DUMPSARENA

Databricks Certified Machine Learning Professional

Databricks Databricks-Machine-Learning-Professional

Version Demo

Total Demo Questions: 10

Total Premium Questions: 60

Buy Premium PDF

https://dumpsarena.co

sales@dumpsarena.co

sales@dumpsarena.co
dumpsarena.co

QUESTION NO: 1

A machine learning engineer wants to view all of the active MLflow Model Registry Webhooks for a specific model.

They are using the following code block:

```
from mlflow.utils.rest_utils import http_request
endpoint = "/api/2.0/mlflow/registry-webhooks/list/?model_name
response = http_request(
    host_creds=host_creds,
    endpoint=endpoint,
    method="POST"
)
```

Which of the following changes does the machine learning engineer need to make to this code block so it will successfully accomplish the task?

A. There are no necessary changes

B. Replace list with view in the endpoint URL

C. Replace POST with GET in the call to http request

D. Replace list with webhooks in the endpoint URL

E. Replace POST with PUT in the call to http request

ANSWER: D

Explanation:

To view (list) the active Model Registry webhooks for a registered model, you must call the Model Registry Webhooks “list” endpoint. In the Databricks REST API, webhooks are managed under the `/api/2.0/mlflow/registry-webhooks` resource, and listing webhooks is done via `GET /api/2.0/mlflow/registry-webhooks/list` with a query parameter such as `model_name`. If the provided code is using an endpoint containing `.../list` but not the correct resource path (e.g., it uses something like `.../webhooks/list` or another incorrect segment), the fix is to replace the incorrect segment with `registry-webhooks` (i.e., “webhooks” in the resource name), not to change HTTP verbs arbitrarily.

Option D is correct because the endpoint must reference the webhooks resource; without that, the request won’t reach the registry webhooks API. Option A is wrong because the original code (as implied by the answer choices) is not pointing at the correct endpoint. Option B is wrong because “view” is not an API resource name. Options C and E are wrong because listing is a GET operation; changing POST to GET/PUT would not fix an incorrect endpoint path and PUT is not used for listing.

References: Databricks REST API: Registry webhooks, Databricks Model Registry webhooks

QUESTION NO: 2

After a data scientist noticed that a column was missing from a production feature set stored as a Delta table, the machine learning engineering team has been tasked with determining when the column was dropped from the feature set.

Which of the following SQL commands can be used to accomplish this task?

A. VERSION

B. DESCRIBE

C. HISTORY

D. DESCRIBE HISTORY

E. TIMESTAMP

ANSWER: D

Explanation:

To determine *when* a column was dropped from a Delta table, you need to inspect the table's transaction log (commit history). Delta Lake exposes this via `DESCRIBE HISTORY`, which returns a row per commit with the version, timestamp, operation (for example, `ALTER TABLE`), and operation parameters. By scanning the history output around the time the schema changed (or by correlating versions with schema snapshots), the team can identify the commit where the column was removed and the exact timestamp of that change.

The other options are not valid standalone SQL commands for this purpose in Databricks SQL/Delta Lake. `VERSION` and `TIMESTAMP` are not commands; they are concepts used with time travel (for example, `VERSION AS OF` / `TIMESTAMP AS OF`) but don't directly list when changes occurred. `DESCRIBE` shows current metadata/schema, not the sequence of changes. `HISTORY` alone is not the Delta SQL command; the supported syntax is `DESCRIBE HISTORY`.

References: Databricks Delta table history (DESCRIBE HISTORY), Delta Lake documentation: Table history.

QUESTION NO: 3

A machine learning engineer needs to deliver predictions of a machine learning model in real-time. However, the feature values needed for computing the predictions are available one week before the query time.

Which of the following is a benefit of using a batch serving deployment in this scenario rather than a real-time serving deployment where predictions are computed at query time?

