Spark Project Readiness Assessment (Beginner → Deployment Gauge)

Audience: Trainees with basic exposure to Python/Spark.
Goal: Decide if a beginner can be safely deployed to real projects.
What we assess: Foundations, Python coding efficiency, Spark fundamentals & architecture, hands-on data processing, troubleshooting, and delivery hygiene.

---

1) Outcomes & Decision Bands

- **Deploy Now** (≥ 80/100 **and** all gates passed): Can own low/medium-complexity Spark tasks with mentorship for design reviews.
- **Deploy with Mentorship** (65–79/100, gates passed): Pair on critical paths; independent on well-scoped tickets.
- **Not Yet** (< 65/100 or any gate failed): Remediation plan before deployment.

Hard Gates (must pass): 1. Environment set up end-to-end (local + cluster access) and reproducible. 2. Code compiles, runs, and is version-controlled with a clear README & configs. 3. Spark job completes on the baseline dataset without errors/OOM.

---

2) Assessment Flow (Step-by-Step)

Duration: ~1–2 days total hands-on time.

1. **Stage 0 – Environment & Hygiene (Gate)**
2. Install: Python 3.10+, Spark 3.x/4.x, Java 11/17, IDE, Git.
3. Prove: `spark-shell`/`pyspark` start; `spark-submit` runs a hello-world job.
4. Repo layout, virtual env, `.env`/config separation, logging enabled, unit test scaffold.
5. **Stage 1 – Python Foundations (Core Skills)**
6. Write 3 small utilities (functions/classes) + simple unit tests.
7. Data wrangling exercise using pandas or pure Python for CSV/JSON.
8. **Stage 2 – Spark Fundamentals & Architecture (Knowledge Check)**
9. Short quiz + whiteboard: DAG → stages → tasks, shuffle, partitions, joins, Catalyst/Tungsten, broadcast/cache, Spark UI, deploy modes.
10. **Stage 3 – Spark Core + SQL (Hands-On)**
11. Build a small ETL: read → clean → transform (joins, windows) → write partitioned Parquet.
12. Show Spark UI screenshots; explain plan; justify optimizations.

13. **Stage 4 – Troubleshooting Lab**

14. Fix an intentionally broken job (schema drift, skew, nulls, small files, OOM). Document root cause + fix.

15. **Stage 5 – Mini-Project Capstone**

16. End-to-end pipeline with configs, tests, and a `spark-submit` command. Short retro write-up.

---

3) Scoring Weights (100 pts)

- Stage 0: Environment & Hygiene – **Gate only** (no points; fail → stop)
- Stage 1: Python Foundations – **20 pts**
- Stage 2: Spark Fundamentals & Architecture – **20 pts**
- Stage 3: Spark Core + SQL (Hands-On) – **35 pts**
- Stage 4: Troubleshooting Lab – **15 pts**
- Stage 5: Mini-Project Delivery & Hygiene – **10 pts**

Readiness Bands: Deploy Now ≥80; Deploy w/ Mentorship 65–79; Not Yet <65.
Bonus (up to +5): Clean code comments, meaningful docstrings, thoughtful performance notes.

---

4) Metrics & Rubrics

4.1 Python Foundations (20 pts)

Tasks - Implement: `parse_config(path) -> dict` (supports JSON/YAML, env overrides). - Implement: `dedupe_records(records, keys)` preserving original order. - Implement: simple quality checks (null %, unique keys, value ranges) with unit tests.

Metrics - **Correctness (8 pts):** All provided unit tests pass; edge cases handled (empty, nulls, bad types). - **Complexity & Efficiency (4 pts):** Avoid $O(n^2)$ where $O(n \log n)$ is feasible; space usage reasonable. - **Readability & Style (4 pts):** PEP8, naming, small functions, type hints. - **Testing (4 pts):** ≥80% coverage on utility module; clear test names; arrange-act-assert format.

Thresholds - Must pass all reference tests; < 5 lint warnings per 200 LOC.

4.2 Spark Fundamentals & Architecture (20 pts)

Format: 15–20 short questions + 2 whiteboard prompts.

