

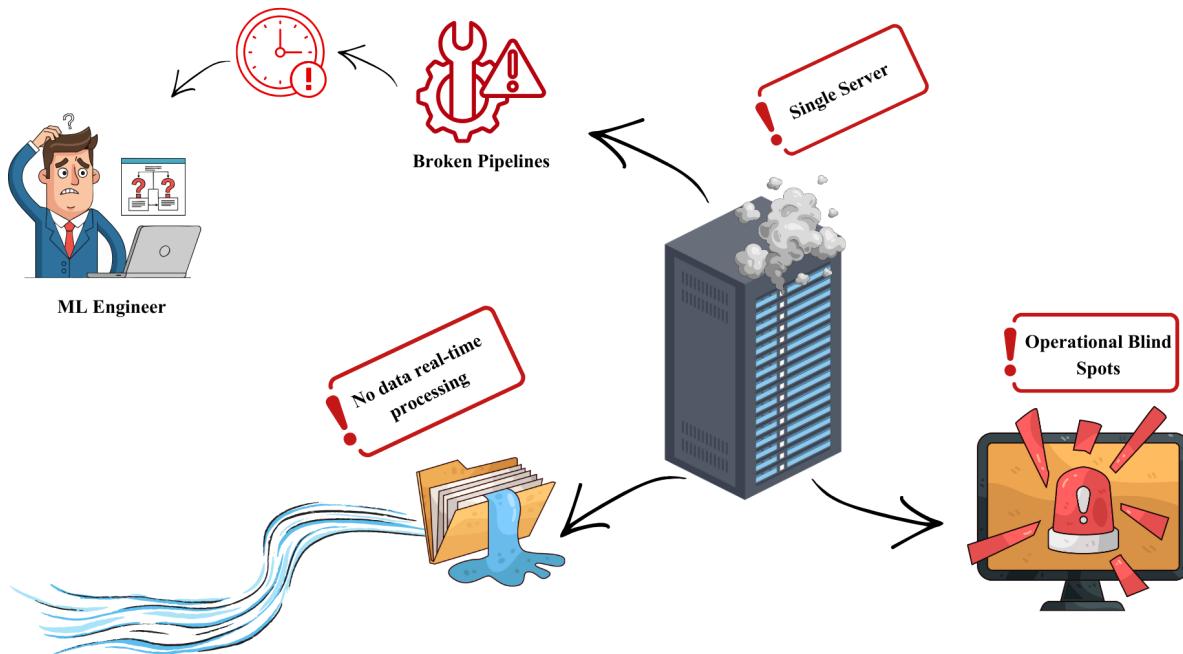
AI VIET NAM – AI COURSE 2025

# Tutorial: Kubernetes, Kubeflow, Spark Streaming và Kafka

Huu-Anh-Duong Ta, Duong-Nguyen-Thuan, Quang-Vinh Dinh

## I. Bối cảnh

Trong thời đại dữ liệu và AI phát triển nhanh, một mô hình tốt là chưa đủ. Các tổ chức cần một hạ tầng tự động, linh hoạt và dễ mở rộng, nơi việc huấn luyện, triển khai, cập nhật mô hình và xử lý dữ liệu real-time diễn ra liên tục mà không gián đoạn dịch vụ.



Hình 1: Kiến trúc Triển khai truyền thống và triển khai qua container

Hệ thống truyền thống (chạy trên máy chủ cố định, thao tác thủ công) nhanh chóng bộc lộ các hạn chế: khó mở rộng khi lưu lượng tăng, thiếu pipeline chuẩn cho huấn luyện – kiểm thử – triển khai, không đáp ứng được dữ liệu thời gian thực và thiếu cơ chế quan sát (observability), autoscaling hay rollback an toàn.

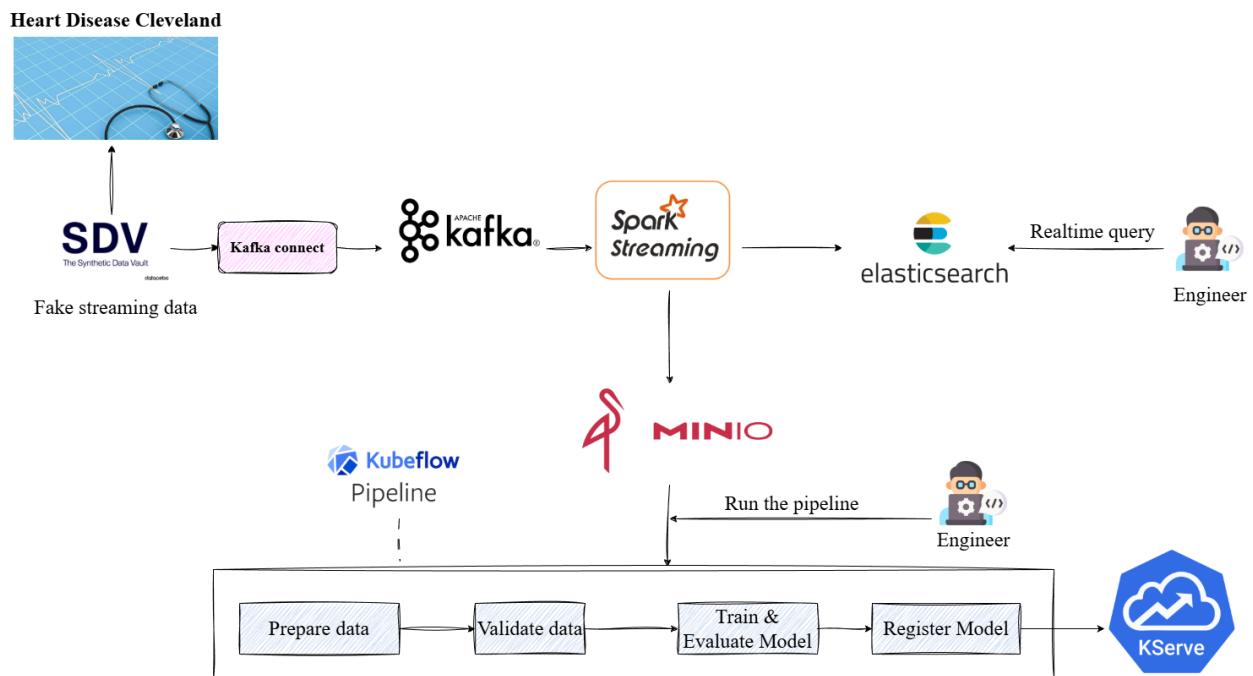
Từ đó, bài toán trọng tâm không chỉ là tạo mô hình chính xác, mà là đảm bảo mô hình có thể vận hành bền vững và cập nhật theo dữ liệu thực tế mà vẫn giữ hệ thống ổn định:

*Làm sao để mô hình được huấn luyện – triển khai – cập nhật liên tục, trong khi toàn bộ hệ thống vẫn ổn định, minh bạch và kiểm soát được rủi ro?*

Giải pháp hiện đại hướng đến MLOps Pipeline, nơi vòng đời mô hình được tự động hóa trên kiến trúc linh hoạt:

- **Kubernetes:** điều phối container, mở rộng theo tải và tránh phụ thuộc hạ tầng.
- **Kubeflow:** pipeline hoá training, evaluation và CI/CD mô hình.
- **Kafka:** trục truyền dữ liệu streaming giữa các dịch vụ.
- **Spark Streaming:** xử lý và tạo luồng dữ liệu real-time cho training hoặc inference.

Khi kết hợp các thành phần này, chúng ta có một kiến trúc hợp nhất, giúp mô hình không dừng lại ở mức phòng thí nghiệm, mà trở thành một phần của hệ thống vận hành:



Hình 2: Luồng xử lý dữ liệu và mô hình trong hệ thống MLOps Streaming hiện đại

# Mục lục

I.	Bối cảnh . . . . .	1
II.	Nền tảng lý thuyết . . . . .	5
II.1.	Kubernetes - k8s . . . . .	5
II.1.1.	Hành trình triển khai ứng dụng . . . . .	5
II.1.2.	Thành phần kiến trúc Kubernetes . . . . .	6
II.1.3.	Các bản phân phối của Kubernetes . . . . .	9
II.1.4.	Hướng dẫn cài đặt Kubernetes . . . . .	9
II.2.	Kubeflow . . . . .	10
II.2.1.	Tổng quan kiến trúc Kubeflow . . . . .	10
II.2.2.	Hướng dẫn cài đặt Kubeflow . . . . .	12
II.2.3.	Kubeflow Notebooks . . . . .	13
II.2.4.	Kubeflow Pipeline . . . . .	15
II.2.5.	Kubeflow Trainers . . . . .	17
II.2.6.	Kubeflow Katib . . . . .	21
II.2.7.	Kubeflow KServe . . . . .	25
II.2.8.	Kubeflow Spark Operator . . . . .	27
II.2.9.	Kubeflow Model Registry . . . . .	28
II.3.	Apache Kafka . . . . .	29
II.4.	Spark Streaming . . . . .	31
II.4.1.	Input DStreams và Receivers . . . . .	32
II.4.2.	Transformations on DStreams . . . . .	33
II.4.3.	Output Operations on DStreams . . . . .	33
III.	Thực hành: . . . . .	34
III.1.	Giới thiệu bài thực hành . . . . .	34
III.2.	Các bước thực hiện . . . . .	35
III.2.1.	Cấu hình Docker Compose, Elasticsearch, Minio . . . . .	35
III.2.2.	Thiết lập luồng stream và xử lí data . . . . .	41
III.2.3.	Cấu hình Kubeflow pipeline cho quá trình huấn luyện và triển khai mô hình với KServe . . . . .	56
IV.	Tài liệu tham khảo . . . . .	71
V.	Source code . . . . .	71

### Bảng Giải Thích Thuật Ngữ

Thuật Ngữ	Giải thích
<b>PoC (Proof of Concept)</b>	Là phiên bản thử nghiệm nhỏ để kiểm tra xem giải pháp hoặc công nghệ có khả thi hay không trước khi đầu tư xây dựng hệ thống hoàn chỉnh.
<b>Staging environment</b>	Là môi trường trung gian giữa dev và production, mô phỏng hầu hết cấu hình hệ thống thật để kiểm thử cuối cùng trước khi triển khai chính thức.
<b>Rollback</b>	Là quá trình khôi phục hệ thống về phiên bản ổn định trước đó khi bản triển khai mới gây lỗi.
<b>Container orchestration engine</b>	Là hệ thống tự động quản lý việc triển khai, mở rộng, kết nối mạng và phục hồi container trên nhiều máy, ví dụ như Kubernetes hay Docker Swarm.
<b>Autoscaling</b>	Cơ chế tự động tăng/giảm số lượng container hoặc tài nguyên hệ thống dựa trên tải thực tế (CPU, RAM, request).
<b>Observability</b>	Là khả năng theo dõi hoạt động hệ thống thông qua log, metric, tracing để hiểu trạng thái nội bộ và phát hiện lỗi.
<b>Model serving</b>	Là quá trình đưa mô hình machine learning vào môi trường production để nhận request (API) và trả về kết quả dự đoán theo thời gian thực (real-time) hoặc theo lô (batch).
<b>Cloud provider</b>	Là đơn vị cung cấp hạ tầng như máy chủ, lưu trữ, mạng và dịch vụ quản lý (ví dụ AWS, GCP, Azure) để triển khai Kubernetes, Kubeflow hoặc hệ thống xử lý dữ liệu như Spark.
<b>Load Balancer</b>	Là thành phần phân phối request từ người dùng đến nhiều pod hoặc service khác nhau, giúp hệ thống hoạt động ổn định, tránh quá tải và tăng khả năng mở rộng.
<b>Kubeflow Manifests</b>	Là tập hợp các file YAML mô tả tài nguyên Kubernetes (Pod, Service, CRD, ...) dùng để cài đặt và cấu hình Kubeflow trên cluster.
<b>Scale-to-Zero</b>	Tính năng quan trọng của Serverless. Nếu không có ai gọi API của model, hệ thống sẽ tắt hẳn Pod để không tốn tiền/tài nguyên. Khi có request mới, nó mới tự bật lại.
<b>Canary Rollouts</b>	Kỹ thuật deploy phiên bản mới cho một nhóm nhỏ người dùng (ví dụ 10% traffic) để test lỗi trước khi bung ra cho 100%.

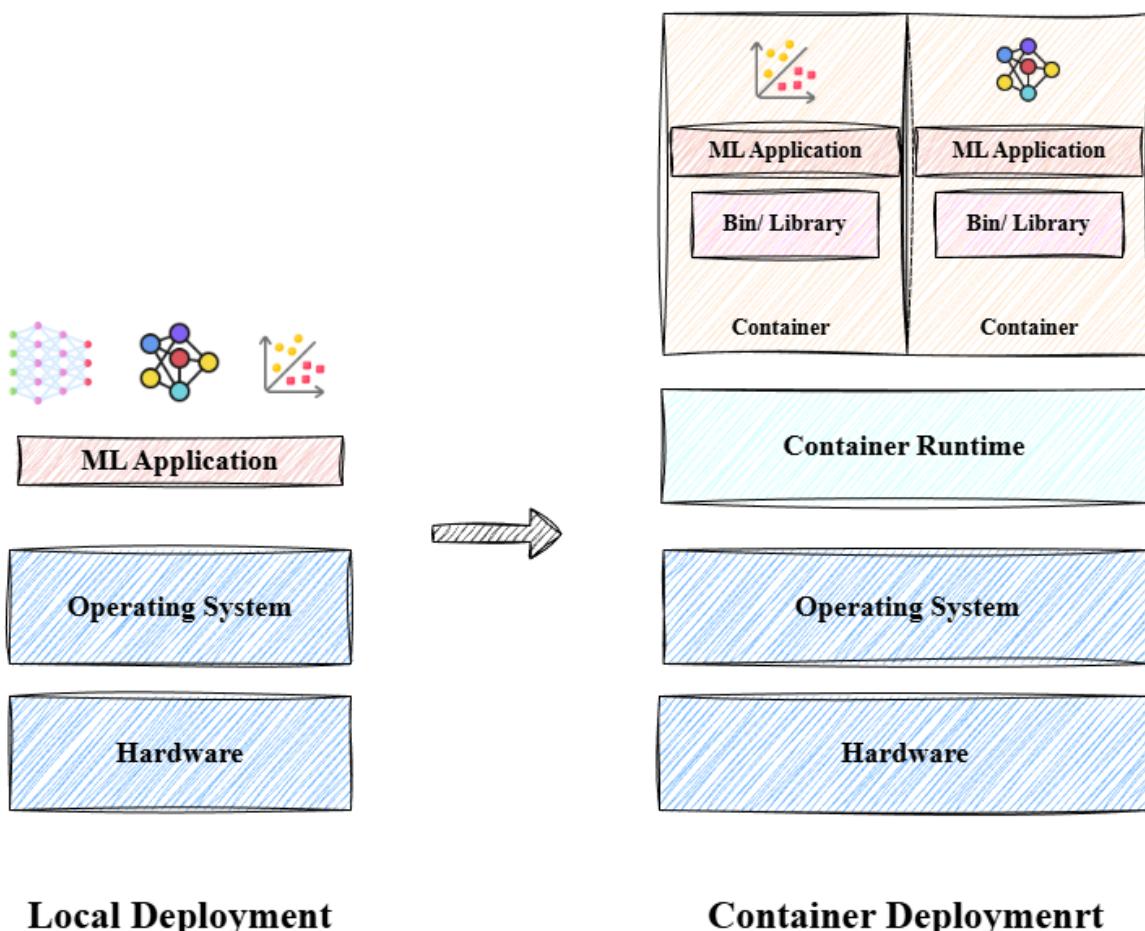
## II. Nền tảng lý thuyết

### II.1. Kubernetes - k8s

Kubernetes là một nền tảng mã nguồn mở, di động (portable), và có thể mở rộng (scalable) để quản lý các ứng dụng dạng container. Nền tảng này hỗ trợ tự động hóa việc triển khai, mở rộng, giám sát và điều phối tài nguyên cho ứng dụng. Kubernetes hiện nay được xem là tiêu chuẩn công nghiệp trong việc vận hành hệ thống phân tán và các workflow MLOps.

