# DevOps for Data: Delivering & Orchestrating Apache Spark on Containers

**CD Foundation / Linux Foundation Workshops**  
**Speaker:** Muhammad Danyal “Sage” Khan  
**Date:** August 28

### Talking script

“Hi everyone, I’m Sage. In the next 30 minutes we’ll turn Spark jobs into **versioned, testable releases** you can ship with the same confidence as application code. First half is the ‘why’ and the blueprint; second half is a live, **local-only** demo on my laptop using kind. You’ll walk away with a starter repo, a repeatable delivery pattern, and a production checklist you can apply at your company.”

## Why DevOps/CD for Spark (not “just Kubernetes”)

* Spark powers **business-critical** data products, yet many jobs bypass CI/CD guardrails.
* We need **immutable artifacts**, promotion across environments, rollbacks, and SLOs—**just like app teams**.
* **Containers** → reproducible runtime. **CD** → trusted, auditable delivery. **CDEvents** → end‑to‑end traceability.

**Speaker notes:** Frame this as *DevOps for data*. Kubernetes is the concrete runtime, but the story is CD-first.

### Talking script

“Quick show of hands—or drop it in chat—who has a Spark job that ‘only works on one machine’? That’s our starting pain. Data teams often ship notebooks or scripts outside the governance that app teams enjoy. The result: drift, unknown dependencies, and fragile releases. **DevOps for data** means we treat Spark jobs as products: versioned images, tests, promotion rules, rollbacks, and SLOs. Kubernetes will be our **example runtime**, but the star of the show is **continuous delivery**—a repeatable, auditable way to get from commit to a safe production run.”

## CD Blueprint for Spark (at a glance)

**Flow:** commit → CI build (image/jar) → tests (code+data+deps) → SBOM & sign → CD apply → observe & rollback

Git push ──▶ CI (build & test) ──▶ OCI Registry (image + SBOM + signature)  
 │  
 CDEvents: artifact.published  
 ▼  
 CD Pipeline (promote/apply)  
 ▼  
 Kubernetes Runtime: Spark Job / CronJob / SparkApplication  
 Driver Pod <──> Executor Pods | Logs/Metrics | Rollback

**CDF fit:** Tekton/Jenkins pipelines • CDEvents for triggers/notifications • Ortelius-style inventory & provenance.

### Talking script

“Here’s the movie in one frame. A commit triggers CI to **build a single immutable artifact**—usually a Docker image with Spark + your code. CI runs **code tests** and **data checks**. We produce an SBOM and optionally **sign** the image. A CD pipeline promotes that image to an environment by applying a **Job/CronJob/SparkApplication**. Everything emits **CDEvents** so we can answer: *who ran what, where, and with which bits?* If anything looks off—bad data, performance regression—we **roll back** by redeploying the previous, known-good image. That’s the backbone you can scale across teams.”

## Packaging & Quality Gates (Code + Data + Supply Chain)

* **Package** one job = one artifact (Docker image or jar/wheel) with pinned Spark & libs.
* **Code tests:** unit & integration (pytest/ScalaTest).
* **Data tests:** schema/contract checks, null/freshness thresholds (pandera/Great Expectations).
* **Security:** dep scan (Trivy), **SBOM** (Syft), image **sign** (Cosign), optional attestations (in‑toto/SLSA).
* **Policy-as-code** gates: build must pass all to promote.

### Talking script

“Treat each Spark job like a microservice: **one image per job** with pinned versions. Before it’s eligible to run anywhere, we gate it. First, **unit/integration tests**. Second, **data contracts**: schemas, nulls, ranges, and freshness using **pandera** or **Great Expectations**. Third, supply chain checks—**Trivy** for CVEs, **Syft** for SBOM, **Cosign** for signatures. Your pipeline becomes a set of **gates** that must go green; if any fails, it never reaches an environment. This is how we keep data quality and security from becoming afterthoughts.”

## Orchestration Targets (containers first)

* **Kubernetes (demo):** Spark driver/executors as pods; Job/CronJob; Spark Operator (SparkApplication CRD).
* **Other runtimes (concept):** OpenShift, Nomad, managed Spark; same CD pattern, different apply method.