What good looks like - Explains: driver vs executor, jobs→stages→tasks, narrow vs wide transformations. - Can read Spark UI: identify shuffle stages, skewed tasks, spilled bytes. - Knows when to use: broadcast join, cache/persist levels, coalesce vs repartition. - Understands: Catalyst optimizer (logical→physical plan), Tungsten/codegen, AQE basics.

Rubric - **Concept Accuracy (12 pts):** Definitions & trade-offs correct. - **Diagnosis Reasoning (4 pts):** Given a DAG/UI snippet, states likely bottleneck. - **Architecture Mapping (4 pts):** Maps deploy mode (client/cluster) and cluster manager (YARN/K8s/Standalone) to where driver/executors live.

Thresholds - ≥ 70% quiz; both whiteboards coherent (not necessarily perfect).

4.3 Spark Core + SQL – Hands-On (35 pts)

Dataset (provide to trainee):
- `transactions` (~5–10M rows synthetic): id, user_id, ts, amount, status, merchant, category.
- `users_dim` (~100k): user_id, join_date, city, plan.

Required Pipeline 1. Ingest CSV/JSON with schema (no schema-on-read), bad records to quarantine. 2. Clean: parse timestamps, trim strings, handle nulls, enforce types. 3. Transform: - Join `transactions` ↔ `users_dim` (broadcast if small). - Compute 7/30-day rolling sums per user (window). - Flag anomalies (amount > p99 per user plan). 4. Write: partitioned Parquet by `dt=YYYY-MM-DD`, overwrite dynamic partitions.

Deliverables - Code (PySpark or Scala), `application.conf`, `logging.conf`, `run_local.sh`, `spark-submit` cmd. - Query plan (`df.explain("extended")`) and 2 Spark UI screenshots (stages, SQL tab).

Metrics - **Correctness (12 pts):** Row counts per step, null policy enforced, dedupe rules applied. - **Performance (10 pts):** - Uses predicate/file pruning, avoids unnecessary UDFs. - Broadcast join when `users_dim < 100MB` OR justified alternative. - Shuffle stages minimized; skew handled if present. - **Resource Use (5 pts):** No OOM; spill reduced; sensible `spark.sql.shuffle.partitions`. - **Code Quality (5 pts):** Modular jobs, params via config/args; logs key metrics. - **Reproducibility (3 pts):** One-command run; deterministic outputs.

Thresholds - Pipeline completes within **2×** assessor baseline on provided machine/cluster. - ≤ 3 shuffle stages unless justified; job succeeds on 2× input scale.

4.4 Troubleshooting Lab (15 pts)

Broken Scenarios (assessor provides any 3): 1. **Schema drift**: unexpected column, wrong type → job fails.
2. **Data skew**: a few keys dominate → stage stragglers.
3. **Small files**: 1000 tiny partitions in lake → slow reads.
4. **Out-of-memory**: collect/action misuse; huge shuffle; wrong cache level.
5. **Bad joins**: exploding row counts due to duplicate keys.

Candidate Tasks - Identify root cause using logs + Spark UI.
- Implement fix (e.g., `broadcast`, salting, `coalesce/repartition`, combine small files, map-side pre-agg, filter early, checkpointing).

Rubric - **Diagnosis (6 pts):** Reads UI metrics (task time, input/shuffle read, spill).
- **Fix Quality (6 pts):** Correct and minimally invasive; measures before/after.
- **Communication (3 pts):** Clear incident note with cause→change→result.

Thresholds - Fix ≥ **2 of 3** scenarios to passing quality; each shows measurable improvement.

4.5 Mini-Project Delivery & Hygiene (10 pts)

Requirements - `README` with problem statement, data contract, run steps, config matrix.
- `tests/` with at least 5 unit tests for pure functions and 2 job-level assertions.
- Parameterized job (env, date range).
- Outputs: partitioned Parquet + data quality report (row counts, null %, distinct keys, p95/p99).

Rubric - **Repo Hygiene (3 pts):** Clear structure, `.gitignore`, versioned configs; no secrets.
- **Testing & CI stub (3 pts):** Local test run + placeholder CI script.
- **Observability (2 pts):** Logs with run id, input/output counts, duration.
- **Docs (2 pts):** Assumptions, limitations, and next steps.

