the `models` folder if needed; `joblib.dump()` writes the trained pipeline into it (section 7.1).
    - `model.score(...)` measures accuracy on the training rows.
  - *Read the output:*

```
Saved: models/fraud_model.joblib
Training accuracy: 0.9375
```

    - The file now exists in the `models` folder, visible in Panel `[1]`.
    - `0.9375` = 15 of 16 correct. It is measured on the rows the model learned from, so it is a ceiling, not an honest test.

- *Quick Check 1 (`os.path.exists()`):*
  - New tool: `os.path.exists(path)` returns `True` if the file exists. Why: the nightly job should confirm the model is there before it starts.
  - Task: add one line to part **3.** **Output**: `print(os.path._____("models/fraud_model.joblib"))`
  - Check: the Terminal shows `True`.

- *Micro-Example 2 (The Nightly Job: Load, Score, Decide):* `YourName_S41_demo_2.py`

```
# 1. Input: the saved model and tonight's new transactions
import joblib
import pandas as pd
model = joblib.load("models/fraud_model.joblib")
batch = pd.read_csv("data/session_04_01_new_batch.csv")
features = ["Transaction_Amount_USD", "Foreign_IP_Flag",
            "Failed_Login_Attempts_Count"]

# 2. Process: fraud probability, then the business rule (hold at 0.5 or above)
batch["Fraud_Probability"] = model.predict_proba(batch[features])[:, 1].round(3)
batch["Decision"] = "Pass"
batch.loc[batch["Fraud_Probability"] >= 0.5, "Decision"] = "Hold"

# 3. Output: print and save tonight's decisions
print(batch[["Transaction_ID", "Fraud_Probability", "Decision"]])
batch.to_csv("data/nightly_decisions.csv", index=False)
```

  - *How it works, line by line:*
    - `joblib.load(...)` brings back the trained pipeline: no training happens here, so this script runs in a fraction of a second.
    - `model.predict_proba(batch[features])[:, 1]` gives each new transaction its fraud probability (section 7.2). The feature columns must have the same names as in training.
    - `batch["Decision"] = "Pass"` sets every row to "Pass"; `batch.loc[batch["Fraud_Probability"] >= 0.5, "Decision"] = "Hold"` changes only the risky rows.
    - `batch.to_csv(...)` saves tonight's decisions for the operations team.
  - *Read the output:*

```
  Transaction_ID  Fraud_Probability Decision
0     N-TXN-9001              0.862     Hold
1     N-TXN-9002              0.070     Pass
2     N-TXN-9003              0.984     Hold
3     N-TXN-9004              0.083     Pass
4     N-TXN-9005              0.789     Hold
5     N-TXN-9006              0.163     Pass
6     N-TXN-9007              0.965     Hold
7     N-TXN-9008              0.073     Pass
```

    - 4 of 8 transactions are held, all foreign purchases above $2,800.
    - The probabilities fall into two clear groups: above `0.78` or below `0.17`. No case sits near the 0.5 cut-off, so the decisions are not borderline.

- *Quick Check 2 (Run date stamp):*
  - New tool: `from datetime import date` and `date.today()` give today's date. Why: every decision file must show when the decisions were made, for audits.

  - Task: add `from datetime import date` to part **1. Input**, and one line at the end of part **2. Process**: `batch["Run_Date"] = str(date._____())`
  - Check: the saved `data/nightly_decisions.csv` has a `Run_Date` column with today's date, written as `YYYY-MM-DD`.