#### II.1.1. Hành trình triển khai ứng dụng

Để hiểu vì sao Kubernetes trở thành cốt lõi trong hạ tầng AI/ML hiện đại, ta cần nhìn lại tiến trình phát triển phương thức triển khai ứng dụng:



Hình 3: Kiến trúc Triển khai Truyền thống và Triển khai qua Container

**Traditional Deployment** là giai đoạn sơ khai khi ứng dụng được chạy trực tiếp trên máy tính cá nhân hoặc máy chủ vật lý. Ở môi trường này, ứng dụng ML, các thư viện (TensorFlow, PyTorch, CUDA) và hệ điều hành gắn chặt vào nhau, dẫn đến vấn đề nghiêm trọng về **Dependency Hell** - tức chỉ một thay đổi nhỏ ở thư viện cũng có thể khiến toàn bộ hệ thống lỗi. Bên cạnh đó, việc

nhiều ứng dụng chia sẻ cùng một máy chủ dễ gây tranh chấp tài nguyên (CPU/GPU/RAM), dẫn tới mất ổn định hoặc treo hệ thống. Việc tách mỗi mô hình sang một máy riêng giúp giảm xung đột nhưng lại gây **lãng phí tài nguyên và chi phí duy trì cao**.

**Container Deployment** xuất hiện như một bước tiến chiến lược. Ứng dụng được đóng gói toàn bộ môi trường chạy (runtime, package, dependency) trong các container độc lập. Điều này đảm bảo:

- Tính nhất quán môi trường giữa Development → Staging → Production
- Triển khai linh hoạt trên nhiều hệ điều hành và hạ tầng khác nhau
- Cố lập tài nguyên tốt hơn, tránh xung đột giữa các mô hình

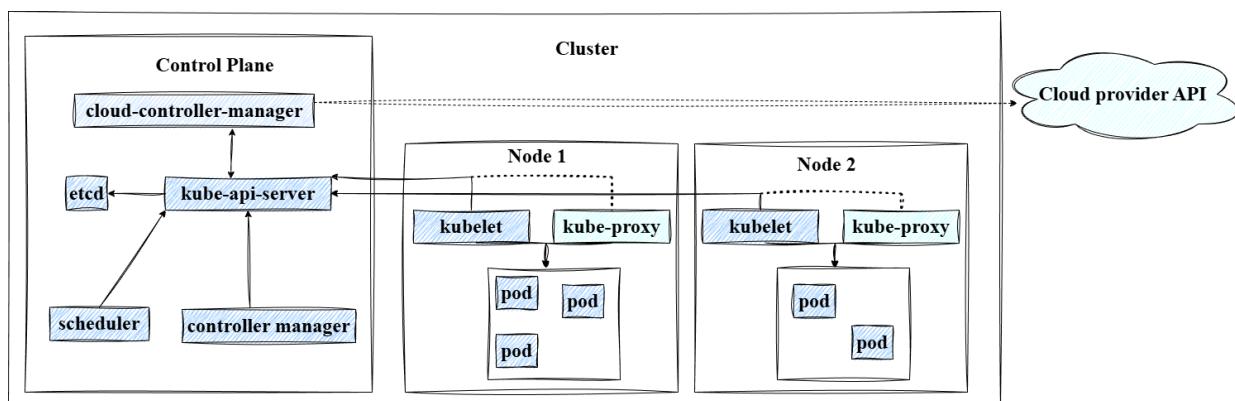
Tuy nhiên, khi hệ thống mở rộng lên hàng trăm/thập chí hàng nghìn container — việc quản lý thủ công trở nên quá tải và không còn khả thi.

Đây chính là lúc **Kubernetes đóng vai trò “nhạc trưởng”**. Kubernetes tự động hoá việc điều phối container, phân phối workload, cân bằng tải, mở rộng/thu hẹp tài nguyên theo nhu cầu và đảm bảo tính sẵn sàng (high availability). Trong bối cảnh MLOps, Kubernetes không chỉ chạy model serving mà còn hỗ trợ toàn bộ vòng đời ML như:

- Training pipeline (Kubeflow, Spark on Kubernetes)
- Model serving (Seldon, KFServing)
- Autoscaling theo tải dự đoán (HPA, KEDA)
- Tích hợp giám sát - logging - tracing

Nhờ đó, Kubernetes trở thành nền tảng xương sống cho các hệ thống AI/ML hiện đại.

### II.1.2. Thành phần kiến trúc Kubernetes

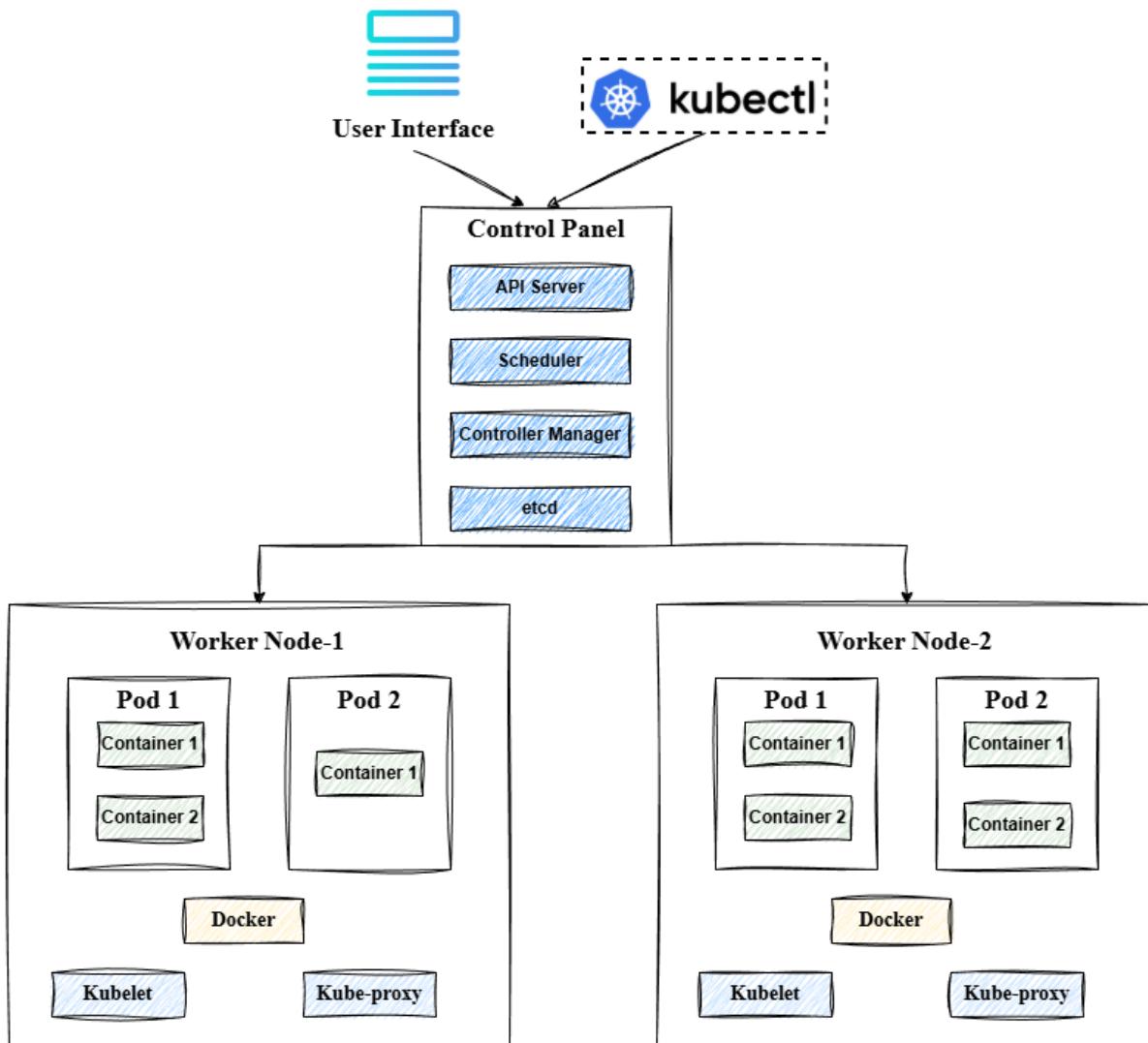


Hình 4: Kubernetes cluster components

Về mặt kiến trúc, Kubernetes được cấu thành từ các thành phần chính sau đây:

- **Cluster:** Là tập hợp toàn bộ tài nguyên máy tính, bao gồm Control Plane và Worker Node, tạo thành một môi trường hoàn chỉnh để vận hành ứng dụng.
- **Control Plane:** Đóng vai trò là hệ thống điều khiển trung tâm, có nhiệm vụ lập lịch trình, theo dõi trạng thái Cluster và đưa ra mọi quyết định điều phối hệ thống.
- **Node:** Là máy vật lý hoặc máy ảo trực tiếp thực thi các tác vụ tính toán và lưu trữ các ứng dụng.
- **Pod:** Là đơn vị nhỏ nhất trong Kubernetes, đóng vai trò như các đơn vị chứa ứng dụng hoặc sản phẩm thực tế của người dùng.
- **Kubelet:** Là một tác nhân (agent) chạy trên mỗi Node, chịu trách nhiệm nhận chỉ thị từ Control Plane để đảm bảo các tiến trình trên Node đó luôn vận hành ổn định.

Dưới đây, chúng ta sẽ đi sâu vào các thành phần kỹ thuật chi tiết của hệ thống.



Hình 5: Details Kubernetes cluster components

Đầu tiên là Control Plane đóng vai trò là cơ quan ra quyết định. Trong môi trường thực tế, nó thường được triển khai trên nhiều máy chủ để đảm bảo Tính sẵn sàng cao (High Availability); nếu một máy gặp sự cố, hệ thống vẫn vận hành bình thường. Trong Control Panel bao gồm:

- **API Server (kube-apiserver):** Đây là Front Desk của cả hệ thống - thành phần duy nhất giao tiếp trực tiếp với người dùng và các thành phần bên ngoài. Mọi lệnh bạn thực hiện (như qua kubectl) đều đi qua API Server để được xác thực và cập nhật trạng thái cụm.
- **Scheduler (kube-scheduler):** Nó theo dõi các Pod (đơn vị chứa container) mới chưa được phân chia, kiểm tra tài nguyên của từng Node để tìm ra vị trí lý tưởng nhất cho Pod đó dựa trên các tiêu chí về dung lượng và hiệu năng.
- **Controller Manager (kube-controller-manager):** Thành phần này chạy một vòng lặp liên tục để so sánh trạng thái thực tế của cụm với trạng thái mong muốn. Nếu một Node gặp sự cố, nó sẽ ngay lập tức kích hoạt việc tạo lại các Pod ở nơi khác để bù đắp.
- **Etcd:** Đây là kho lưu trữ dữ liệu dạng key-value, nơi ghi lại mọi thông tin cấu hình, bảo mật và trạng thái của toàn bộ cụm Kubernetes.
- **Cloud Controller Manager:** Thành phần này kết nối nội bộ (internal cluster) của người dùng với nhà cung cấp dịch vụ đám mây (cloud provider) bên ngoài (như AWS, Azure hoặc Google Cloud). Nó xử lý các tác vụ như tạo Cân bằng tải (Load Balancer) trên đám mây hoặc quản lý khối lượng lưu trữ do nhà cung cấp dịch vụ đám mây.

Thứ hai là Worker Node đóng vai trò là các máy chủ vật lý hoặc máy ảo trực tiếp thực hiện các tác vụ nặng. Một cụm thường có nhiều Node để xử lý lưu lượng truy cập lớn. Trong một Node Worker sẽ bao gồm:

- **Kubelet:** Đây là một agent nhỏ chạy trên từng Node, nhận chỉ thị từ Control Plane (ví dụ khởi chạy Pod) và báo cáo định kỳ về tình trạng sức khỏe của Node đó.
- **Kube-Proxy:** Thành phần xử lý các quy tắc mạng. Nó đảm bảo rằng lưu lượng truy cập có thể chảy chính xác giữa các Pod và Services khác nhau, duy trì các quy tắc mạng trên máy chủ.
- **Container Runtime:** Đây là phần mềm chịu trách nhiệm chạy các container.
- **Pod:** Như đã đề cập ở trên Pod là đơn vị tính toán nhỏ nhất có thể triển khai. Một Pod đại diện cho một tiến trình đang chạy trong Cluster.

Ngoài ra, Kubernetes cung cấp một số tài nguyên built-in workload cần chú ý như sau:

- **Deployment:** Một Deployment quản lý một tập hợp các Pods. nó đảm bảo một số lượng "replicas" (bản sao) nhất định của ứng dụng luôn chạy tại mọi thời điểm. Thành phần này cũng xử lý Rolling Updates, cho phép bạn cập nhật phần mềm mà không gây gián đoạn dịch vụ (downtime).
- **Service:** Một Service nhóm một tập hợp các Pods lại với nhau và cung cấp cho chúng một địa chỉ IP ổn định để chúng có thể liên lạc với nhau.

- **The Gateway API:** Trong khi Ingress là cách cũ để quản lý lưu lượng truy cập bên ngoài, Gateway API là tiêu chuẩn hiện đại. Nó linh hoạt hơn để quản lý định tuyến lưu lượng (traffic routing), phân tách rõ ràng vai trò của nhà cung cấp hạ tầng (người quản lý load balancer) và nhà phát triển (người định nghĩa các tuyến đường/routes).

### II.1.3. Các bản phân phối của Kubernetes

Kubernetes có nhiều cách triển khai (distribution) tùy theo nhu cầu sử dụng: học tập, phát triển, thử nghiệm hệ thống CI/CD, hoặc vận hành môi trường sản xuất. Dưới đây là các lựa chọn phổ biến kèm mục tiêu sử dụng:

- **textbf{Kubernetes} trên Docker Desktop hay OrbStack:** Phù hợp cho người mới bắt đầu hoặc nhà phát triển muốn chạy nhanh trên máy cá nhân. Docker Desktop cung cấp k8s tích hợp sẵn, cấu hình đơn giản, thích hợp để thử nghiệm ứng dụng container.
- **kind (Kubernetes IN Docker):** kind tạo cụm Kubernetes chạy bên trong Docker containers. Đây là lựa chọn lý tưởng cho local dev, CI/CD pipelines, luyện tập Helm, Kustomize hoặc thử nghiệm deployment.
- **Minikube:** Một bản phân phối độc lập, chạy trên nhiều hypervisor (Docker, VirtualBox, Hyper-V). Ổn định và phổ biến trong đào tạo, phù hợp cho các workshop và lab thực hành.
- **MicroK8s (Canonical – Ubuntu):** Bản phân phối nhẹ, tối ưu cho edge computing, AI/ML lab và cụm nội bộ. Cấu hình đơn giản, có thể mở rộng nhiều node. Dùng tốt cho môi trường dev, staging hoặc PoC.
- **k3s:** bản Kubernetes nhẹ, tối ưu cho môi trường tài nguyên hạn chế như VPS, edge computing hoặc IoT.
- **kubeadm:** công cụ chính thống để triển khai Kubernetes chuẩn production trên on-premise hoặc cloud riêng.
- **Managed Kubernetes (Dùng trên Cloud):** Triển khai trực tiếp trên dịch vụ được quản lý bởi nhà cung cấp cloud như GKE (Google Cloud), EKS (AWS) hay AKS (Microsoft Azure). Dùng trong production, có SLA, autoscaling, tích hợp monitoring, networking và bảo mật.