**Speaker notes:** Emphasize portability of the CD blueprint across runtimes.

### Talking script

“Kubernetes is our **concrete** runtime because it’s widely available and open source. Spark runs the driver and executors as **pods**. For batch, a simple **Job** does the trick; for schedules, use **CronJob**; at scale, you might adopt the **Spark Operator**. If your company uses OpenShift or Nomad, great—the **CD pattern doesn’t change**. The only difference is how we apply manifests. This separation—**artifact is constant, apply is pluggable**—is what keeps you portable.”

## Live Demo Plan (15 min)

1. make kind-up → local cluster.
2. make build && (optional) pip install -r requirements-dev.txt && make test → image + checks.
3. make load → preload image into kind (no registry needed).
4. make deploy-local → run Spark job.
5. kubectl get pods -n spark -w and kubectl logs job/spark-pi -n spark -f → observe.
6. (Optional) tweak spark.executor.instances and re‑apply.
7. Cleanup make kind-down.

**Tip:** Pre-pull images to avoid long waits.

### Talking script (live narration)

“Alright, let’s ship a Spark job locally with **no paid services**. Step 1, I spin up a **kind** cluster and apply the namespace/RBAC. Step 2, I build the image—this bakes Spark + app code into **one artifact**. Optional: I run **pandera/GE** tests to gate data. Step 3, I **load** that image into kind so Kubernetes doesn’t need a registry. Step 4, I **deploy** a Kubernetes **Job**; the driver pod will create executors. I’ll watch pods come up, then tail logs until we see ‘**Pi is roughly…**’. If we have time, I’ll bump executors from 2→3 and rerun to show elasticity. Finally, we tidy up. This is the exact loop your CI/CD will automate.”

## Observability & SLOs (Production)

* **Metrics:** Spark → Prometheus/Grafana (job duration, stage time, shuffle read/write, executor CPU/mem).
* **Logs:** Centralize (ELK/Cloud logging) with correlation IDs.
* **SLOs:** job latency, success rate, cost/run; alert on error budget burn.

### Talking script

“In production, two questions matter: *Did it work?* and *Was it healthy?* Expose Spark **metrics** to Prometheus/Grafana—duration, stage times, shuffle I/O, executor CPU/memory. Centralize **logs** so on-call can jump from an alert to driver/executor logs quickly. Define **SLOs** like ‘99% of runs finish < 20 minutes’ and track **error budgets**. This turns ad‑hoc jobs into managed, reliable services.”

## Platform Guardrails (for big software orgs)

* **Multi‑tenancy:** Namespaces, ResourceQuota, LimitRange, admission policies.
* **Secrets:** Vault/ESO; KMS‑backed keys; least privilege service accounts.
* **Cost:** requests/limits, VPA for drivers, cluster autoscaler, spot/priority, TTL for finished jobs.
* **Release mgmt:** environments; immutable images; rollback via previous tag.

### Talking script

“Platform teams, this one’s for you. Carve the cluster with **namespaces** and apply **quotas/limits** so a single runaway job can’t starve others. Route **secrets** through Vault or ESO and apply least-privilege service accounts. For **cost**, set requests/limits, enable autoscaling, TTL finished jobs, and consider spot pools. Enforce **immutable images** and keep a prior tag handy for instant **rollback**. These guardrails let many teams share one platform safely.”

## Risks & Fixes (Field-tested)

* **Pods Pending:** account for node reserves & pod overhead; right‑size executor mem/cores; autoscale.
* **OOMKilled (Python/shuffle):** raise spark.executor.memoryOverhead (25–40%); ≤5 cores/executor.
* **Cold starts:** pre-pull; slim base layers.
* **Config drift:** GitOps overlays; one repo owns manifests.
* **Data regressions:** block deploy on data‑test failures; expose data SLOs.

### Talking script

“Here are the top five issues you’re likely to hit. **Pending pods** usually mean requests plus overhead don’t fit—shrink executors or scale nodes. **OOMKilled** on executors? Python and heavy shuffle need more **memoryOverhead** and fewer cores per executor. **Cold starts** are image pulls—pre‑pull or slim the image. **Config drift**—solve with GitOps and a single source of truth. And **data regressions**—fail the pipeline early with schema and freshness checks so bad data never ships.”

## Production Mapping & CTA

* **Local:** kind + Job + embedded spark-submit.
* **Prod:** EKS/GKE/AKS + Spark Operator or Argo Workflows; GitOps (Argo CD/Flux); Vault/ESO; autoscaling; SSO/RBAC; cost dashboards.