---

5) Student Expectation Guide (share with trainees)

What we'll measure: - You can write clean Python utilities with tests. - You understand Spark's execution model and can read the Spark UI. - You can build an ETL that joins, aggregates (window), and writes partitioned data efficiently. - You can diagnose/fix common Spark issues (skew, small files, schema drift, OOM). - Your work is reproducible, documented, and version-controlled.

Coding Standards - Prefer built-ins & Spark SQL functions over UDFs; if UDFs, justify and unit-test.
- Avoid `collect()`/`show()` on large data; sample with `limit()`/`sample()`. - Log input/output counts, partitions, and plan hints at INFO level. - Parameterize paths, dates, and thresholds via config/args; no hard-coded credentials.

Performance Principles - Prune early (columns/rows).
- Broadcast small dims; filter before join; aggregate before shuffle.
- Coalesce writes to avoid tiny files; partition on query-friendly columns.
- Cache only when reused; use correct storage levels; unpersist.

---

6) Evaluation Templates

6.1 Scoring Sheet (per candidate)

Area	Max	Score	Notes
Python Foundations	20		
Spark Fundamentals & Architecture	20		
Spark Core + SQL (Hands-On)	35		

Area	Max	Score	Notes
Troubleshooting Lab	15		
Mini-Project Delivery & Hygiene	10		
Total	**100**		
Decision (Deploy Now / Deploy w/ Mentorship / Not Yet)			

6.2 Run Log (attach to PR/README)

- Dataset size (rows/GB):
- Cluster/machine spec:
- `spark-submit` args:
- Stage metrics (from UI): input rows, shuffle read/write, spill, skewed tasks.
- Job duration (first run / optimized run):
- Notes & follow-ups:

6.3 Incident Note (for Troubleshooting)

- **Symptom:**
- **Root Cause:**
- **Fix Applied:**
- **Evidence:** (plan/UI metrics before/after)
- **Residual Risk:**

7) Assessor Pack (what you provide)

- Baseline dataset generator or CSV dumps (transactions + users).
- Reference config + baseline `spark-submit` (with partitions).
- Broken job variants for the lab (toggle via config flags).
- Answer key for fundamentals quiz (not shared with trainees until after).

8) Remediation Plan (if Not Yet)

- **Week 1:** Python drills (functions, tests, file IO), re-attempt Stage 1.
- **Week 2:** Spark foundations deep-dive + Spark UI reading workshop, re-quiz.
- **Week 3:** Guided ETL rebuild with performance checklist, re-attempt Stage 3.
- **Exit:** Redo Troubleshooting Lab with new scenarios.

9) Quick Reference (Glossary)

- **Driver/Executor:** Orchestration vs work processes.
- **Job/Stage/Task:** Execution hierarchy; stage = shuffle boundary.
- **Narrow/Wide Transformations:** Whether shuffle occurs.
- **Broadcast Join:** Send small table to all executors.
- **AQE:** Adapts joins/partitions at runtime.

- **Skew:** Uneven key distribution causing stragglers.
- **Small Files:** Many tiny output files harm read performance.

Appendix A – Suggested Repo Skeleton

```
project-root/
  src/
    main/python/etl/
      __init__.py
      jobs/
        daily_enrich.py
      utils/
        io.py
        quality.py
        config.py
  tests/
    test_quality.py
  conf/
    application.conf
    logging.conf
  scripts/
    run_local.sh
  README.md
  requirements.txt (or pyproject.toml)
  .gitignore
```

Appendix B – Sample `spark-submit`

```
spark-submit \
  --master local[*] \
  --deploy-mode client \
  --conf spark.sql.shuffle.partitions=200 \
  --conf spark.sql.adaptive.enabled=true \
  --conf spark.sql.broadcastTimeout=300 \
  src/main/python/etl/jobs/daily_enrich.py \
  --date 2025-08-01 --env local --config conf/application.conf
```

Use this document as both the candidate handout (expectations) and the assessor checklist (rubrics + scoring).