Việc lựa chọn bản phân phối phù hợp phụ thuộc vào mục tiêu sử dụng, tài nguyên hệ thống và mức độ kiểm soát mong muốn.

### II.1.4. Hướng dẫn cài đặt Kubernetes

**kubectl** là công cụ dòng lệnh chính để tương tác và quản lý cụm Kubernetes. Bạn cần cài đặt **kubectl** trước khi truy cập hoặc điều khiển cluster Kubernetes. Chi tiết các bạn xem tại đường dẫn sau: <https://kubernetes.io/vi/docs/tasks/tools/install-kubectl>

Tiếp theo, chúng ta sẽ cài đặt Kubernetes với bản phân phối **kind** (Kubernetes IN Docker) trên Windows, Cách cài đặt gồm các bước:

1. Cài đặt Docker Desktop: <https://docs.docker.com/desktop/setup/install/windows-install>

2. Cài đặt kind để tạo cụm Kubernetes chạy trên máy tính cá nhân:

### setup\_kind

```

1 New-Item -Path "C:\k8s-tools" -ItemType Directory -Force
2 cd C:\k8s-tools
3 curl.exe -Lo kind.exe https://kind.sigs.k8s.io/dl/v0.30.0/kind-
    windows-amd64
4 [Environment]::SetEnvironmentVariable("Path", $env:Path + ";C:\k8s-
    tools", "User")
5 # Tắt cửa sổ PowerShell hiện tại và mở cửa sổ mới và chạy:
6 kind --version
7

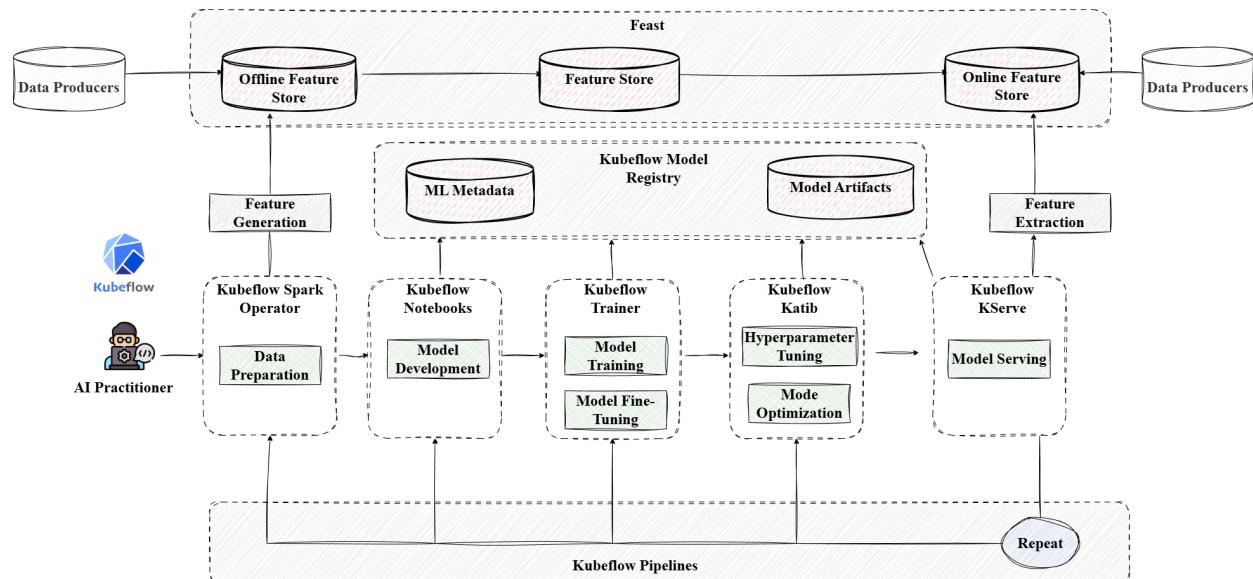
```

Đối với các hệ điều hành khác như MacOS, Ubuntu,... các bạn xem chi tiết tại đường dẫn sau: <https://kind.sigs.k8s.io/docs/user/quick-start/#installation>

## II.2. Kubeflow

Kubeflow là một bộ công cụ học máy dành cho Kubernetes, với mục tiêu cung cấp các manifests đơn giản giúp thiết lập một AI stack dễ sử dụng trên bất kỳ cụm Kubernetes nào, đi kèm với khả năng tự động cấu hình (self-configuration) dựa theo môi trường đích.

### II.2.1. Tổng quan kiến trúc Kubeflow



Hình 6: Kubeflow Lifecycle

Hệ sinh thái Kubeflow điều phối toàn bộ các giai đoạn từ xử lý dữ liệu đến triển khai mô hình thông qua các thành phần chuyên biệt như sau:

- **Data Preparation:** Tiếp nhận dữ liệu thô, thực hiện Feature Engineering để trích xuất các đặc trưng ML. Thông thường, bước này được liên kết với các công cụ xử lý dữ liệu như Spark, Dask, Flink hoặc Ray.
- **Model Development:** Huấn luyện hoặc tinh chỉnh các mô hình của mình trên môi trường điện toán quy mô lớn. Thông thường để tối ưu việc sử dụng chương trình đào tạo phân tán nếu GPU đơn lẻ không thể xử lý kích thước mô hình của bạn. Kết quả của việc huấn luyện mô hình là tạo phẩm mô hình đã huấn luyện mà bạn có thể lưu trữ trong Model Registry.
- **Model Optimization:** Tối ưu hóa siêu tham số mô hình và tối ưu hóa mô hình bằng các thuật toán AutoML khác nhau. Trong quá trình tối ưu hóa mô hình, người dùng có thể lưu trữ ML metadata trong Model Registry.
- **Model Serving:** Cung cấp mô hình dưới dạng dịch vụ để thực hiện Online Inference hoặc Batch Inference. Trong quá trình này, người dùng có thể sử dụng online feature store để trích xuất các features. Ngoài ra, người dùng sẽ theo dõi hiệu suất của mô hình và đưa kết quả vào các bước trước đó cho vòng đời AI.

Sau khi đã hiểu bốn giai đoạn chính trên, chúng ta sẽ tìm hiểu các thành phần của Kubeflow phù hợp như thế nào với từng giai đoạn:

- **Kubeflow Spark Operator:** Được sử dụng cho giai đoạn chuẩn bị dữ liệu (Data Preparation) và kỹ thuật đặc trưng (Feature Engineering).
- **Kubeflow Notebooks:** Môi trường tương tác dùng để phát triển mô hình và thử nghiệm các luồng công việc AI (AI workflows) cho khoa học dữ liệu.
- **Kubeflow Trainer:** Phục vụ cho việc huấn luyện phân tán quy mô lớn hoặc tinh chỉnh các mô hình ngôn ngữ lớn.
- **Kubeflow Katib:** Dùng để tối ưu hóa mô hình và tinh chỉnh siêu tham số (Hyperparameter Tuning) thông qua nhiều thuật toán AutoML khác nhau.
- **Kubeflow Model Registry:** Được sử dụng để lưu trữ ML Metadata, các thành phẩm mô hình (Model Artifacts) và chuẩn bị mô hình cho việc triển khai thực tế (production serving).
- **KServe:** Hỗ trợ thực hiện suy luận trực tuyến (Online Inference) và suy luận theo lô (Batch Inference) trong giai đoạn Model Serving.
- **Feast:** Đóng vai trò là một kho lưu trữ đặc trưng (Feature Store) để quản lý các đặc trưng ngoại tuyến và trực tuyến.
- **Kubeflow Pipelines:** Được sử dụng để xây dựng, triển khai và quản lý từng bước trong toàn bộ vòng đời AI (AI Lifecycle).

## II.2.2. Hướng dẫn cài đặt Kubeflow

Việc cài đặt Kubeflow khá linh hoạt, mọi người có thể cài đặt từng Kubeflow projects theo thông tin bảng phía dưới:

Kubeflow Projects		
Kubeflow Project	Source Code	Github link
<b>Kubeflow KServe</b>	kserve/kserve	<a href="https://github.com/kserve/kserve">https://github.com/kserve/kserve</a>
<b>Kubeflow Katib</b>	kubeflow/katib	<a href="https://github.com/kubeflow/katib">https://github.com/kubeflow/katib</a>
<b>Kubeflow Model Registry</b>	kubeflow/model-registry	<a href="https://github.com/kubeflow/model-registry">https://github.com/kubeflow/model-registry</a>
<b>Kubeflow Notebooks</b>	kubeflow/notebooks	<a href="https://github.com/kubeflow/notebooks">https://github.com/kubeflow/notebooks</a>
<b>Kubeflow Pipelines</b>	kubeflow/pipelines	<a href="https://github.com/kubeflow/pipelines">https://github.com/kubeflow/pipelines</a>
<b>Kubeflow Spark Operator</b>	kubeflow/spark-operator	<a href="https://github.com/kubeflow/spark-operator">https://github.com/kubeflow/spark-operator</a>
<b>Kubeflow Trainer</b>	kubeflow/trainer	<a href="https://github.com/kubeflow/trainer">https://github.com/kubeflow/trainer</a>
<b>Central Dashboard</b>	kubeflow/dashboard	<a href="https://github.com/kubeflow/dashboard">https://github.com/kubeflow/dashboard</a>

Ngoài ra mọi người có thể cài đặt tất cả các projects (dành cho Windows) với cách làm như sau:

1. Tạo cluster và clone repository:

```
setup_kubeflow

1 kind create cluster --name kubeflow-cluster --image kindest/node:v1
2 git clone https://github.com/kubeflow/manifests.git
3 cd manifests
```

2. Chạy script sau để cài đặt Kubeflow:

```
setup_kubeflow

1 while ($true) {
2     Write-Host "Setting Kubeflow (Server-Side Mode)... " -
3     ForegroundColor Cyan
4
5     kubectl apply -k example --server-side --force-conflicts
```

```

6      if ($LASTEXITCODE -eq 0) {
7          Write-Host "Setup successfully!" -ForegroundColor Green
8          break
9      }
10
11     Write-Host " Waiting other source ..." -ForegroundColor Yellow
12     Start-Sleep -Seconds 15
13 }
14

```

3. Tiến hành port-forward để truy cập Kubeflow Dashboard:

### setup\_kubeflow

```

1 kubectl port-forward svc/istio-ingressgateway -n istio-system 8080:80
2

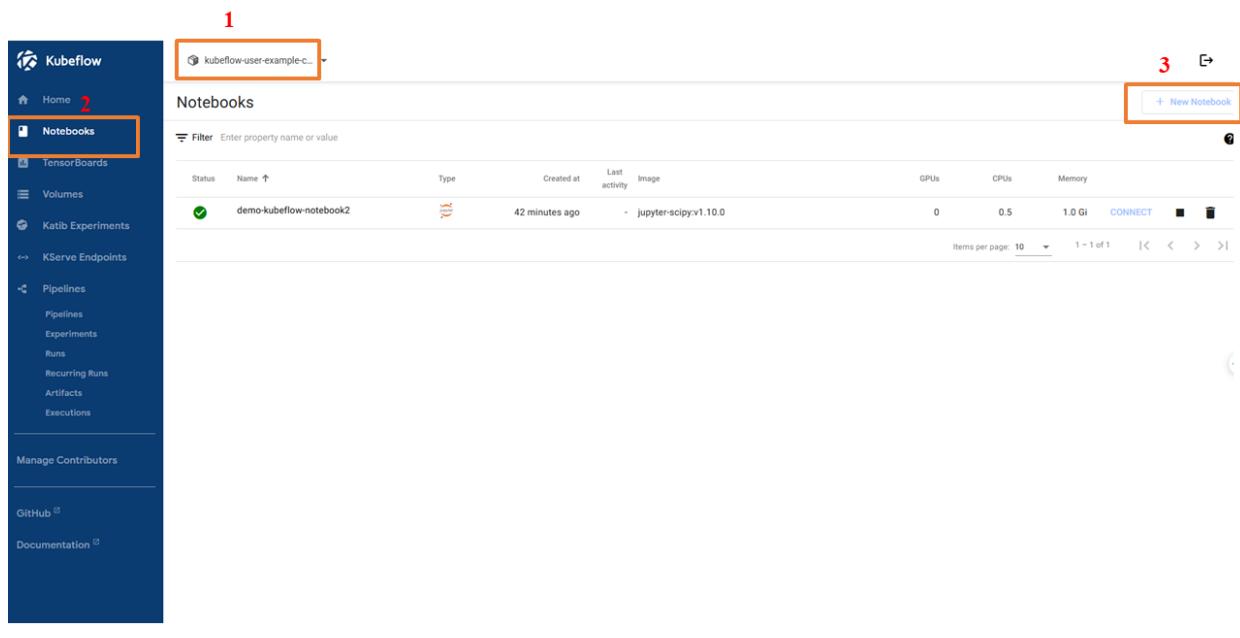
```

#### II.2.3. Kubeflow Notebooks

Kubeflow Notebooks cung cấp một cách để chạy môi trường phát triển dựa trên web bên trong cụm Kubernetes của bạn bằng cách chạy chúng bên trong các Pod. Một số điểm nổi bật của Kubeflow Notebooks gồm:

- Hỗ trợ sẵn cho các môi trường phát triển bao gồm JupyterLab, RStudio và Visual Studio Code.
- Người dùng có thể trực tiếp khởi tạo các container chứa môi trường lập trình (notebook containers) ngay trên hệ thống cụm thay vì triển khai cục bộ trên các máy trạm cá nhân.
- Cơ chế phân quyền được quản trị thông qua hệ thống RBAC (Role-Based Access Control) của Kubeflow, giúp tối ưu hóa và đơn giản hóa việc chia sẻ tài nguyên notebook trong phạm vi tổ chức.
- Quản trị viên có thể phân phối các pre-configured images, trong đó các thư viện và gói phần mềm cần thiết đã được cài đặt sẵn.