**Links (to share):** Template repo • Makefile commands • CDF DataOps WG Slack.

### Talking script

“Map the demo straight to prod: swap kind for **EKS/GKE/AKS**, swap Job for **SparkApplication** via the **Spark Operator**, and put GitOps in front so all changes go through pull requests. Add Vault/ESO for secrets, SSO/RBAC for access, and autoscaling for elasticity. Your action items: clone the starter repo, run the demo, and pilot one pipeline with your platform team in the next sprint.”

# Appendix (optional slides for Q&A)

* Minimal resource sizing cheat‑sheet and common configs.
* Example Tekton/Jenkins pipeline snippets.

# Template Repo README (drop into your repo root)

## DevOps for Data — Spark on Containers (Template)

**Purpose:** A minimal, production‑minded template to **package, test, sign, and deliver** Apache Spark jobs to container‑orchestrated platforms, with a fast local demo on kind.

### Repository Layout

repo/  
 ├─ spark-app/  
 │ ├─ app.py # Example PySpark job (Pi or wordcount)  
 │ ├─ requirements.txt # Py deps pinned  
 │ └─ tests/  
 │ ├─ test\_unit\_app.py # Unit tests  
 │ └─ test\_data\_contract.py # Simple data checks (pandera/ge)  
 ├─ docker/  
 │ └─ Dockerfile # Build Spark app image  
 ├─ k8s/  
 │ ├─ job-spark.yaml # K8s Job: runs driver which spawns executors  
 │ ├─ cronjob-spark.yaml # Nightly schedule example  
 │ ├─ rbac.yaml # ServiceAccount/RoleBinding for driver  
 │ └─ namespace.yaml # Optional: dedicated ns  
 ├─ ci/  
 │ ├─ github-actions/  
 │ │ └─ ci.yaml # Build, test, sbom, scan, sign (example)  
 │ └─ tekton/  
 │ ├─ pipeline.yaml # Build→Test→SBOM→Sign→Deploy  
 │ └─ tasks/\*.yaml # Kaniko, pytest, syft, cosign, kubectl  
 ├─ .gitignore  
 ├─ Makefile  
 └─ README.md (this file)

### Prerequisites

* Docker, kubectl, kind (or Minikube), Make
* Python 3.11, Java 11+
* Optional: **syft** (SBOM), **cosign** (signing), **trivy** (scan)

### Quickstart (local demo)

# 1) Cluster up  
make kind-up  
  
# 2) Build + test  
make build  
make test  
  
# 3) Generate SBOM + sign (optional)  
make sbom  
make sign # requires COSIGN\_EXPERIMENTAL=1 and a keypair  
  
# 4) Deploy and watch  
make deploy  
kubectl get pods -w  
kubectl logs job/spark-pi -f  
  
# 5) Schedule nightly (optional)  
make cron-deploy  
  
# 6) Tear down  
make kind-down

### Example PySpark Job (spark-app/app.py)

# app.py — simple Pi estimation using PySpark  
from pyspark.sql import SparkSession  
import random  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 spark = (SparkSession.builder.appName("spark-pi").getOrCreate())  
 sc = spark.sparkContext  
 n = 10\_000\_000  
 def inside(\_):  
 x, y = random.random(), random.random()  
 return 1 if x\*x + y\*y <= 1 else 0  
 count = sc.parallelize(range(n)).map(inside).reduce(lambda a,b: a+b)  
 pi = 4.0 \* count / n  
 print(f"Pi is roughly {pi}")  
 spark.stop()

### Unit & Data Tests (spark-app/tests/)

# test\_unit\_app.py — trivial example  
import importlib  
  
def test\_import\_app():  
 importlib.import\_module("app")

# test\_data\_contract.py — schema/quality smoke (replace with GE/pandera in prod)  
import pandas as pd  
  
def test\_data\_contract():  
 df = pd.DataFrame({"id":[1,2,3], "value":[10,20,30]})  
 assert df["value"].notnull().all()  
 assert df.shape[0] >= 3

### Dockerfile (docker/Dockerfile)

# Build a minimal Spark + Py deps image (example base)  
FROM apache/spark-py:v3.5.1  
WORKDIR /opt/app  
COPY spark-app/requirements.txt ./  
RUN pip install --no-cache-dir -r requirements.txt || true  
COPY spark-app/ ./  
# Keep Spark examples jar available via base image (for alternate demos)