- *Micro-Example 3 (Drift Check: Does Tonight Look Like Training?):* `YourName_S41_demo_3.py`

```
# 1. Input: training data and tonight's batch
import pandas as pd
train = pd.read_csv("data/session_03_04_dataset.csv")
batch = pd.read_csv("data/session_04_01_new_batch.csv")
features = ["Transaction_Amount_USD", "Foreign_IP_Flag",
            "Failed_Login_Attempts_Count"]

# 2. Process: compare the averages, feature by feature
drift = pd.DataFrame({
    "Train_Avg": train[features].mean().round(2),
    "Batch_Avg": batch[features].mean().round(2),
})
drift["Change_Pct"] = (
    (drift["Batch_Avg"] - drift["Train_Avg"]) / drift["Train_Avg"] * 100
).round(1)
drift["Drift_Flag"] = drift["Change_Pct"] > 20

# 3. Output: the drift report
print(drift)
```

  - *How it works, line by line:*
    - `train[features].mean()` and `batch[features].mean()` give each feature's average in the two files.
    - Putting both into `pd.DataFrame({...})` lines them up by feature name, one row per feature.
    - `Change_Pct` is the change in percent from the training average (section 7.3).
    - `Drift_Flag` marks features that changed by more than 20%.
  - *Read the output:*

	Train_Avg	Batch_Avg	Change_Pct	Drift_Flag
Transaction_Amount_USD	1371.72	2555.00	86.3	True
Foreign_IP_Flag	0.50	0.50	0.0	False
Failed_Login_Attempts_Count	1.56	1.75	12.2	False

    - Tonight's purchases are on average 86% larger than the training purchases: amount has drifted.
    - The share of foreign purchases is unchanged, and failed logins rose only slightly.
    - The model still runs, but its scores for large purchases deserve a person's review tonight.

- *Quick Check 3 (`.abs()`):*

- New tool: `column.abs()` removes the minus sign. Why: a feature that **falls** by 30% has drifted too; `> 20` alone misses it.
- Task: in part **2. Process**, change the flag line to `drift["Drift_Flag"] = drift["Change_Pct"].___() > 20`
- Check: the flags do not change for tonight's batch, but a change of `-30.0` would now be flagged as well.

10. Official Python Documentation & References

- Scikit-Learn Model Persistence Guide (Joblib Documentation).
- Python Library: `os.path` (Checking Files and Paths).
- MLflow Official Documentation: Open Source Platform for the Machine Learning Lifecycle.
- Sculley et al.: Hidden Technical Debt in Machine Learning Systems (NeurIPS Standards).

11. Exercise 1: Guided Practice - Read the Run Log

- *Step 1: Open Script:* In Google Antigravity IDE, create a new script named `YourName_S41_monitoring.py`, replacing `YourName` with your own name (for example, `Reacher_S41_monitoring.py`).
- *Step 2: Find the Nights That Need Attention:* Type the following code:

```
# 1. Input: the run log, 5 nights x 4 stages
import pandas as pd
log = pd.read_csv("data/session_04_01_dataset.csv")

# 2. Process: health rules for the Scoring stage, then totals per night
scoring = log[log["Pipeline_Stage"] == "Scoring"].copy()
scoring["Alert"] = (
    (scoring["Latency_Ms"] > 900) | (scoring["Model_Accuracy"] < 0.93)
)
nightly = log.groupby("Run_Date").agg(
    Total_Latency_Ms=("Latency_Ms", "sum"),
    Transactions_In=("Records_Count", "first"),
)

# 3. Output: the Scoring stage per night, then the nightly totals
print(scoring[["Run_Date", "Latency_Ms", "Model_Accuracy", "Alert"]])
print(nightly)
```

  - `.copy()` makes `scoring` an independent table, so adding the `Alert` column does not touch `log`.
  - `|` means "or": a night is flagged if either rule is broken.
  - `"first"` in `.agg()` takes the first value of each group; here, the `Ingestion` row's count.

- *Step 3: Interpret:* In two comment lines, describe the trend in latency and accuracy across the 5 nights, and what the team should do next.

- *Step 4: Execute in Terminal:* Run `python YourName_S41_monitoring.py` in Panel `[4]` and confirm return code `0`.
- *Success Check:* alerts on `2024-12-04` (latency `910`, accuracy `0.921`) and `2024-12-05` (`980`, `0.904`); total latency is `1470` ms on the first two nights, then rises every night to `1730` ms, while incoming transactions grow from `120000` to `138900`. Growing volume and falling accuracy together suggest drift: the model should be reviewed and possibly retrained.