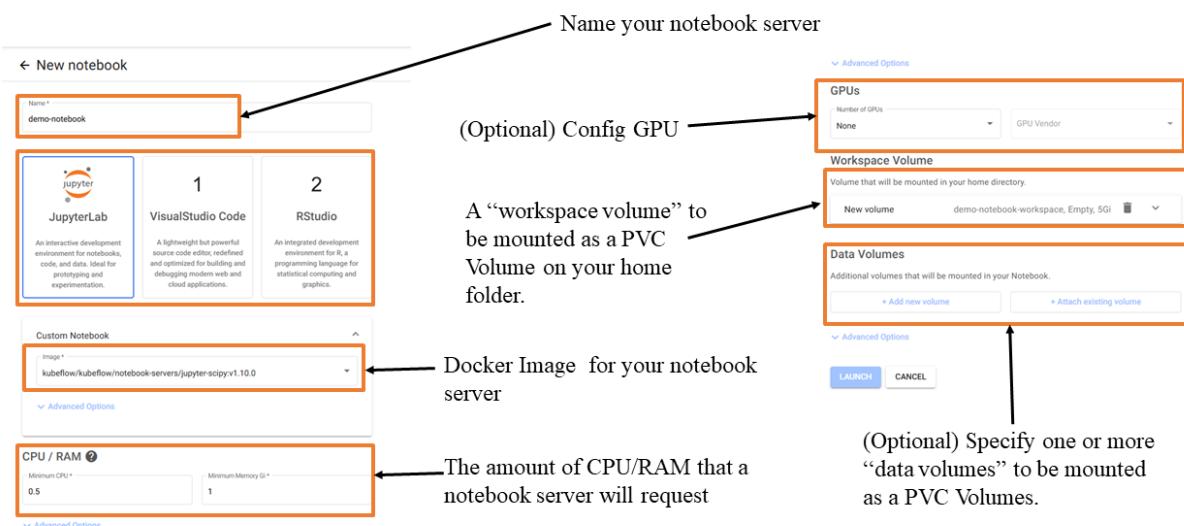
Để sử dụng Kubeflow Notebooks (Trên Dashboard), chúng ta sẽ cần truy cập vào Kubeflow Dashboard, sau đó chọn "Namespace" mình muốn, tiếp theo sẽ chọn "Notebook" và tạo "New Notebook" như hình số 5:



8

Hình 7: Thực hiện tạo trên giao diện Kubeflow

Tiếp theo chúng ta sẽ cấu hình Notebook theo nhu cầu của người dùng



Hình 8: Cấu hình Notebook cơ bản

Cuối cùng, với Notebook vừa tạo thành công, chúng ta sẽ chọn "Connect" để xem giao diện web do máy chủ notebook cung cấp.

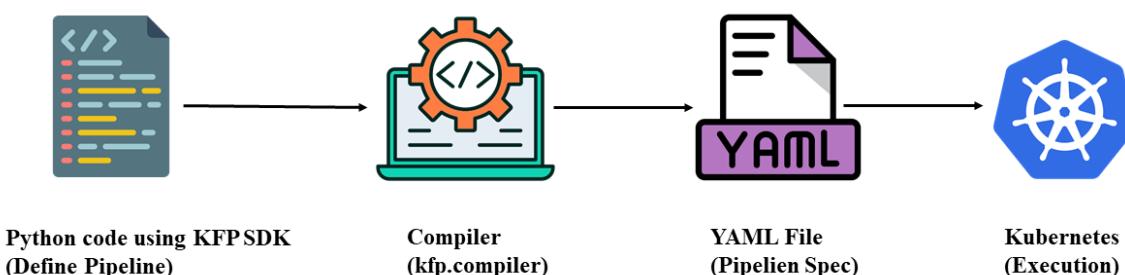
The screenshot shows the Kubeflow UI interface for a notebook named 'demo-notebook'. At the top right, there are buttons for 'CONNECT' (highlighted with a red box), 'STOP', and 'DELETE'. Below the header, there are tabs for 'OVERVIEW', 'LOGS', 'EVENTS', and 'YAML'. The 'OVERVIEW' tab is selected. Under 'Volumes', it lists 'PersistentVolumeClaims' (demo-notebook-workspace) and 'Memory-backed Volumes' (dshm). Configuration details include 'Shared memory enabled' (Yes), 'Notebook creator' (user@example.com), and various resource requirements like CPU and memory. The 'Image' field shows 'ghcr.io/kubeflow/kubeflow/notebook-servers/jupyter-scipy:v1.10.0'. The 'Environment' section includes a 'NB\_PREFIX' environment variable set to '/notebook/kubeflow-user-example-com/demo-notebook'. A 'Conditions' section is also present.

Hình 9: Cấu hình thành công và tiến hành kết nối

#### II.2.4. Kubeflow Pipeline

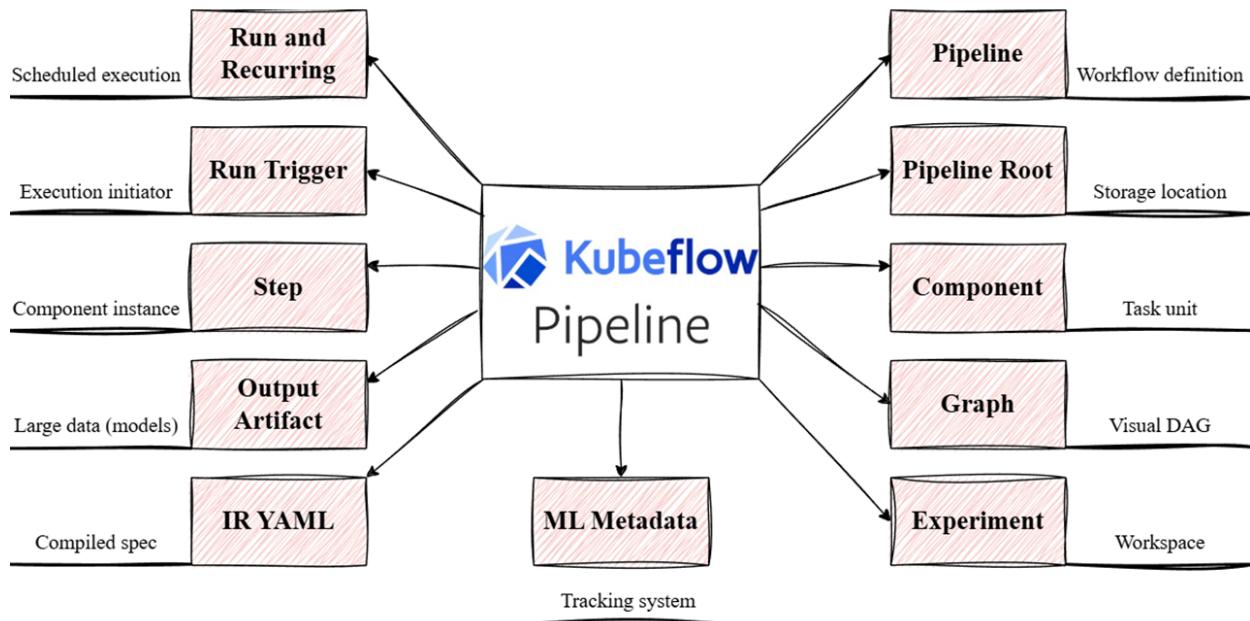
Kubeflow Pipelines (KFP) là nền tảng xây dựng và tự động hóa quy trình học máy (ML Workflow) dựa trên container. KFP cho phép định nghĩa, theo dõi, tái chạy và mở rộng các pipeline trên mọi cụm Kubernetes một cách chuẩn hoá và có khả năng mở rộng linh hoạt.

Quy trình hoạt động chính của Kubeflow Pipelines (KFP) chuyển từ mã nguồn (.py) sang cấu hình (YAML) và cuối cùng là thực thi trên Kubernetes



Hình 10: Tổng quan quy trình hoạt động của Kubeflow Pipeline

Tiếp theo chúng ta sẽ tìm hiểu một số concepts có trong Kubeflow Pipeline:



Hình 11: Các khái niệm cốt lõi trong kiến trúc Kubeflow Pipelines

- **Pipeline:** Đại diện cho một quy trình làm việc (workflow) máy học hoàn chỉnh. Pipeline là một tập hợp các thành phần (components) được sắp xếp theo một trình tự logic, bao gồm định nghĩa về đầu vào, đầu ra và thứ tự thực thi của chúng. Mục tiêu của Pipeline là đóng gói luồng công việc ML để có thể tái sử dụng và chia sẻ.
- **Component:** Đây là đơn vị thực thi cơ bản nhất trong Kubeflow. Một component đại diện cho một bước đơn lẻ trong quy trình (ví dụ: tiền xử lý dữ liệu, huấn luyện mô hình, hoặc đánh giá). Về mặt kỹ thuật, mỗi component là một đoạn mã được đóng gói trong container (Docker image), đảm bảo tính độc lập và nhất quán về môi trường chạy.
- **Graph:** Graph là hình ảnh trực quan hóa của một Pipeline. Trong Kubeflow, Pipeline được biểu diễn dưới dạng Đồ thị có hướng không chu trình (DAG - Directed Acyclic Graph). Các nút (nodes) trong đồ thị đại diện cho các bước (steps/components), và các cạnh (edges) thể hiện sự phụ thuộc dữ liệu hoặc thứ tự thực thi giữa chúng.
- **Step:** Trong khi "Component" là định nghĩa của một chức năng, thì "Step" là một thể hiện (instance) cụ thể của component đó khi được đặt vào trong một Pipeline.
- **Experiment:** Đóng vai trò là một không gian làm việc (workspace) hoặc một nhóm logic dùng để tổ chức và quản lý các lần chạy (Runs). Trong MLOps, việc so sánh các cấu hình mô hình khác nhau là rất quan trọng; Experiment giúp gom nhóm các lần chạy liên quan lại với nhau để dễ dàng theo dõi và so sánh kết quả.
- **Run:** Là một lần thực thi cụ thể của một Pipeline. Khi bạn nhấn "Start" một pipeline, hệ thống sẽ tạo ra một Run.
- **Recurring Run:** Là tính năng cho phép lên lịch tự động cho các Run (ví dụ: chạy pipeline huấn luyện lại mô hình vào 12h đêm hàng ngày).

- **Run Trigger:** Là cơ chế kích hoạt một Run. Trigger có thể là hành động thủ công từ người dùng, kích hoạt theo lịch trình (Scheduled/Recurring), hoặc kích hoạt dựa trên sự kiện (Event-based - ví dụ: khi có dữ liệu mới đẩy lên Git hoặc Object Storage).
- **Pipeline Root:** Đây là đường dẫn gốc đến nơi lưu trữ các tệp kết quả (artifacts) của Pipeline trên hệ thống lưu trữ.
- **Output Artifact:** Là các sản phẩm đầu ra được tạo ra bởi một bước trong pipeline. Artifact có thể là một mô hình đã huấn luyện, một tập dữ liệu đã xử lý, hay các biểu đồ đánh giá. Kubeflow tự động theo dõi và lưu trữ các artifact này vào Pipeline Root.
- **ML Metadata:** Đây là cơ sở dữ liệu lưu trữ thông tin về metadata của quy trình.
- **IR YAML:** Khi người dùng viết code định nghĩa Pipeline (bằng Python SDK), nó sẽ được biên dịch (compile) thành một tệp tin định dạng YAML gọi là IR YAML. Tệp này chứa toàn bộ đặc tả kỹ thuật của Pipeline độc lập với ngôn ngữ lập trình, cho phép Pipeline được gửi đến và thực thi bởi KFP Backend một cách chính xác.

### II.2.5. Kubeflow Trainers

Kubeflow Trainer dành cho việc fine-tuning và cho phép huấn luyện phân tán, có khả năng mở rộng các mô hình ngôn ngữ lớn (LLM) trên nhiều khung AI khác nhau, bao gồm PyTorch, HuggingFace, DeepSpeed, MLX, JAX, XGBoost và các khung khác.

Dưới đây sẽ là một đoạn code huấn luyện mô hình CNN đơn giản và triển khai với Kubeflow Trainers:

#### kubeflow\_trainer\_example.py

```

1 from kubeflow.trainer import TrainerClient, CustomTrainer
2 import time
3 def simple_pytorch_training():
4     import torch
5     import torch.nn as nn
6     import torch.nn.functional as F
7     from torchvision import datasets, transforms
8     from torch.utils.data import DataLoader
9     import os
10
11    rank = int(os.getenv('RANK', 0))
12
13    print("=="*60)
14    print(f"Node {rank}: PyTorch MNIST Training")
15    print("=="*60)
16
17    # Device
18    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
19    print(f"\nNode {rank}: Using device: {device}")
20    print(f"Node {rank}: PyTorch version: {torch.__version__}")
21
22    # 1. Define simple CNN model
23    class SimpleCNN(nn.Module):

```

```
24     def __init__(self):
25         super(SimpleCNN, self).__init__()
26         self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
27         self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
28         self.fc1 = nn.Linear(320, 50)
29         self.fc2 = nn.Linear(50, 10)
30
31     def forward(self, x):
32         x = F.relu(F.max_pool2d(self.conv1(x), 2))
33         x = F.relu(F.max_pool2d(self.conv2(x), 2))
34         x = x.view(-1, 320)
35         x = F.relu(self.fc1(x))
36         x = self.fc2(x)
37         return F.log_softmax(x, dim=1)
38
39 model = SimpleCNN().to(device)
40 num_params = sum(p.numel() for p in model.parameters())
41 print(f"Node {rank}: Model created ({num_params:,} parameters)")
42
43 # 2. Load MNIST dataset
44 print(f"\nNode {rank}: Loading MNIST dataset...")
45
46 transform = transforms.Compose([
47     transforms.ToTensor(),
48     transforms.Normalize((0.1307,), (0.3081,)))
49 ])
50 train_dataset = datasets.MNIST(
51     './data', train=True, download=True, transform=transform
52 )
53 # Use subset for quick demo
54 train_dataset = torch.utils.data.Subset(train_dataset, range(1000))
55
56 train_loader = DataLoader(
57     train_dataset,
58     batch_size=32,
59     shuffle=True
60 )
61
62 print(f"Node {rank}: Dataset loaded ({len(train_dataset)} samples)")
63
64 # 3. Training setup
65 optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
66 criterion = nn.CrossEntropyLoss()
67
68 # 4. Training loop
69 print(f"\nNode {rank}: Starting training...")
70 print("-"*60)
71
72 epochs = 3
73 for epoch in range(epochs):
74     model.train()
75     total_loss = 0
76     correct = 0
77     total = 0
```

```
78
79     for batch_idx, (data, target) in enumerate(train_loader):
80         data, target = data.to(device), target.to(device)
81
82         optimizer.zero_grad()
83         output = model(data)
84         loss = criterion(output, target)
85         loss.backward()
86         optimizer.step()
87
88         total_loss += loss.item()
89         pred = output.argmax(dim=1)
90         correct += pred.eq(target).sum().item()
91         total += target.size(0)
92
93         if batch_idx % 10 == 0:
94             accuracy = 100. * correct / total
95             avg_loss = total_loss / (batch_idx + 1)
96             print(f"Epoch [{epoch+1}/{epochs}] "
97                  f"Batch [{batch_idx}/{len(train_loader)}] "
98                  f"Loss: {avg_loss:.4f} "
99                  f"Acc: {accuracy:.2f}%")
100
101        epoch_acc = 100. * correct / total
102        epoch_loss = total_loss / len(train_loader)
103        print(f"\n{'='*60}")
104        print(f"Epoch {epoch+1} - Loss: {epoch_loss:.4f}, Acc: {epoch_acc:.2f}%")
105        print('='*60 + "\n")
106
107        print(f"\n{'='*60}")
108        print(f"Node {rank}: Training Complete!")
109        print(f"Node {rank}: Final Accuracy: {epoch_acc:.2f}%")
110        print('='*60)
111
112 if __name__ == "__main__":
113
114     print("\n PyTorch MNIST Training Demo")
115     print("=".*80)
116     time.sleep(3)
117
118     # Submit
119     client = TrainerClient()
120
121     job_id = client.train(
122         trainer=CustomTrainer(
123             func=simple_pytorch_training,
124             num_nodes=1,
125             resources_per_node={
126                 "cpu": "500m",
127                 "memory": "1Gi"
128             }
129         )
130     )
131     print(f" Job ID: {job_id}\n")
```

```

132
133     # Monitor
134     print(" Monitoring...")
135     print("*"*80)
136
137     for i in range(60):
138         time.sleep(3)
139
140     try:
141         job = client.get_job(name=job_id)
142
143         if not job.steps:
144             print(f"[{i*3}s] Initializing...", end='\r')
145             continue
146
147         status = job.steps[0].status
148         print(f"[{i*3}s] Status: {status} ", end='\r')
149
150         if status == "Running":
151             print("\n\n Training started!\n")
152             print(" Logs:")
153             print("*"*80)
154
155             for log in client.get_job_logs(job_id, follow=True):
156                 print(log)
157                 break
158
159         elif status == "Succeeded":
160             print("\n\n Completed!\n")
161             for log in client.get_job_logs(job_id, follow=False):
162                 print(log)
163                 break
164
165         elif status == "Failed":
166             print("\n\n Failed!")
167             for log in client.get_job_logs(job_id, follow=False):
168                 print(log)
169                 break
170
171     except KeyboardInterrupt:
172         print("\n\nInterrupted")
173         break
174     except Exception as e:
175         print(f"\n Error: {e}")
176         break

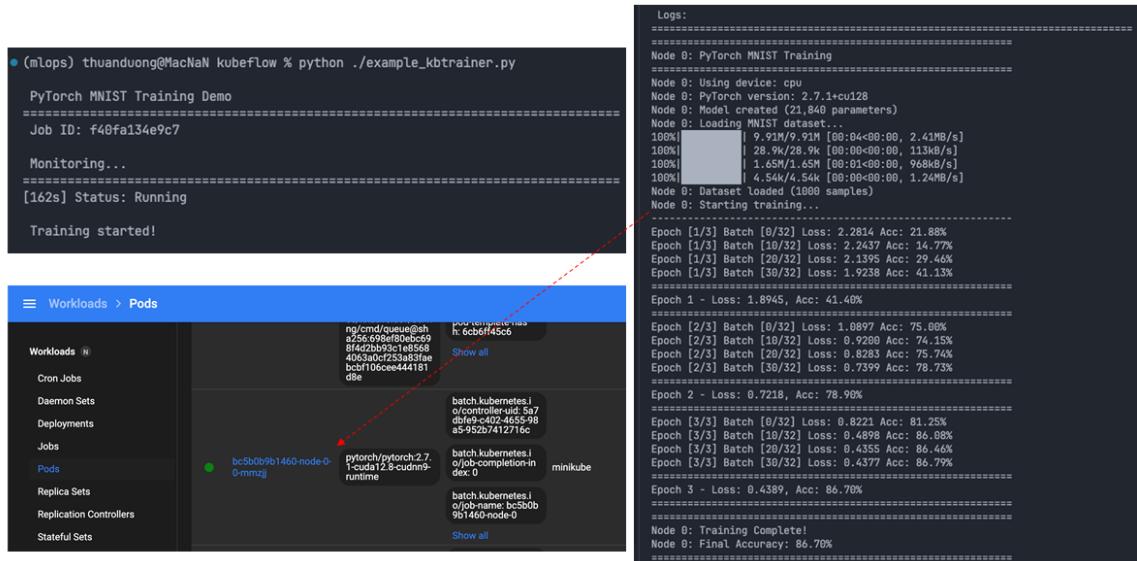
```