### Kubernetes Manifests (k8s/)

**Namespace & RBAC (k8s/rbac.yaml)**

apiVersion: v1  
kind: Namespace  
metadata: { name: spark }  
---  
apiVersion: v1  
kind: ServiceAccount  
metadata: { name: spark-runner, namespace: spark }  
---  
apiVersion: rbac.authorization.k8s.io/v1  
kind: Role  
metadata: { name: spark-edit, namespace: spark }  
rules:  
- apiGroups: ["", "batch", "apps"]  
 resources: ["pods", "pods/log", "services", "configmaps", "secrets", "jobs"]  
 verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]  
---  
apiVersion: rbac.authorization.k8s.io/v1  
kind: RoleBinding  
metadata: { name: spark-edit, namespace: spark }  
subjects:  
- kind: ServiceAccount  
 name: spark-runner  
 namespace: spark  
roleRef:  
 kind: Role  
 name: spark-edit  
 apiGroup: rbac.authorization.k8s.io

**Job (k8s/job-spark.yaml)**

apiVersion: batch/v1  
kind: Job  
metadata:  
 name: spark-pi  
 namespace: spark  
spec:  
 ttlSecondsAfterFinished: 600  
 template:  
 spec:  
 restartPolicy: Never  
 serviceAccountName: spark-runner  
 containers:  
 - name: driver  
 image: ghcr.io/YOURORG/spark-app:{{GIT\_SHA}}  
 imagePullPolicy: IfNotPresent  
 command: ["/opt/spark/bin/spark-submit"]  
 args: [  
 "--master", "k8s://https://kubernetes.default.svc",  
 "--deploy-mode", "cluster",  
 "--conf", "spark.kubernetes.container.image=ghcr.io/YOURORG/spark-app:{{GIT\_SHA}}",  
 "--conf", "spark.kubernetes.namespace=spark",  
 "--conf", "spark.executor.instances=2",  
 "--conf", "spark.executor.memory=2g",  
 "--conf", "spark.executor.cores=1",  
 "local:///opt/app/app.py"  
 ]  
 resources:  
 requests: { cpu: "500m", memory: "512Mi" }  
 limits: { cpu: "1", memory: "1Gi" }

**CronJob (k8s/cronjob-spark.yaml)**

apiVersion: batch/v1  
kind: CronJob  
metadata:  
 name: spark-pi-nightly  
 namespace: spark  
spec:  
 schedule: "0 2 \* \* \*"  
 jobTemplate:  
 spec:  
 template:  
 spec:  
 restartPolicy: Never  
 serviceAccountName: spark-runner  
 containers:  
 - name: driver  
 image: ghcr.io/YOURORG/spark-app:{{GIT\_SHA}}  
 command: ["/opt/spark/bin/spark-submit"]  
 args: [  
 "--master", "k8s://https://kubernetes.default.svc",  
 "--deploy-mode", "cluster",  
 "--conf", "spark.kubernetes.container.image=ghcr.io/YOURORG/spark-app:{{GIT\_SHA}}",  
 "--conf", "spark.kubernetes.namespace=spark",  
 "--conf", "spark.executor.instances=2",  
 "local:///opt/app/app.py"  
 ]

### Makefile

REGISTRY ?= ghcr.io/YOURORG  
APP ?= spark-app  
TAG ?= $(shell git rev-parse --short HEAD)  
IMAGE := $(REGISTRY)/$(APP):$(TAG)  
  
kind-up:  
 kind create cluster --name spark --wait 60s || true  
 kubectl apply -f k8s/rbac.yaml  
  
kind-down:  
 kind delete cluster --name spark || true  
  
build:  
 docker build -t $(IMAGE) -f docker/Dockerfile .  
  