A. Batch serving has built-in capabilities in Databricks Machine Learning

B. There is no advantage to using batch serving deployments over real-time serving deployments

C. Computing predictions in real-time provides more up-to-date results

D. Testing is not possible in real-time serving deployments

E. Querying stored predictions can be faster than computing predictions in real-time

ANSWER: E

Explanation:

Because the required feature values are known a week ahead of time, you can precompute predictions in advance and store them (for example in a Delta table) for fast lookup at request time. The key benefit of a batch-serving approach here is lower online latency and reduced compute at query time: the “serving” path becomes a simple read of already-materialized predictions rather than running feature retrieval + model inference on every request. This is especially valuable when you still need to respond in real time but don’t actually need real-time feature freshness.

Option E captures this: querying stored predictions is often much faster and more predictable than computing predictions on demand, and it can also improve reliability under traffic spikes.

Option A is not a reliable “benefit” statement: while Databricks supports batch inference patterns (e.g., via jobs/notebooks and writing results to Delta), the question asks for a scenario-driven advantage over real-time inference, not a generic product capability claim. Option B is incorrect because there is a clear advantage here (precomputation). Option C describes a potential advantage of real-time inference, but it doesn’t apply when features are only available in advance. Option D is false; real-time endpoints can be tested like any other service.

References: Databricks Model Serving, Databricks model inference (batch/online patterns)

QUESTION NO: 4

A machine learning engineer wants to move their model version model_version for the MLflow Model Registry model model from the Staging stage to the Production stage using MLflow Client client.

Which of the following code blocks can they use to accomplish the task? A)

```
client.transition_model_version_stage(
    name=model,
    version=model_version,
    stage="Staging"
)
```

B)

```
client.transition_model_stage(
    name=model,
    version=model_version,
    stage="Production"
)
```

C)

```
client.transition_model_version_stage(
    name=model,
    version=model_version,
    stage="Production"
)
```

D)

```
client.transition_model__stage(
    name=model,
    version=model_version,
    from="Staging",
    to="Production"
)
```

E)

```
client.transition_model_version_stage(
    name=model,
    version=model_version,
    from="Staging",
    to="Production"
)
```

A. Option A

B. Option B

C. Option C

D. Option D

E. option E

ANSWER: A

Explanation:

To move an MLflow Model Registry version from `Staging` to `Production` using an `MlflowClient`, you use the client's stage-transition API: `client.transition_model_version_stage(name=model, version=model_version,`

`stage="Production", ...)`. This is the canonical MLflow Registry operation for promoting a specific registered model version to a new stage. Option A is the only choice that corresponds to this required action (i.e., it transitions a particular version of a registered model to the `Production` stage via the MLflow client).

The other options are incorrect because they typically (a) call non-existent/incorrect client methods, (b) attempt to set stage via model URI or loading APIs (which do not change registry stage), or (c) use the wrong parameters/objects (for example, confusing run IDs with model versions, or using registry search/update calls that don't perform stage transitions). In MLflow, stage changes are explicit registry operations; simply registering, loading, or tagging a model does not promote it between stages.

References: MLflow Python API: MlflowClient, MLflow Model Registry.

QUESTION NO: 5

Which of the following describes the purpose of the context parameter in the predict method of Python models for MLflow?

A. The context parameter allows the user to specify which version of the registered MLflow Model should be used based on the given application's current scenario

B. The context parameter allows the user to document the performance of a model after it has been deployed

C. The context parameter allows the user to include relevant details of the business case to allow downstream users to understand the purpose of the model

D. The context parameter allows the user to provide the model with completely custom if-else logic for the given application's current scenario

E. The context parameter allows the user to provide the model access to objects like preprocessing models or custom configuration files

ANSWER: E

Explanation:

In MLflow's PythonModel (pyfunc) flavor, the `predict(self, context, model_input)` method receives a `context` object that provides access to artifacts and other runtime information needed for inference. Most importantly, `context.artifacts` contains the local filesystem paths to any artifacts that were logged with the model (e.g., a preprocessing transformer, label encoder, vocabulary file, or a custom config). This enables the model's prediction code to load and use those dependencies at inference time in a portable way.

Option E matches this purpose precisely. Option A is incorrect because model version selection is handled by how you load the model (URI, stage/alias), not via the `context` parameter. Option B is about monitoring/observability, which is outside `predict`'s context object. Option C describes documentation/metadata, which belongs in model descriptions, tags, or the Model Registry, not `context`. Option D is misleading: while you can implement arbitrary logic inside `predict`, `context` is not specifically for "custom if-else logic"; it's for accessing artifacts and environment information.