Ở đây code định nghĩa kiến trúc mạng SimpleCNN và hàm simple\_pytorch\_training(), chịu trách nhiệm chuẩn bị dữ liệu (sử dụng tập con MNIST 1000 mẫu), nhận diện thiết bị (GPU/CPU), và thực thi vòng lặp huấn luyện với Optimizer Adam.

Điểm cốt lõi nằm ở việc khởi tạo TrainerClient và cấu hình CustomTrainer. Đối tượng này sẽ chịu trách nhiệm bọc hàm simple\_pytorch\_training thành một Training Job, thiết lập hạn mức tài nguyên compute (500m CPU, 1Gi Memory) và điều phối việc khởi chạy Pod trên hạ tầng

Kubernetes mà không cần viết Dockerfile thủ công. Cuối cùng, code thực hiện gửi Job lên cluster và duy trì một vòng lặp giám sát (monitoring loop); cơ chế này liên tục kiểm tra trạng thái Pod và stream logs về client theo thời gian thực, cho phép theo dõi chi tiết tiến độ từ lúc khởi tạo ("Initializing") đến khi hoàn tất ("Succeeded").

Sau khi chạy code, chúng ta sẽ thấy được thông tin huấn luyện như sau:

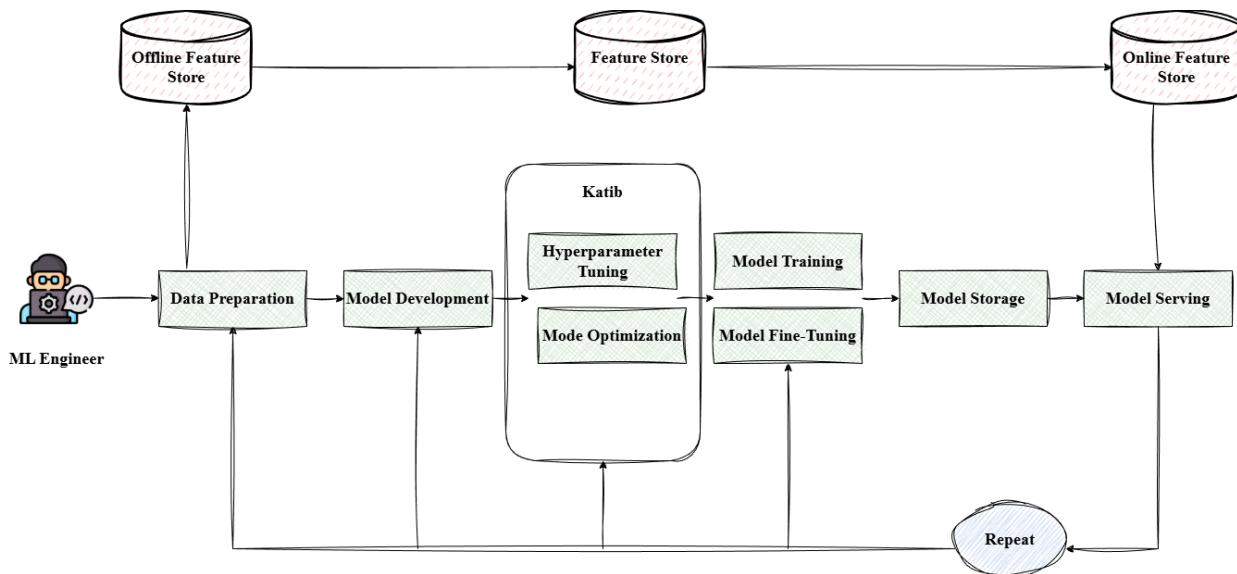


Hình 12: Thông tin huấn luyện mô hình với Kubeflow Trainers

### II.2.6. Kubeflow Katib

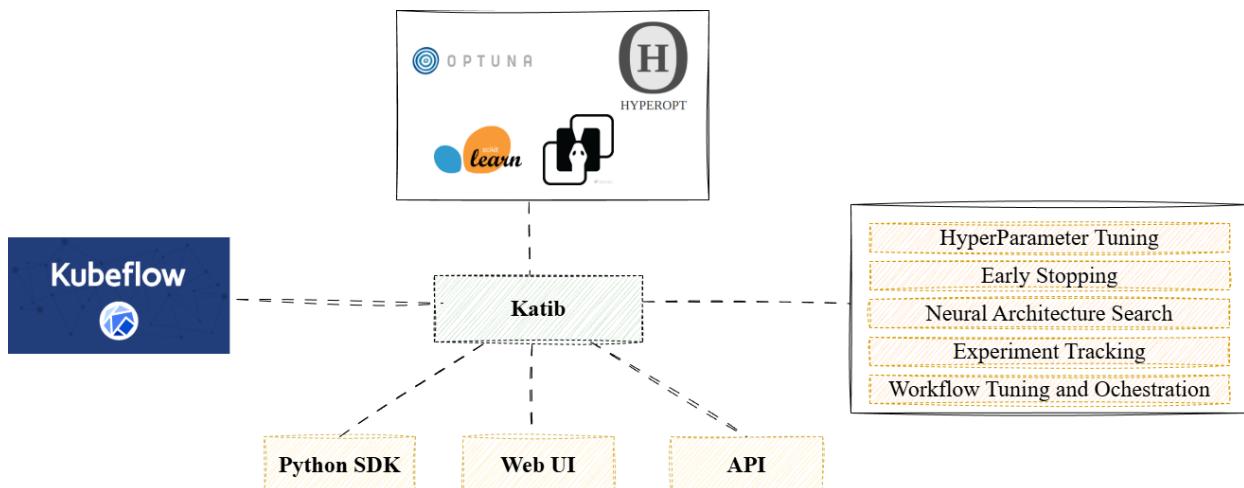
Katib là một dự án được xây dựng trên nền tảng Kubernetes dành cho học máy tự động (AutoML). Katib hỗ trợ điều chỉnh siêu tham số, dừng sớm và tìm kiếm kiến trúc mạng nơ-ron (NAS).

Về khả năng vận hành, Katib thể hiện sức mạnh trong việc điều phối các tác vụ huấn luyện phân tán quy mô lớn (multi-node, multi-GPU). Nó tích hợp chặt chẽ với Kubeflow Training Operator (ví dụ: PyTorchJob) để tối ưu hóa các mô hình lớn, đồng thời hỗ trợ các workflow nâng cao thông qua Argo Workflows và Tekton Pipelines.



Hình 13: Vai trò của Katib trong Kubeflow lifecycle

Ngoài ra, điểm ưu việt của Katib nằm ở tính linh hoạt và khả năng mở rộng: do hoạt động dựa trên container, Katib hoàn toàn độc lập với framework (framework-agnostic). Người dùng có thể sử dụng nó cho bất kỳ thư viện ML nào, hoặc thậm chí là các tác vụ phi máy học (non-ML tasks) miễn là thu thập được chỉ số tối ưu hóa. Ngoài việc tích hợp sẵn các thư viện thuật toán tiên tiến như Hyperopt và Optuna, Katib còn cho phép người dùng tự triển khai và đánh giá (benchmark) các thuật toán tùy chỉnh thông qua Control Plane.



Hình 14: Kiến trúc tổng quan và các tính năng chính của Katib

Ở đây chúng ta sẽ thử một ví dụ sử dụng Kubeflow Katib để tự động tìm best hyperparameters cho MNIST digit classification model:

**example\_katib.py**

```
1 from kubeflow import katib
2 def train_mnist(parameters):
3     import torch
4     import torch.nn as nn
5     import torch.nn.functional as F
6     from torchvision import datasets, transforms
7     from torch.utils.data import DataLoader
8
9     # Get hyperparameters
10    lr = float(parameters["learning_rate"])
11    batch_size = int(parameters["batch_size"])
12    epochs = 3
13
14    print(f"Training with lr={lr}, batch_size={batch_size}")
15
16    # Model
17    class Net(nn.Module):
18        def __init__(self):
19            super().__init__()
20            self.fc1 = nn.Linear(784, 128)
21            self.fc2 = nn.Linear(128, 10)
22
23        def forward(self, x):
24            x = x.view(-1, 784)
25            x = F.relu(self.fc1(x))
26            return F.log_softmax(self.fc2(x), dim=1)
27
28    model = Net()
29
30    # Data
31    transform = transforms.Compose([
32        transforms.ToTensor(),
33        transforms.Normalize((0.1307,), (0.3081,))
34    ])
35
36    train_dataset = datasets.MNIST('./data', train=True, download=True, transform=
37                                  transform)
37    train_dataset = torch.utils.data.Subset(train_dataset, range(1000)) # Subset for
38                                  speed
39
40    train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
41
42    # Training
43    optimizer = torch.optim.Adam(model.parameters(), lr=lr)
44
45    for epoch in range(epochs):
46        model.train()
47        correct = 0
48        total = 0
49
50        for data, target in train_loader:
51            optimizer.zero_grad()
```

```

51         output = model(data)
52         loss = F.nll_loss(output, target)
53         loss.backward()
54         optimizer.step()

55
56         pred = output.argmax(dim=1)
57         correct += pred.eq(target).sum().item()
58         total += target.size(0)

59
60         accuracy = 100. * correct / total
61         print(f"Epoch {epoch+1}: accuracy={accuracy:.2f}")
62         print(f"accuracy={accuracy:.2f}")

63
64 # Search space
65 parameters = {
66     "learning_rate": katib.search.double(min=0.001, max=0.1, step=0.001),
67     "batch_size": katib.search.int(min=16, max=128, step=16),
68 }
69 # Submit experiment
70 client = katib.KatibClient(namespace="kubeflow-user-example-com")
71 client.tune(
72     name="mnist-hpo",
73     objective=train_mnist,
74     parameters=parameters,
75     objective_metric_name="accuracy",
76     objective_type="maximize", # Maximize accuracy
77     max_trial_count=10,
78     parallel_trial_count=2,
79     resources_per_trial={
80         "cpu": "500m", # Reduced from 1 CPU to 0.5 CPU
81         "memory": "1Gi" # Reduced from 2Gi to 1Gi
82     }
83 )
84
85 print("\nWaiting for Experiment to complete...")
86 try:
87     # Wait with timeout
88     client.wait_for_experiment_condition(
89         name="mnist-hpo",
90         namespace="kubeflow-user-example-com",
91         timeout=300
92     )
93
94     print("\nBest hyperparameters:")
95     print(client.get_optimal_hyperparameters("mnist-hpo"))
96 except Exception as e:
97     print(f"\nError while waiting for experiment: {e}")

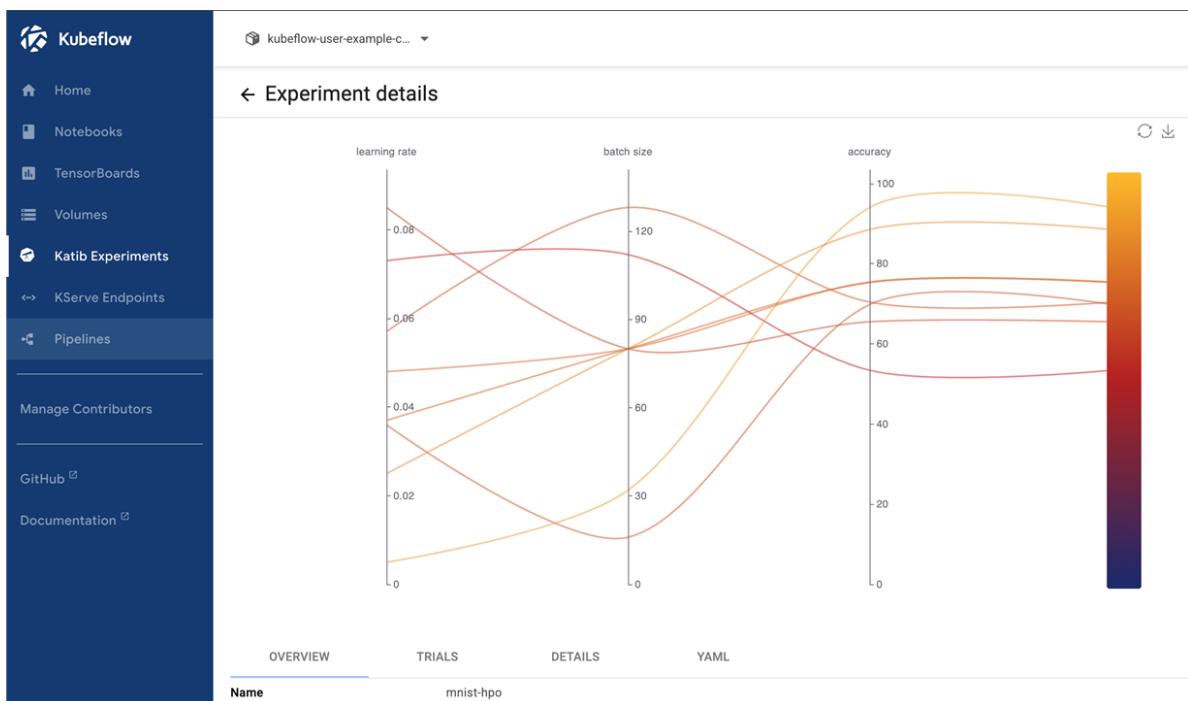
```

Ở đây, chúng ta sẽ có

- Training Function "train\_mnist" định nghĩa workflow: build simple 2-layer neural network và load MNIST dataset (subset 1000 samples), train 3 epochs với Adam optimizer và quan trọng là print accuracy=value ở cuối để Katib parse metric.