push:  
 docker push $(IMAGE)  
 sed -e 's/{{GIT\_SHA}}/$(TAG)/g' k8s/job-spark.yaml | kubectl apply -f -  
 sed -e 's/{{GIT\_SHA}}/$(TAG)/g' k8s/cronjob-spark.yaml | kubectl apply -f -  
  
test:  
 pytest -q  
  
sbom:  
 syft $(IMAGE) -o spdx-json > sbom-$(TAG).spdx.json || true  
  
scan:  
 trivy image --exit-code 1 --ignore-unfixed $(IMAGE) || true  
  
sign:  
 COSIGN\_EXPERIMENTAL=1 cosign sign --key cosign.key $(IMAGE) || true  
  
deploy: build push  
  
cron-deploy:  
 sed -e 's/{{GIT\_SHA}}/$(TAG)/g' k8s/cronjob-spark.yaml | kubectl apply -f -  
  
logs:  
 kubectl logs job/spark-pi -n spark -f || true

### GitHub Actions (ci/github-actions/ci.yaml)

name: ci  
on:  
 push:  
 branches: [ main ]  
 pull\_request:  
  
jobs:  
 build-test-publish:  
 runs-on: ubuntu-latest  
 permissions:  
 contents: read  
 packages: write  
 id-token: write  
 steps:  
 - uses: actions/checkout@v4  
 - uses: actions/setup-python@v5  
 with: { python-version: '3.11' }  
 - name: Install test deps  
 run: pip install -r spark-app/requirements.txt pytest  
 - name: Run tests  
 run: pytest -q  
 - name: Build image  
 run: |  
 echo "IMAGE=ghcr.io/${{ github.repository\_owner }}/spark-app:${{ github.sha }}" >> $GITHUB\_ENV  
 docker build -t $IMAGE -f docker/Dockerfile .  
 - name: Login GHCR  
 uses: docker/login-action@v3  
 with:  
 registry: ghcr.io  
 username: ${{ github.actor }}  
 password: ${{ secrets.GITHUB\_TOKEN }}  
 - name: Push image  
 run: docker push $IMAGE  
 - name: SBOM (Syft)  
 uses: anchore/syft-action@v0.17.0  
 with:  
 image: ${{ env.IMAGE }}  
 output: 'spdx-json=sbom.spdx.json'  
 - name: Upload SBOM artifact  
 uses: actions/upload-artifact@v4  
 with:  
 name: sbom  
 path: sbom.spdx.json  
 - name: Sign image (Cosign)  
 uses: sigstore/cosign-installer@v3  
 - name: Cosign sign  
 env:  
 COSIGN\_EXPERIMENTAL: '1'  
 run: cosign sign --yes $IMAGE

### Tekton (ci/tekton/pipeline.yaml — excerpt)

apiVersion: tekton.dev/v1beta1  
kind: Pipeline  
metadata:  
 name: spark-cd  
spec:  
 params:  
 - name: image  
 tasks:  
 - name: build  
 taskRef: { name: kaniko }  
 params:  
 - name: IMAGE  
 value: $(params.image)  
 - name: test  
 runAfter: [ build ]  
 taskRef: { name: pytest }  
 - name: sbom  
 runAfter: [ test ]  
 taskRef: { name: syft }  
 - name: sign  
 runAfter: [ sbom ]  
 taskRef: { name: cosign }  
 - name: deploy  
 runAfter: [ sign ]  
 taskRef: { name: kubectl-apply }  
 params:  
 - name: manifests  
 value: k8s/job-spark.yaml

### Production Checklist (copy into your wiki)

* **Multi‑tenancy:** Namespaces, quotas, admission; golden base images.
* **Secrets:** Vault/ESO; rotate; restrict mount paths; audit access.
* **Supply chain:** SBOM + sign + attest; block unsigned images.
* **Autoscaling:** Cluster autoscaler; VPA for drivers; TTL for finished jobs.
* **Observability:** Export Spark metrics; central log aggregation; SLOs & alerts.
* **Promotion:** dev→stage→prod via GitOps; change tickets emitted from CDEvents.

### FAQ

* **Why not YARN?** CD, portability, and unified platform with the rest of your apps; reduced spin‑up & better bin‑packing.
* **Do I need the Spark Operator?** Helpful for retries & CRDs at scale; the Job pattern here is simpler for a workshop.
* **How do I manage costs?** Requests/limits, quotas, spot pools, and per‑job cost tagging; alert on cost/run SLO.