12. Exercise 2: Applied Practice - Survive a Missing Model File

- *Coding Challenge:* In `YourName_S41_practice.py`, make the nightly job keep working when the model file cannot be found.
- *Task 1 (Read the FileNotFoundError & AI Diagnosis):* Starting from Micro-Example 2, change the path to `"models/fraud_modle.joblib"` (a typo). The script stops with `FileNotFoundError: [Errno 2] No such file or directory: 'models/fraud_modle.joblib'`. Capture the traceback with `Win + Shift + S`, paste it into Panel `[3]` (AI Agent Chat) with `Ctrl + V`, and ask how to check for a file before loading it.
- *Task 2 (Add a Fallback Rule):* Store the path in a variable `model_path`. If `os.path.exists(model_path)`, load the model and decide as in Micro-Example 2, and set `method = "model"`. Otherwise, use the fallback rule: hold when `Foreign_IP_Flag == 1` **and** `Failed_Login_Attempts_Count >= 2`, and set `method = "fallback rule"`. Print which method was used.
- *Task 3 (Compare):* Run the script once with the typo and once with the correct path. Print the held transaction IDs each time.
- *Success Check:* with the typo, the fallback rule holds 3 transactions (`N-TXN-9001`, `N-TXN-9003`, `N-TXN-9007`); with the correct path, the model holds 4 (it also holds `N-TXN-9005`, a foreign purchase of `$3,900` with only 1 failed login). The script finishes with return code `0` both times: the fallback keeps the business safe, but it is weaker than the model, so the missing file must still be fixed quickly.

13. Exercise 3: Advanced Practice - One Function for the Whole Night

- *Goal:* Combine everything into one function that the operations team could schedule to run every night: validate, check drift, score (or fall back), act, and report.
- *What you need:*
  - Files: `data/session_03_04_dataset.csv` (training), `data/session_04_01_new_batch.csv` (tonight), and `models/fraud_model.joblib` (from Micro-Example 1).
  - Script name: `YourName_S41_challenge.py`.
  - Tools you already know: functions (Session 2.4), `assert` (Session 2.3), `os.path.exists()`, `joblib.load()`, `.predict_proba()`, the drift table from Micro-Example 3, and the fallback rule from Exercise 2.
- *Requirements (follow them in order):*
  1. Write `run_nightly(batch_path, model_path, train_path)` that returns a dictionary summary.
  2. **Validation:** check with `assert` that the batch has the three feature columns and no missing values in them.

3. **Drift:** compute the change in percent of each feature's average, as in Micro-Example 3, and collect the names of the features whose absolute change is above 20.
4. **Scoring:** if the model file exists, hold transactions with a fraud probability of 0.5 or above; otherwise, use the fallback rule from Exercise 2.
5. **Action:** save the batch with `Decision` and `Method` columns to `data/nightly_decisions.csv`.
6. **Report:** return `{"records": ..., "held": ..., "method": ..., "drift_features": [...], "needs_review": ...}`, where `needs_review` is `True` if any feature drifted **or** the fallback was used.
7. Call the function twice, once with the correct model path and once with a wrong path, and print both summaries.

- *New tools for this exercise:*
  - Checking that columns exist: `set(features).issubset(batch.columns)` is `True` when every name in `features` is a column of `batch`.
  - A list of names from a Boolean column: `list(drift.index[drift["Drift_Flag"]])` keeps the row labels (feature names) where the flag is `True`.
  - Any `True` in a list: `len(drift_features) > 0`.
- *Hints:*
  - Write and test each stage as a few lines first, then move them into the function.
  - Use `int(...)` around counts so the dictionary prints plain numbers.
- *Self-Check (expected results):*
  - Correct path: `{"records": 8, "held": 4, "method": "model", "drift_features": ["Transaction_Amount_USD"], "needs_review": True}`.
  - Wrong path: `{"records": 8, "held": 3, "method": "fallback rule", "drift_features": ["Transaction_Amount_USD"], "needs_review": True}`.
  - Business meaning: even when the model runs, tonight still needs a person's review, because the amounts are far outside the training data. A decision system should report its own doubts, not just its decisions.