- Search Space định nghĩa 2 hyperparameters cần tune: learning\_rate từ 0.001-0.1 và batch\_size từ 16-128.
- Experiment Submission dùng client.tune() để submit: Katib sẽ chạy 10 trials (2 parallel), mỗi trial test different hyperparameter combination, collect accuracy metrics, và maximize objective. Sau khi experiment complete (timeout 300s), get\_optimal\_hyperparameters() retrieve trial có highest accuracy cùng với best hyperparameters configuration.

Sau khi chúng ta chạy code trên và truy cập "Katib Experiments" sẽ thấy được chi tiết Experiment vừa chạy với các thông số về hyperparameters, accuracy qua các lần thử (Hình số 13).



Hình 15: Kiến trúc tổng quan và các tính năng chính của Katib

### II.2.7. Kubeflow KServe

KServe là một dự án mã nguồn mở cung cấp giải pháp Serverless Inference tiêu chuẩn trên nền tảng Kubernetes, cho phép vận hành hiệu năng cao với các framework phổ biến như TensorFlow, PyTorch, XGBoost, scikit-learn và ONNX trong môi trường production.

Ngoài ra, KServe không chỉ đơn giản hóa việc triển khai mô hình mà còn mang lại các tính năng vận hành tiên tiến như: đóng gói toàn bộ sự phức tạp của hạ tầng—từ cấu hình mạng (networking), kiểm tra sức khỏe (health checking), cấu hình server, tự động mở rộng theo nhu cầu (Autoscaling) bao gồm cả GPU và Scale-to-Zero, cũng như chiến lược triển khai Canary. Hơn nữa, KServe cung cấp một quy trình inference hoàn chỉnh và tích hợp sẵn (out-of-the-box) bao gồm các bước: Tiền xử lý (Pre-processing), Dự đoán (Prediction), Hậu xử lý (Post-processing) và Giải thích mô hình (Explainability).

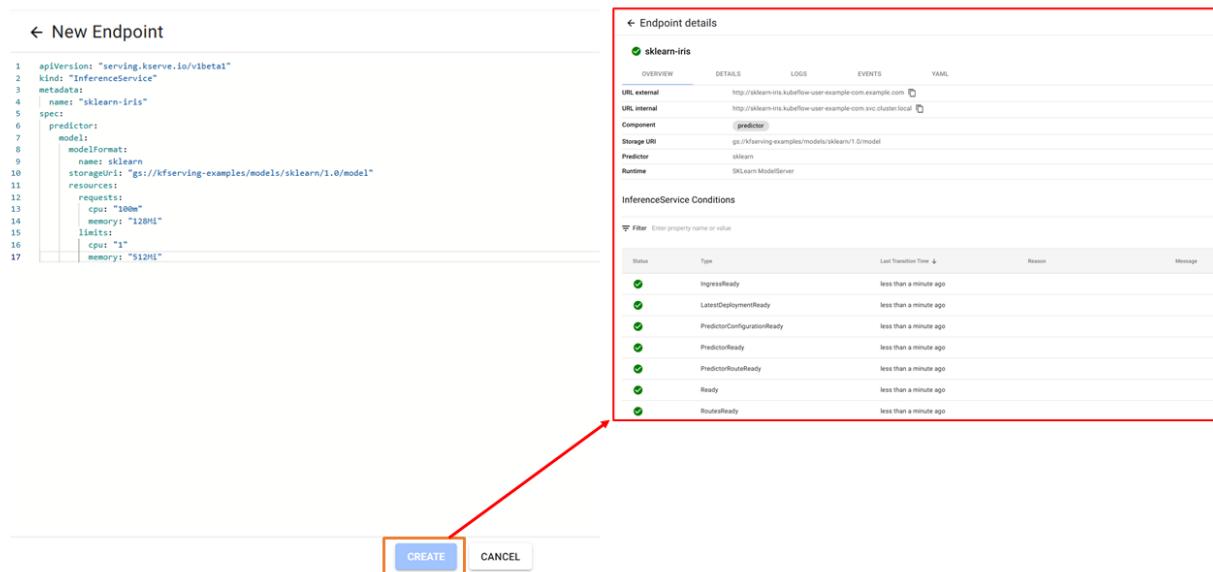
Đoạn mã dưới đây minh họa tệp cấu hình triển khai mô hình:

```
model_serving.yml

1 apiVersion: "serving.kserve.io/v1beta1"
2 kind: "InferenceService"
3 metadata:
4   name: "sklearn-iris"
5 spec:
6   predictor:
7     model:
8       modelFormat:
9         name: sklearn
10      storageUri: "gs://kfserving-examples/models/sklearn/1.0/model"
11      resources:
12        requests:
13          cpu: "100m"
14          memory: "128Mi"
15        limits:
16          cpu: "1"
17          memory: "512Mi"
```

Tệp cấu hình trên định nghĩa một InferenceService cho bài toán phân loại hoa Iris sử dụng thư viện Scikit-learn. Trong đó, mô hình đã huấn luyện được tải trực tiếp từ Google Cloud Storage và được cấp phát giới hạn tài nguyên (CPU, Memory) cụ thể.

Tiếp theo chúng ta sẽ tạo Endpoint với cấu hình code trên trên Dashboard của Kubeflow cụ thể ở mục "Kserve Endpoints" và được kết quả như hình:



Hình 16: Tạo Endpoint trên Kubeflow Dashboard

Trong phạm vi bài viết, chúng ta sẽ chỉ kiểm tra việc dự đoán của mô hình thông qua URL internal, và đây là kết quả khi đưa các instances vào mô hình để đưa ra kết quả dự đoán:

The screenshot shows a Jupyter Notebook interface with a terminal window titled 'Terminal 2'. The terminal output is as follows:

```
(base) jryan@jryan-dell-notebook-01:~$ curl -v http://sklearn-iris.kubeflow-user-example-com.svc.cluster.local/v1/models/sklearn-iris:predict -H "Content-Type: application/json" -d '[{"instances": [[6.8, 2.8, 4.6, 1.4]]}'
* Connected to sklearn-iris.kubeflow-user-example-com.svc.cluster.local (10.96.221.216) port 80 (#0)
> POST /v1/models/sklearn-iris:predict HTTP/1.1
> Host: sklearn-iris.kubeflow-user-example-com.svc.cluster.local
> User-Agent: curl/7.81.0
> Accept: */*
> Content-Type: application/json
> Content-Length: 37
>
< HTTP/1.1 200 OK
< Content-Type: application/json
< content-length: 19
< content-type: application/json
< date: Tue, 10 Dec 2025 19:43:33 GMT
< server: envoy
< x-envoy-upstream-service-time: 11
<
* Connection #0 to host sklearn-iris.kubeflow-user-example-com.svc.cluster.local left intact
[{"predictions": "[1]"}](base) jryan@jryan-dell-notebook-01:~$
```

An annotation with an arrow points from the text 'Test with URL internal' to the first line of the terminal output. Another annotation with an arrow points from the text 'Prediction' to the JSON response at the bottom of the output.

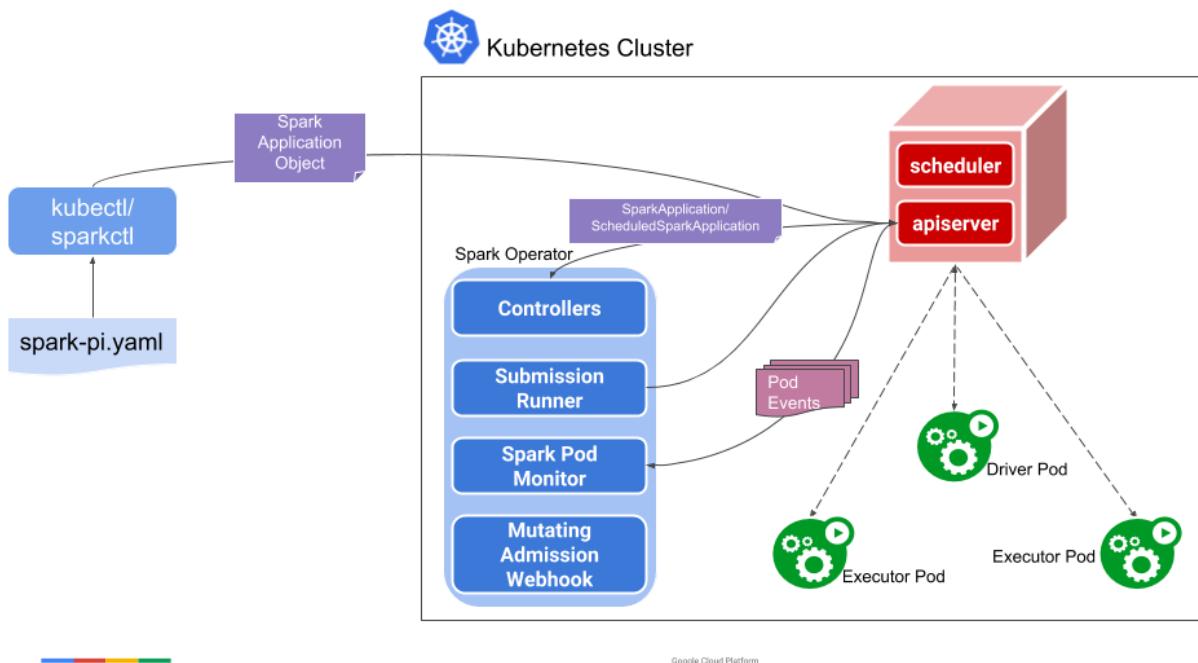
Hình 17: Kiểm tra dự đoán của mô hình với URL internal

### II.2.8. Kubeflow Spark Operator

**Kubeflow Spark Operator** là một thành phần cho phép người dùng triển khai và quản lý các ứng dụng Apache Spark trực tiếp trên cụm Kubernetes theo mô hình khai báo (declarative). Thay vì phải tự cấu hình Spark thủ công bằng script hoặc CLI, Spark Operator giúp người dùng mô tả toàn bộ job Spark trong một tệp YAML và để Kubernetes thực thi, theo dõi và mở rộng tự động.

Spark Operator phù hợp cho các yêu cầu như:

- Tiền xử lý dữ liệu (Data Preprocessing) và Làm sạch dữ liệu (Data Cleansing)
- Feature Engineering và Feature Aggregation cho ML
- Chạy ETL/ELT trên các nguồn dữ liệu lớn (Data Lake / Object Storage)
- Kết nối với Streaming framework (Kafka, Pulsar, Kinesis) để xử lý dữ liệu real-time
- Thực thi batch job định kỳ phục vụ Training Pipeline hoặc Data Pipeline



Hình 18: Kiến trúc tổng quan của Spark Application trong K8s

Một số ưu điểm nổi bật:

- **Declarative Spark Job:** Định nghĩa Spark job bằng YAML thay vì chạy câu lệnh `spark-submit`.
- **Auto-Scaling:** Tự động mở rộng số lượng Executor dựa theo tài nguyên cần thiết.
- **Tối ưu GPU & Accelerator:** Hỗ trợ Spark RAPIDS (GPU) cho workload tăng tốc trên CUDA.
- **Dễ tích hợp với Kubeflow Pipelines:** Dùng như một bước (component) trong pipeline ETL/ML.
- **Khả năng theo dõi & Logs gọn trên UI:** Theo dõi trạng thái job thông qua Kubeflow Dashboard hoặc Spark UI.

### II.2.9. Kubeflow Model Registry

**Kubeflow Model Registry** là hệ thống quản lý vòng đời mô hình (Model Lifecycle Management) trong Kubeflow, được dùng để lưu trữ, theo dõi, phân loại và phiên bản hóa các mô hình sau khi huấn luyện. Nó đóng vai trò như “Model Hub nội bộ” cho dự án MLOps, nơi các mô hình từ nhiều pipeline hoặc nhóm phát triển khác nhau được lưu lại và phân phối theo chuẩn thống nhất.

Trong kiến trúc Kubeflow, Model Registry thường được sử dụng kết hợp với:

- **Pipeline Runner:** lưu mô hình sau khi huấn luyện tự động
- **ML Metadata Store (MLMD):** theo dõi lineage, metrics và artifacts

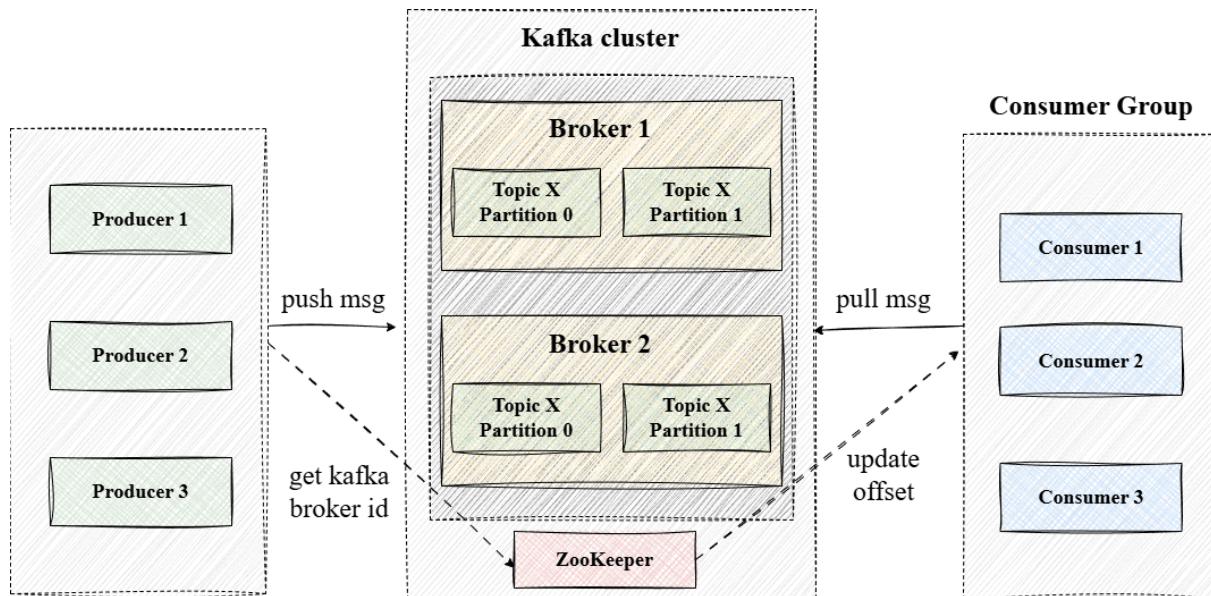
- **KServe:** triển khai mô hình sang môi trường inference
- **CI/CD (Argo, GitHub Actions):** tự động đẩy mô hình mới sang staging/production

Các thành phần quan trọng trong Model Registry:

- **Model Artifact:** file mô hình (ONNX, TorchScript, TensorFlow SavedModel, v.v.)
- **Model Version:** mỗi lần huấn luyện lại sẽ tạo phiên bản mới
- **Model Lineage:** quan hệ giữa dữ liệu → huấn luyện → mô hình → triển khai
- **Performance Metrics:** accuracy, F1, latency... được gắn cùng phiên bản
- **Promotion Stage:** trạng thái của mô hình: draft → staging → production

### II.3. Apache Kafka

Apache Kafka hay Kafka là kho lưu trữ dữ liệu phân tán được tối ưu hóa để nhập và xử lý dữ liệu truyền phát trong thời gian thực. Dữ liệu truyền trực tuyến là dữ liệu được tạo liên tục bởi hàng nghìn nguồn dữ liệu, thường gửi các bản ghi dữ liệu cùng một lúc.