**End of deck & README template.**

## Key Takeaways (slide)

* **CD blueprint for Spark:** Commit → build artifact (image/jar/wheel) → automated checks → promote to envs → safe rollouts (Job/CronJob/SparkApplication) with rollback.
* **Quality gates for code *and***\*\* data:\*\* Unit/integration tests, schema & freshness checks, dependency scanning, SBOM, signatures.
* **CDEvents everywhere:** Event-driven orchestration + notifications across CI, registry, and runtime for traceability.
* **Platform guardrails:** Namespaces & quotas, secrets management, cost controls, multi-tenancy patterns, batch SLOs.
* **From laptop to prod:** Reproducible local demo (kind/Minikube) → managed K8s (EKS/GKE/AKS) with Tekton/Jenkins/GitOps.

## Resource Sizing Cheat‑Sheet (slide)

| Tier | Driver (vCPU/RAM) | Executor (vCPU/RAM) | Count | Notes |
| --- | --- | --- | --- | --- |
| **Dev/Small (<100 GB)** | 1 / 2 GiB | 1 / 2 GiB | 2–3 | kind/minikube will need ≥3 vCPU, 4 GiB total; expect pod overhead. |
| **Medium (0.1–1 TB)** | 2–4 / 4–8 GiB | 3–4 / 8–16 GiB | 10–20 | Keep ≤5 cores/executor; memoryOverhead ≥ 10–25%. |
| **Large (1–10 TB)** | 4–8 / 8–16 GiB | 4–5 / 24–32 GiB | 50–100 | Use SSD local dirs; consider Cluster Autoscaler + Dynamic Allocation. |

**Rules of thumb:** executors ≤32 GiB; ≤5 cores/executor; raise spark.executor.memoryOverhead for Python/shuffle‑heavy jobs; account for node reserves + pod overhead on K8s.

## CDEvents Mapping (slide)

**Artifact published (CI → registry):**

{  
 "context": {"version": "0.4.0", "id": "<uuid>", "source": "ci://build/123"},  
 "subject": {"id": "spark-app:${GIT\_SHA}", "type": "artifact"},  
 "type": "dev.cdevents.artifact.published.v1",  
 "data": {"repository": "ghcr.io/yourorg/spark-app", "digest": "sha256:...", "sbom": "spdx-json"}  
}

**Pipeline run finished (CI):**

{"type":"dev.cdevents.pipelinerun.finished.v1","subject":{"id":"build-123"},"data":{"outcome":"success"}}

**Deployment started/finished (CD):**

{"type":"dev.cdevents.deployment.started.v1","subject":{"id":"job/spark-pi"}}

## Great Expectations / Pandera quickstart (README add‑on)

* **Option A (pandera):** Already shown in tests/test\_data\_contract.py — extend with column ranges, uniqueness, and freshness.
* **Option B (Great Expectations):**

pip install great\_expectations  
great\_expectations --v3-api init

Create a checkpoint that validates sample input before deploy; fail the pipeline if expectations fail.

## Spark → Prometheus (slide & README add‑on)

**metrics.properties** (bundle into image at /opt/spark/conf/):

jmxReporter.enabled=true  
metrics.namespace=spark

**Driver/Executor JMX exporter (agent) example:**

--conf spark.metrics.conf=/opt/spark/conf/metrics.properties \  
--conf spark.driver.extraJavaOptions=-javaagent:/opt/jmx/jmx\_prometheus\_javaagent.jar=9404:/opt/jmx/config.yaml \  
--conf spark.executor.extraJavaOptions=-javaagent:/opt/jmx/jmx\_prometheus\_javaagent.jar=9504:/opt/jmx/config.yaml

**K8s scrape annotations (if using Prometheus Operator):**

metadata:  
 annotations:  
 prometheus.io/scrape: "true"  
 prometheus.io/port: "9404"

## License & Contributing (README section)

* **License:** Apache‑2.0. Include LICENSE file; images and sample code under same license.
* **Contributing:** PRs welcome — add tests, keep Docker layers slim, and update SBOM/signing steps.
* **Security:** Report vulnerabilities privately; avoid committing secrets; prefer external secret stores.

# BARE\_BONES\_DEMO.md (hand-off for Windsurf)

**Goal:** Minimal, self-contained demo of **CD for a Spark job on containers** using a local Kubernetes (kind), a tiny Docker image, and a single Kubernetes **Job** manifest. Copy this section into a file named BARE\_BONES\_DEMO.md at the repo root and give it to Windsurf to scaffold the files.