References: MLflow pyfunc (PythonModel) documentation, MLflow Models documentation.

QUESTION NO: 6

A machine learning engineer wants to move their model version model_version for the MLflow Model Registry model model from the Staging stage to the Production stage using MLflow Client client. At the same time, they would like to archive any model versions that are already in the Production stage.

Which of the following code blocks can they use to accomplish the task? A)

```
client.transition_model_version_stage(
    name=model,
    version=model_version,
    stage="Archived"
)
client.transition_model_version_stage(
    name=model,
    version=model_version,
    stage="Production"
)
```

B)

C)

```
client.transition_model_stage(
    name=model,
    version=model_version,
    stage="Production",
    archive_existing_versions=True
)
```

D)

```
client.transition_model_version_stage(
    name=model,
    version=model_version,
    stage="Production",
    archive_existing_versions=True
)
```

A. Option A

B. Option B

C. Option C

D. Option D

ANSWER: C

Explanation:

To move a registered model version to `Production` and simultaneously archive any existing versions already in `Production`, the MLflow Model Registry API provides the `archive_existing_versions` flag on `MlflowClient.transition_model_version_stage`. The correct code block is the one that calls:

```
client.transition_model_version_stage(name=model, version=model_version,
stage="Production", archive_existing_versions=True)
```

This performs the stage transition for the specified version and automatically transitions any other versions currently in `Production` to `Archived`, which is exactly what the prompt asks for. Option C is the only choice that matches this intended usage.

The incorrect options typically fail for one of these reasons: they omit `archive_existing_versions=True` (so existing Production versions remain active), they use the wrong API/method name, or they attempt to "archive" by setting an invalid stage value or by manipulating stages in multiple steps that don't guarantee atomicity/consistency.

References: MLflow Python API: MlflowClient, MLflow Model Registry.

QUESTION NO: 7

A data scientist has created a Python function compute_features that returns a Spark DataFrame with the following schema:

```
customer_id STRING,
spend DOUBLE,
units INT,
loyal INT,
region STRING
```

The resulting DataFrame is assigned to the features_df variable. The data scientist wants to create a Feature Store table using features_df.

Which of the following code blocks can they use to create and populate the Feature Store table using the Feature Store Client fs?

A)

```
fs.create_table(
    name="new_table",
    primary_keys="customer_id",
    df=features_df,
    description="Customer features"
)
```

B)

```
fs.create_table(
    name="new_table",
    primary_keys="customer_id",
    description="Customer features"
)
```

C)

features_df.write.mode("fs").path("new_table") D)

A. Option A

B. Option B

C. Option C

D. Option D

E. features_df.write.mode("feature").path("new_table")

ANSWER: D

Explanation:

To create and populate a Databricks Feature Store table from an existing Spark DataFrame, you use the Feature Store client's `create_table` method (to define the table name, primary key(s), schema, and optional partition columns) and then write data into it using `fs.write_table` (or `fs.write_table(..., mode="overwrite")` depending on desired behavior). Option D is the only choice that matches this supported workflow: it uses the Feature Store client (`fs`) to create the feature table and then writes `features_df` into that table.

The other options are incorrect because they attempt to use the Spark DataFrame writer with non-existent modes like `"fs"` or `"feature"`, or otherwise don't use the Feature Store client APIs required to register a Feature Store table and manage

metadata (primary keys, feature lookups, etc.). In Databricks Feature Store, you don't create a feature table by calling `DataFrame.write.mode("fs")`; you must use the Feature Store client methods.

References: Databricks Feature Store documentation, Work with feature tables (create/write).

QUESTION NO: 8

A machine learning engineer is attempting to create a webhook that will trigger a Databricks Job job_id when a model version for model model transitions into any MLflow Model Registry stage. They have the following incomplete code block:

```
job_json = {
    "model_name": model,
    "events": [_____],
    "description": "Job webhook trigger",
    "status": "Active",
    "job_spec": {
        "job_id": job_id,
        "workspace_url": url,
        "access_token": token
    }
}

response = http_request(
    host_creds=host_creds,
    endpoint=endpoint,
    method="POST",
    json=job_json
)
```

Which of the following lines of code can be used to fill in the blank so that the code block accomplishes the task?