Hình 19: Kafka Architecture with ZooKeeper

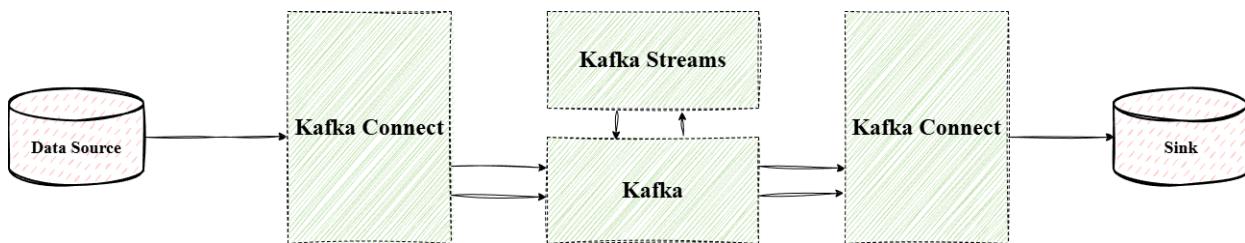
Bây giờ chúng ta sẽ cùng nhau tìm hiểu về các thành phần cốt lõi của kiến trúc Kafka:

- **Kafka Cluster:** Một hệ thống phân tán gồm nhiều Kafka broker, đảm bảo khả năng chịu lỗi, khả năng mở rộng và tính sẵn sàng cao cho việc truyền dữ liệu theo thời gian thực.
- **Broker:** Các máy chủ Kafka xử lý việc lưu trữ dữ liệu, các thao tác đọc/ghi và quản lý việc sao chép dữ liệu để đảm bảo độ tin cậy.

- **Topics & Partitions:** Dữ liệu được tổ chức thành các topic (logic channels), được chia thành các partition để xử lý song song và mở rộng theo chiều ngang.
- **Producers:** Các ứng dụng client ghi dữ liệu vào Kafka topics, phân phối các bản ghi trên các partition.
- **Consumers:** Các ứng dụng đọc dữ liệu từ các topic; ngoài ra "Consumer Group" cho phép cân bằng tải và khả năng chịu lỗi.
- **ZooKeeper:** Quản lý và điều phối các Kafka broker, xử lý cấu hình, đồng bộ hóa và bầu chọn leader.
- **Offsets:** Mã định danh duy nhất cho mỗi tin nhắn trong một partition, được người dùng sử dụng để theo dõi tiến trình đọc.

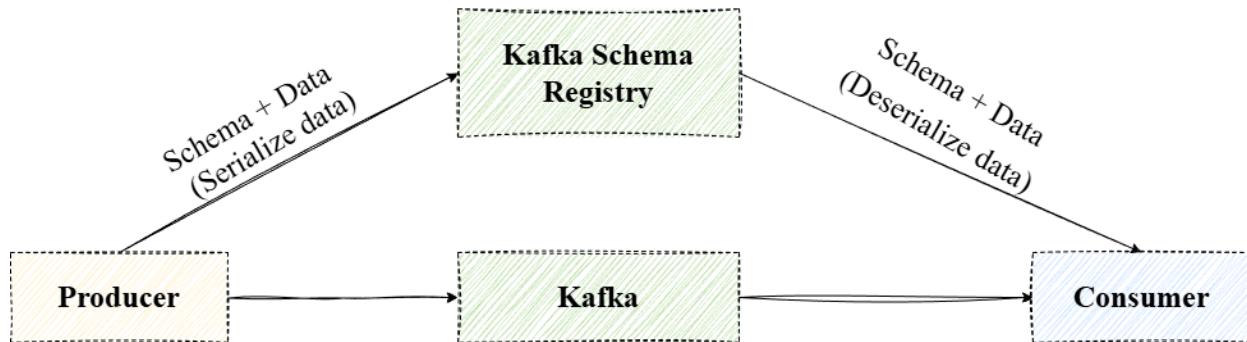
Kafka là một nền tảng truyền phát dữ liệu phân tán có thể được mở rộng và tích hợp với nhiều framework khác nhau để tăng cường khả năng và kết nối với các hệ thống khác. Một số framework quan trọng trong hệ sinh thái Kafka bao gồm:

- **Kafka Connect:** Một công cụ trong hệ sinh thái Kafka, cho phép tích hợp dữ liệu đáng tin cậy và có khả năng mở rộng giữa Kafka và các hệ thống bên ngoài như cơ sở dữ liệu hoặc hệ thống tệp. Nó cung cấp các trình kết nối tích hợp sẵn để đơn giản hóa quá trình di chuyển dữ liệu vào và ra khỏi Kafka.
- **Kafka Streams:** Một thư viện client để xây dựng các ứng dụng xử lý và phân tích dữ liệu trong các topic Kafka. Nó cung cấp các API để sử dụng cho các tác vụ như lọc, kết hợp và tổng hợp dữ liệu luồng.



Hình 20: Kafka Connect and Streams

- **Schema Registry:** Một dịch vụ tập trung quản lý các lược đồ Avro cho các Kafka messages, đảm bảo producers và consumers sử dụng các định dạng dữ liệu tương thích trong quá trình serialization và deserialization.



Hình 21: Kafka Schema Registry

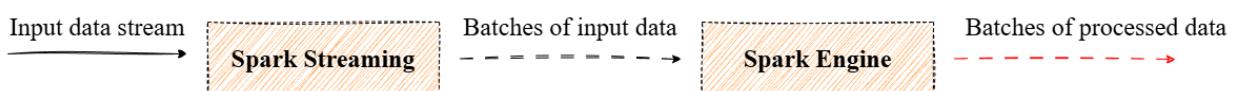
## II.4. Spark Streaming

Spark Streaming là một extension của Spark API, cho phép xử lý luồng dữ liệu trực tiếp với khả năng mở rộng, high-throughput và chịu lỗi. Dữ liệu có thể được thu thập từ nhiều nguồn như Kafka, Kinesis hoặc socket TCP, và có thể được xử lý bằng các thuật toán phức tạp với các high-level functions như map, reduce, join và window. Cuối cùng, dữ liệu đã xử lý có thể được đẩy ra các hệ thống file system, cơ sở dữ liệu và live dashboards. Trên thực tế, người có thể áp dụng các thuật toán học máy và xử lý đồ thị của Spark trên các luồng dữ liệu trực tuyến.



Hình 22: Kiến trúc Event-Driven Workflow của Github Actions

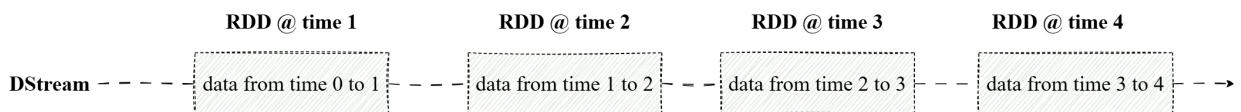
Spark Streaming nhận dữ liệu đầu vào và chia chúng thành các batches, sau đó chúng được xử lý bởi Spark engine để tạo ra luồng kết quả cuối cùng theo batches.



Hình 23: Kiến trúc Event-Driven Workflow của Github Actions

Spark Streaming cung cấp một khái niệm trừu tượng cấp cao gọi là luồng rời rạc (discretized stream) hay DStream, đại diện cho một luồng dữ liệu liên tục. DStream có thể được tạo ra từ các luồng dữ liệu đầu vào từ các nguồn như Kafka và Kinesis, hoặc bằng cách áp dụng các high-level operations trên các DStream khác. Về bản chất, DStream là một chuỗi các RDD (Resilient distributed dataset) liên tiếp.

Mỗi RDD trong DStream chứa dữ liệu từ một khoảng thời gian nhất định, như được minh họa trong hình dưới đây:



Hình 24: Kiến trúc Event-Driven Workflow của Github Actions

Tiếp theo chúng ta sẽ cùng tìm hiểu một số DStream Operations trong Spark Stream (Mọi người có thể tìm hiểu thêm về DStream Operations qua mục Tài liệu tham khảo).

#### II.4.1. Input DStreams và Receivers

Input DStreams là DStreams đại diện cho luồng dữ liệu đầu vào nhận được từ các nguồn phát trực tuyến. Mọi input DStream (ngoại trừ luồng file) đều được liên kết với đối tượng Receiver để nhận dữ liệu từ source và lưu trữ trong bộ nhớ của Spark để xử lý.

Spark Streaming cung cấp hai loại built-in streaming sources:

- **Basic Sources:** Các nguồn có sẵn trực tiếp trong API StreamingContext. Ví dụ đọc file từ HDFS/S3.
  - **Advanced sources:** Các source cần thêm các thư viện bổ trợ như Kafka, Kinesis, v.v. có sẵn thông qua các lớp tiện ích bổ sung.

Như đã đề cập ở trên, riêng File Stream (đọc dữ liệu từ thư mục) là source duy nhất không cần Receiver. Spark sẽ trực tiếp theo dõi sự thay đổi của thư mục hệ thống để nạp dữ liệu.

Dưới đây là ví dụ code về hai nguồn khác nhau:

## example.py

```
1 # 1. Nguồn Socket (Cần Receiver - chiếm 1 Core)
2 lines = ssc.socketTextStream("localhost", 9999)
3
4 # 2. Nguồn File (Không cần Receiver - không chiếm thêm Core)
5 # Spark sẽ quét thư mục này mỗi khi có file mới xuất hiện
6 file_lines = ssc.textFileStream("hdfs://path/to/logs/")
```

## II.4.2. Transformations on DStreams

Các phép biến đổi (Transformations) cho phép người dùng thay đổi, lọc hoặc tính toán trên dữ liệu nhận được từ luồng đầu vào. DStream hỗ trợ hầu hết các phép toán quen thuộc trên RDD của Spark thông thường, nhưng được chia thành các cấp độ xử lý khác nhau:

- **Basic operations:** Các phép toán này hoạt động trên từng RDD (từng batch dữ liệu) một cách độc lập. Nếu người dùng đã quen với Spark Core, phần này hoàn toàn tương tự. Có thể kể đến một số operations tiêu biểu như: map(func) - Trả về DStream mới bằng cách áp dụng hàm func lên từng phần tử, filter(func) - Trả về DStream mới chỉ chứa các bản ghi thỏa mãn điều kiện func trả về true.
- **Window Operations:** Đây là tính năng đặc trưng nhất của Streaming. Thay vì chỉ xử lý dữ liệu trong 1 giây (1 batch), người dùng có thể tính toán trên một khoảng thời gian dài hơn (ví dụ: 30 giây vừa qua). Để sử dụng nhóm này, người dùng cần hiểu hai tham số:
  - **Window length:** Độ dài thời gian của cửa sổ (xem lại bao lâu).
  - **Sliding interval:** Tần suất thực hiện tính toán (bao lâu cập nhật kết quả một lần).
  - Cần lưu ý thêm cả hai tham số này phải là bội số của Batch interval.
- **UpdateStateByKey Operation:** Trong nhiều trường hợp, người dùng cần "ghi nhớ" kết quả từ quá khứ đến hiện tại (ví dụ: đếm tổng số lượt truy cập từ lúc ứng dụng bắt đầu chạy). UpdateStateByKey operation được sinh ra cho việc này.
  - Về cơ chế, chúng ta có thể hiểu nó lấy "Kết quả cũ" cộng với "Dữ liệu mới nhận được" để tạo ra "Kết quả mới nhất".
  - Điều kiện bắt buộc là người dùng phải cấu hình Checkpointing để lưu trữ trạng thái này, phòng trường hợp hệ thống bị sập thì Spark có thể khôi phục lại con số đang đếm dở.
- **Transform operations:** Hàm transform (và transformWith) cho phép áp dụng bất kỳ hàm RDD-to-RDD nào lên DStream. Điều này cực kỳ mạnh mẽ vì nó cho phép người dùng sử dụng các chức năng của RDD mà DStream API không hỗ trợ trực tiếp. Ứng dụng phổ biến thường được thấy chính là kết hợp dữ liệu stream với một dataset tĩnh.

## II.4.3. Output Operations on DStreams

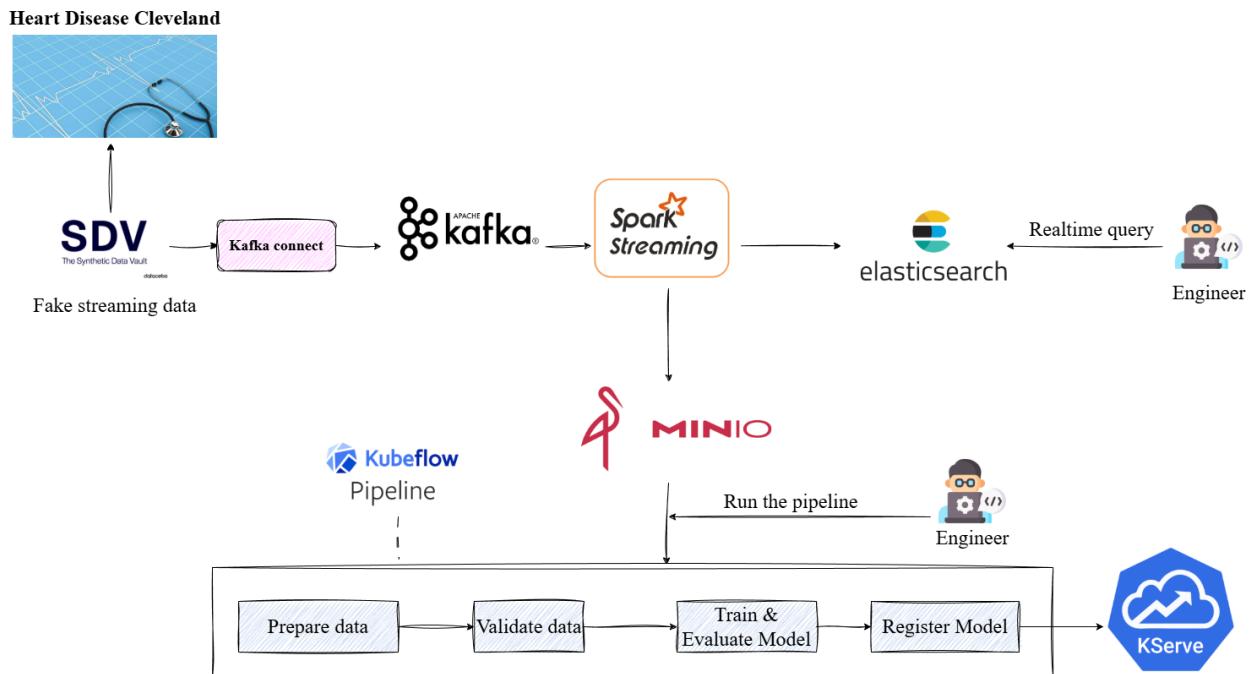
Output operations cho phép dữ liệu của DStream được đẩy ra các hệ thống bên ngoài như cơ sở dữ liệu (MySQL, MongoDB), hệ thống tệp (HDFS, S3) hoặc các bảng điều khiển (Dashboards). Các Output operations phổ biến:

- **pprint():** In 10 phần tử đầu tiên của mỗi batch dữ liệu lên màn hình console. Đây là lệnh cực kỳ hữu ích trong quá trình phát triển và kiểm thử (Debug).
- **saveAsTextFiles(prefix, [suffix]):** Lưu nội dung của DStream thành các tệp văn bản. Tên tệp sẽ được tạo tự động dựa trên thời gian của từng batch (ví dụ: prefix-TIME\_IN\_MS.suffix).
- **foreachRDD(func):** Đây là lệnh quan trọng và linh hoạt nhất. Nó cho phép người dùng áp dụng một hàm tùy ý lên mỗi RDD được tạo ra từ DStream. Thông thường, hàm này để gửi dữ liệu qua mạng hoặc lưu vào Database.