## What you get

* One PySpark job (spark-app/app.py)
* Docker image (apache/spark-py base)
* Kubernetes namespace/RBAC + **Job** manifest
* Make targets to build → deploy → watch
* Optional GitHub Actions workflow to build & push the image

## Prereqs (local)

* Docker, kubectl, kind, make
* GitHub Container Registry (GHCR) or any OCI registry (set REGISTRY)

## Quickstart

make kind-up # Create kind cluster + namespace/RBAC  
make build # Build container image  
make deploy # Push image + apply K8s Job  
kubectl logs job/spark-pi -n spark -f # Tail job logs  
make kind-down # Delete kind cluster

**Note:** Set your registry once (e.g., export REGISTRY=ghcr.io/YOURUSER). If using GHCR, ensure you’re logged in: echo $GH\_TOKEN | docker login ghcr.io -u YOURUSER --password-stdin.

## Files to create

Create these files verbatim.

### 1) Makefile

REGISTRY ?= ghcr.io/YOURUSER  
APP ?= spark-app  
TAG ?= $(shell git rev-parse --short HEAD 2>/dev/null || echo dev)  
IMAGE := $(REGISTRY)/$(APP):$(TAG)  
  
.PHONY: kind-up kind-down build push deploy logs clean  
  
kind-up:  
 kind create cluster --name spark --wait 60s || true  
 kubectl apply -f k8s/namespace.yaml  
 kubectl apply -f k8s/rbac.yaml  
  
kind-down:  
 kind delete cluster --name spark || true  
  
build:  
 docker build -t $(IMAGE) -f docker/Dockerfile .  
  
push:  
 docker push $(IMAGE)  
  
# Replace {{IMAGE}} placeholder in manifests with the concrete image tag  
\_deploy\_apply:  
 sed 's#{{IMAGE}}#$(IMAGE)#g' k8s/job-spark.yaml | kubectl apply -f -  
  
deploy: build push \_deploy\_apply  
  
logs:  
 kubectl logs job/spark-pi -n spark -f || true  
  
clean:  
 kubectl delete job spark-pi -n spark --ignore-not-found

### 2) docker/Dockerfile

FROM apache/spark-py:v3.5.1  
WORKDIR /opt/app  
COPY spark-app/ /opt/app/  
# If you need extra Py deps, add requirements.txt and uncomment below  
# COPY spark-app/requirements.txt ./  
# RUN pip install --no-cache-dir -r requirements.txt

### 3) spark-app/app.py

from pyspark.sql import SparkSession  
import random  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 spark = SparkSession.builder.appName("spark-pi").getOrCreate()  
 sc = spark.sparkContext  
 n = 1\_000\_000 # keep small for fast demo  
 def inside(\_):  
 x, y = random.random(), random.random()  
 return 1 if x\*x + y\*y <= 1 else 0  
 count = sc.parallelize(range(n), numSlices=8).map(inside).reduce(lambda a,b: a+b)  
 pi = 4.0 \* count / n  
 print(f"Pi is roughly {pi}")  
 spark.stop()

### 4) k8s/namespace.yaml

apiVersion: v1  
kind: Namespace  
metadata:  
 name: spark

### 5) k8s/rbac.yaml

apiVersion: v1  
kind: ServiceAccount  
metadata:  
 name: spark-runner  
 namespace: spark  
---  
apiVersion: rbac.authorization.k8s.io/v1  
kind: Role  
metadata:  
 name: spark-edit  
 namespace: spark  
rules:  
- apiGroups: ["", "batch", "apps"]  
 resources: ["pods", "pods/log", "services", "configmaps", "secrets", "jobs"]  
 verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]  
---  
apiVersion: rbac.authorization.k8s.io/v1  
kind: RoleBinding  
metadata:  
 name: spark-edit  
 namespace: spark  
subjects:  
- kind: ServiceAccount  
 name: spark-runner  
 namespace: spark  
roleRef:  
 apiGroup: rbac.authorization.k8s.io  
 kind: Role  
 name: spark-edit