A. "MODEL_VERSION_CREATED"

B. "MODEL_VERSION_TRANSITIONED_TO_PRODUCTION"

C. "MODEL_VERSION_TRANSITIONED_TO_STAGING"

D. "MODEL_VERSION_TRANSITIONED_STAGE"

E. "MODEL_VERSION_TRANSITIONED_TO_STAGING", "MODEL_VERSION_TRANSITIONED_TO_PRODUCTION"

ANSWER: D

Explanation:

To trigger a Databricks Job when a model version transitions into *any* MLflow Model Registry stage, you must subscribe the webhook to the generic “stage transitioned” event, not to a specific target stage like Staging or Production. MLflow/Databricks model registry webhooks support an event type that fires whenever the stage changes (e.g., None → Staging, Staging → Production, Production → Archived). Therefore, the correct value to fill in the blank is the event type that represents “model version transitioned stage”.

Option D is correct because it corresponds to the generic stage-transition event, which will fire for transitions into any stage. Options B and C are incorrect because they only fire when transitioning specifically to Production or specifically to Staging, respectively, and would miss other stage transitions (e.g., to Archived). Option A is incorrect because model version creation is not the same as a stage transition. Option E is incorrect in a single-choice context and also wouldn’t cover transitions to other stages beyond Staging/Production.

References: Databricks documentation: Model registry webhooks, MLflow documentation: Model Registry.

QUESTION NO: 9

Which of the following deployment paradigms can centrally compute predictions for a single record with exceedingly fast results?

A. Streaming

B. Batch

C. Edge/on-device

D. None of these strategies will accomplish the task.

E. Real-time

ANSWER: E

Explanation:

The correct choice is **Real-time**. Real-time (online) inference is designed to serve predictions for *individual* requests with very low latency (often milliseconds) from a *central* service endpoint (e.g., a model serving API). This matches the prompt’s key requirements: “centrally compute” and “single record” with “exceedingly fast results.” In Databricks, this aligns with online serving patterns such as Databricks Model Serving, where a deployed model is queried per request and returns immediate predictions.

Streaming is typically used for continuous processing of event streams (micro-batches/continuous pipelines). While it can score events quickly, it’s not primarily a request/response paradigm for a single record with strict online latency expectations. **Batch** is explicitly for offline scoring of many records at once and is not suited to single-record, low-latency needs. **Edge/on-device** can be extremely fast for single-record inference, but it is the opposite of “centrally compute” because inference happens locally on the device. Therefore it doesn’t satisfy the “central” requirement. **None of these** is incorrect because real-time serving does accomplish the task.

References: Databricks Model Serving documentation, MLflow model deployment concepts.

QUESTION NO: 10

Which of the following is a simple statistic to monitor for categorical feature drift?

A. Mode

B. None of these

C. Mode, number of unique values, and percentage of missing values

D. Percentage of missing values

E. Number of unique values

ANSWER: C

Explanation:

For categorical feature drift, a common “simple statistics” approach is to track summary properties of the category distribution over time. In practice, this includes monitoring the *mode* (most frequent category), the *number of unique values* (cardinality, which can change if new categories appear or old ones disappear), and the *percentage of missing values* (which can indicate upstream pipeline/data quality changes). Option C is correct because it bundles several lightweight, easy-to-compute indicators that together provide a more reliable signal of drift than any single statistic alone.

Option A (mode) by itself can miss drift when the top category stays the same but the rest of the distribution shifts. Option D (missing %) is important but is more of a data quality/availability signal and does not capture distributional changes among non-missing categories. Option E (unique values) can detect new categories but won’t detect shifts in frequency among existing categories. Option B is incorrect because there are indeed simple statistics commonly used for categorical drift monitoring.

For deeper drift detection, you’d typically compare full distributions (e.g., PSI, chi-square, KL divergence), but the question asks specifically for a simple statistic set. See: Databricks MLflow Model Monitoring and Microsoft guidance on model/data drift.