### 6) k8s/job-spark.yaml

apiVersion: batch/v1  
kind: Job  
metadata:  
 name: spark-pi  
 namespace: spark  
spec:  
 backoffLimit: 0  
 ttlSecondsAfterFinished: 600  
 template:  
 spec:  
 restartPolicy: Never  
 serviceAccountName: spark-runner  
 containers:  
 - name: driver  
 image: {{IMAGE}}  
 imagePullPolicy: IfNotPresent  
 command: ["/opt/spark/bin/spark-submit"]  
 args:  
 [  
 "--master", "k8s://https://kubernetes.default.svc",  
 "--deploy-mode", "cluster",  
 "--conf", "spark.kubernetes.container.image={{IMAGE}}",  
 "--conf", "spark.kubernetes.namespace=spark",  
 "--conf", "spark.executor.instances=2",  
 "--conf", "spark.executor.memory=1g",  
 "--conf", "spark.executor.cores=1",  
 "local:///opt/app/app.py"  
 ]  
 resources:  
 requests: { cpu: "500m", memory: "512Mi" }  
 limits: { cpu: "1", memory: "1Gi" }

### 7) *(Optional)* .github/workflows/ci.yml

name: ci  
on: { push: { branches: [ main ] }, pull\_request: {} }  
jobs:  
 build-push:  
 runs-on: ubuntu-latest  
 permissions: { contents: read, packages: write }  
 steps:  
 - uses: actions/checkout@v4  
 - name: Set IMAGE  
 run: echo "IMAGE=ghcr.io/${GITHUB\_REPOSITORY\_OWNER}/spark-app:${GITHUB\_SHA}" >> $GITHUB\_ENV  
 - uses: docker/login-action@v3  
 with: { registry: ghcr.io, username: ${{ github.actor }}, password: ${{ secrets.GITHUB\_TOKEN }} }  
 - name: Build  
 run: docker build -t $IMAGE -f docker/Dockerfile .  
 - name: Push  
 run: docker push $IMAGE

## Notes for production mapping

* Swap kind with managed K8s (EKS/GKE/AKS).
* Replace Job with **SparkApplication** (Spark Operator) or schedule via **CronJob**.
* Add tests, SBOM/signing steps, and policy gates in CI before push.
* Add Prometheus scrape annotations to capture Spark metrics.

This is intentionally minimal—no tests, no SBOM, no signing—so Windsurf can scaffold quickly. Use the fuller README in the deck for production-ready extras.

# LOCAL-ONLY VARIANT (no registry, 100% on your laptop)

Use this if you **cannot** push to any registry. It preloads the Docker image into the kind/minikube node image store so Kubernetes finds it locally.

## Local-only Quickstart

make kind-up  
make build  
make load # kind: preload image into cluster nodes  
make deploy-local # apply manifests with local image reference  
kubectl logs job/spark-pi -n spark -f  
make kind-down

**Minikube alternative:**

minikube start --cpus=3 --memory=4096  
make build  
make load-minikube  
make deploy-local  
kubectl logs job/spark-pi -n spark -f

## Local-only Makefile (drop-in replacement)

# Local-only Makefile: no registry, no pushes required  
APP ?= spark-app  
TAG ?= $(shell git rev-parse --short HEAD 2>/dev/null || echo dev)  
IMAGE := $(APP):$(TAG)  
  
.PHONY: kind-up kind-down build load load-minikube deploy-local logs clean  
  
kind-up:  
 kind create cluster --name spark --wait 60s || true  
 kubectl apply -f k8s/namespace.yaml  
 kubectl apply -f k8s/rbac.yaml  
  
kind-down:  
 kind delete cluster --name spark || true  
  
build:  
 docker build -t $(IMAGE) -f docker/Dockerfile .  
  
# Local-only: preload image into kind nodes (so K8s finds it without pulling)  
load:  
 kind load docker-image $(IMAGE) --name spark  
  
# Minikube alternative  
load-minikube:  
 minikube image load $(IMAGE)  
  
# Replace {{IMAGE}} placeholder and apply manifests  
\_deploy\_apply:  
 sed 's#{{IMAGE}}#$(IMAGE)#g' k8s/job-spark.yaml | kubectl apply -f -  
  
# Deploy without any registry/push  
deploy-local: build load \_deploy\_apply  
  
logs:  
 kubectl logs job/spark-pi -n spark -f || true  
  
clean:  
 kubectl delete job spark-pi -n spark --ignore-not-found

**That’s it.** Completely local, free, and open-source: Docker + kind/minikube + kubectl. Pull the apache/spark-py base image once, then everything runs offline.