## III. Thực hành:

### III.1. Giới thiệu bài thực hành

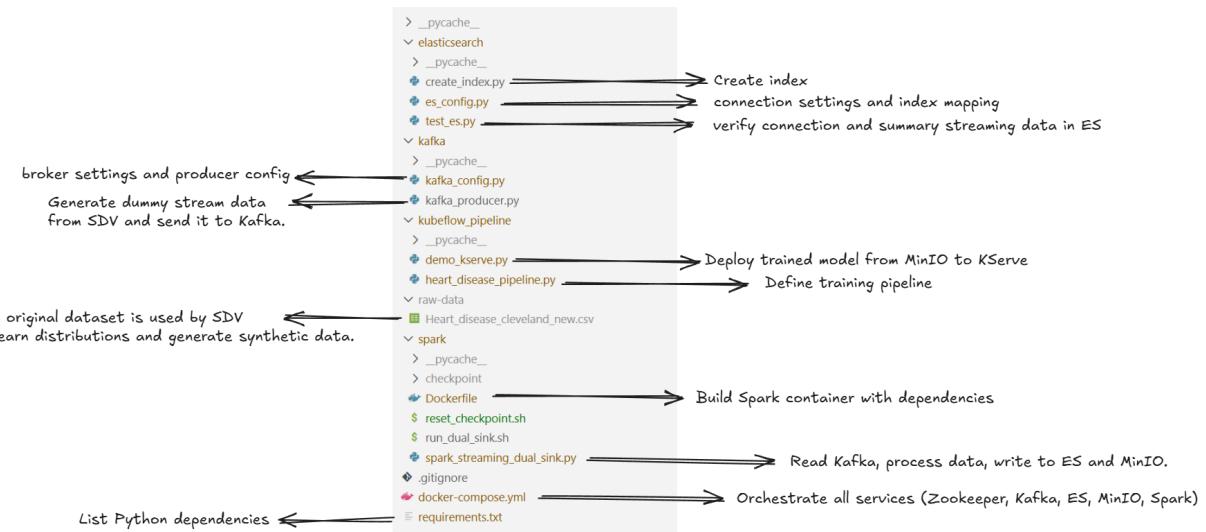
Với bài thực hành lần này, chúng ta sẽ tạo một hệ thống nhỏ tích hợp xử lý dữ liệu streaming real-time với quy trình machine learning tự động hóa. Hệ thống được thiết kế để xử lý dữ liệu bệnh tim từ dataset Cleveland, từ khâu thu thập dữ liệu cho đến triển khai model.



Hình 25: Cấu trúc tổng thể của bài thực hành

Cụ thể kiến trúc được xây dựng sau:

- **Data Ingestion:** SDV tạo fake stream data mô phỏng dữ liệu từ dataset, đẩy vào Kafka thông qua Kafka Connect.
- **Streaming Processing:** Kafka làm message broker trung tâm, Spark Streaming đọc và xử lý dữ liệu real-time. Output được chia làm hai luồng: Elasticsearch cho realtime query và monitoring, MinIO làm data lake lưu trữ training data.
- **ML Pipeline:** Developer manually trigger Kubeflow Pipeline từ MinIO khi cần retrain. Pipeline gồm bốn bước tuần tự: Prepare Data (preprocessing, feature engineering), Validate (data quality checks), Training and Evaluate Model, và Register model.
- **Model Serving:** KServe triển khai model đã được huấn luyện, đánh giá và lưu trữ trên MinIO.



Hình 26: Cấu trúc thư mục và files của bài thực hành

## III.2. Các bước thực hiện

### III.2.1. Cấu hình Docker Compose, Elasticsearch, Minio

Ở đây trước tiên sẽ cấu hình Dockerfile cụ thể để khởi tạo Spark Container:

**Dockerfile**

```

1 FROM spark:3.4.4-python3
2 USER root
3 RUN pip install --no-cache-dir pyspark==3.4.4 elasticsearch requests
4 WORKDIR /app
5 COPY spark/*.py /app/spark/
6 COPY spark/*.sh /app/spark/
7 RUN chmod +x /app/spark/*.sh
8 RUN mkdir -p /app/spark/checkpoint && chmod -R 777 /app/spark/checkpoint
9 ENV SPARK_HOME=/opt/spark
10 ENV PATH=$PATH:$SPARK_HOME/bin
11 ENV PYTHONPATH=/app:$PYTHONPATH
12 CMD ["tail", "-f", "/dev/null"]
  
```

Dockerfile tạo một Spark container từ base image "spark:3.4.4-python3". Nó cài đặt các thư viện Python cần thiết (pyspark, elasticsearch, requests), copy các script Python và shell vào /app/spark, và tạo folder checkpoint để Spark Streaming lưu trữ state khi xử lý dữ liệu. Container được giữ chạy liên tục bằng tail -f /dev/null.

Tiếp đến chúng ta tiến hành cấu hình Docker Compose:

**docker-compose.yml**

```
1 services:
2   zookeeper:
3     image: confluentinc/cp-zookeeper:7.4.0
4     hostname: stream-project-zookeeper
5     container_name: stream-project-zookeeper
6     ports:
7       - "12181:2181"
8     environment:
9       ZOOKEEPER_CLIENT_PORT: 2181
10      ZOOKEEPER_TICK_TIME: 2000
11    networks:
12      - stream-network
13
14 kafka:
15   image: confluentinc/cp-kafka:7.4.0
16   hostname: stream-project-kafka
17   container_name: stream-project-kafka
18   depends_on:
19     - zookeeper
20   ports:
21     - "19092:19092"
22     - "9092:9092"
23   environment:
24     KAFKA_BROKER_ID: 1
25     KAFKA_ZOOKEEPER_CONNECT: 'stream-project-zookeeper:2181'
26     KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:
27       PLAINTEXT
28     KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://stream-project-kafka:9092,PLAINTEXT_HOST
29       ://localhost:19092
30     KAFKA_LISTENERS: PLAINTEXT://0.0.0.0:9092,PLAINTEXT_HOST://0.0.0.0:19092
31     KAFKA_INTER_BROKER_LISTENER_NAME: PLAINTEXT
32     KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
33     KAFKA_TRANSACTION_STATE_LOG_MIN_ISR: 1
34     KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR: 1
35     KAFKA_GROUP_INITIAL_REBALANCE_DELAY_MS: 0
36     KAFKA_AUTO_CREATE_TOPICS_ENABLE: 'true'
37   networks:
38     - stream-network
39   healthcheck:
40     test: ["CMD", "kafka-broker-api-versions", "--bootstrap-server", "localhost:9092"]
41     interval: 10s
42     timeout: 10s
43     retries: 5
44
45 kafka-ui:
46   image: provectuslabs/kafka-ui:latest
47   container_name: stream-project-kafka-ui
48   depends_on:
49     - kafka
50   ports:
51     - "18080:8080"
```

```
50    environment:
51      KAFKA_CLUSTERS_0_NAME: stream-project
52      KAFKA_CLUSTERS_0_BOOTSTRAPSERVERS: stream-project-kafka:9092
53      KAFKA_CLUSTERS_0_ZOOKEEPER: stream-project-zookeeper:2181
54  networks:
55    - stream-network
56
57 elasticsearch:
58   image: docker.elastic.co/elasticsearch/elasticsearch:8.11.0
59   container_name: stream-project-elasticsearch
60   environment:
61     - discovery.type=single-node
62     - xpack.security.enabled=false
63     - "ES_JAVA_OPTS=-Xms512m -Xmx512m"
64   ports:
65     - "19200:9200"
66     - "19300:9300"
67   networks:
68     - stream-network
69   healthcheck:
70     test: ["CMD-SHELL", "curl -f http://localhost:9200/_cluster/health || exit 1"]
71     interval: 30s
72     timeout: 10s
73     retries: 5
74   volumes:
75     - es-data:/usr/share/elasticsearch/data
76
77 minio:
78   image: minio/minio:latest
79   container_name: stream-project-minio
80   ports:
81     - "19000:9000"      # API port
82     - "19001:9001"      # Console port
83   environment:
84     MINIO_ROOT_USER: minioadmin
85     MINIO_ROOT_PASSWORD: minioadmin
86   command: server /data --console-address ":9001"
87   networks:
88     - stream-network
89   volumes:
90     - minio-data:/data
91   healthcheck:
92     test: ["CMD", "curl", "-f", "http://localhost:9000/minio/health/live"]
93     interval: 30s
94     timeout: 10s
95     retries: 5
96
97 spark:
98   build:
99     context: .
100    dockerfile: spark/Dockerfile
101   container_name: stream-project-spark
102   depends_on:
103     - kafka
```

```

104      - elasticsearch
105      - minio
106  environment:
107      - KAFKA_BOOTSTRAP_SERVERS=stream-project-kafka:9092
108      - KAFKA_TOPIC=heart-disease-stream
109      - ES_NODES=stream-project-elasticsearch
110      - ES_PORT=9200
111      - ES_INDEX=heart-disease-stream
112      - MINIO_ENDPOINT=http://stream-project-minio:9000
113      - MINIO_ACCESS_KEY=minioadmin
114      - MINIO_SECRET_KEY=minioadmin
115      - MINIO_BUCKET=ml-data
116  networks:
117      - stream-network
118  volumes:
119      - ./spark:/app/spark
120      - ./spark/checkpoint:/app/spark/checkpoint
121  command: tail -f /dev/null # Keep container running
122
123 networks:
124   stream-network:
125     driver: bridge
126
127 volumes:
128   es-data:
129     driver: local
130   minio-data:
131     driver: local

```

Ở đây, Docker Compose orchestrate 6 services làm việc cùng nhau: Zookeeper (port 12181) quản lý Kafka metadata, Kafka (ports 9092 internal, 19092 external) làm message broker nhận streaming data, Kafka UI (port 18080) để visualize topics và messages, Elasticsearch (port 19200) lưu trữ processed data cho realtime query, MinIO (port 19000 API, 19001 console) làm object storage cho ML training data, và Spark build từ Dockerfile để xử lý streaming từ Kafka rồi ghi vào cả ES và MinIO. Tất cả services kết nối qua stream-network, volumes es-data và minio-data persist dữ liệu khi restart, và environment variables config connection strings giữa các services.

Sau khi chạy thành công Docker Compose, chúng ta tiến hành kiểm tra kết nối tới Elasticsearch và tạo Index. Để cập qua một chút về Elasticsearch (ES), chúng ta hiểu đơn giản đó là một công cụ tìm kiếm và phân tích dữ liệu phân tán, mã nguồn mở, dựa trên Apache Lucene, cho phép lưu trữ, tìm kiếm, phân tích dữ liệu (văn bản, số,...) nhanh chóng và gần như thời gian thực.

### es\_config.py

```

1 class ElasticsearchConfig:
2     # Elasticsearch connection
3     ES_HOSTS = ["localhost"]
4     ES_PORT = "19200"
5     ES_HTTP_AUTH = None # No authentication
6

```

```

7 # Index settings
8 INDEX_NAME = "heart-disease-stream"
9
10 # Index settings
11 INDEX_SETTINGS = {
12     "number_of_shards": 1,
13     "number_of_replicas": 0,
14     "refresh_interval": "1s" # Near real-time
15 }
16
17 # Field mappings
18 INDEX_MAPPING = {
19     "properties": {
20         "batch_number": {"type": "integer"},
21         "record_id": {"type": "long"},
22         "timestamp": {
23             "type": "date",
24             "format": "strict_date_optional_time||epoch_millis"
25         },
26         "age": {"type": "integer"},
27         "sex": {"type": "keyword"},
28         "cp": {"type": "keyword"},
29         "trestbps": {"type": "integer"},
30         "chol": {"type": "integer"},
31         "fbs": {"type": "keyword"},
32         "restecg": {"type": "keyword"},
33         "thalach": {"type": "integer"},
34         "exang": {"type": "keyword"},
35         "oldpeak": {"type": "float"},
36         "slope": {"type": "keyword"},
37         "ca": {"type": "keyword"},
38         "thal": {"type": "keyword"},
39         "target": {"type": "keyword"}
40     }
41 }

```

File này định nghĩa toàn bộ cấu hình cho Elasticsearch index. Connection Settings kết nối đến ES tại localhost:19200 không authentication, index name là heart-disease-stream. Index Settings cấu hình 1 shard và 0 replicas phù hợp cho single-node development, refresh\_interval là 1s đảm bảo data available gần real-time sau khi được index.

Field Mappings định nghĩa chính xác data types cho từng field, mapping này rất quan trọng để ES index và query data đúng kiểu, tránh ES tự động infer sai type dẫn đến lỗi aggregation hoặc filtering.

### create\_index.py

```

1 import requests
2 import json
3 from es_config import ElasticsearchConfig
4
5 def create_index():

```

```
6     """Create Elasticsearch index with mapping"""
7     config = ElasticsearchConfig()
8
9     url = f"http://[{config.ES_HOSTS[0]}:{config.ES_PORT}]/[{config.INDEX_NAME}]"
10
11    print("=" * 70)
12    print("CREATING ELASTICSEARCH INDEX")
13    print("=" * 70)
14
15    # Check if index exists
16    print(f"\n[1] Checking if index '{config.INDEX_NAME}' exists...")
17    response = requests.get(url)
18
19    if response.status_code == 200:
20        print(f"Index '{config.INDEX_NAME}' already exists")
21
22        # Ask to delete
23        delete = input("Delete and recreate? (y/n): ")
24        if delete.lower() == 'y':
25            print(f"Deleting index '{config.INDEX_NAME}'...")
26            requests.delete(url)
27            print("Deleted!")
28        else:
29            print("Keeping existing index")
30            return
31
32    # Create index
33    print(f"\n[2] Creating index '{config.INDEX_NAME}'...")
34
35    index_body = {
36        "settings": config.INDEX_SETTINGS,
37        "mappings": config.INDEX_MAPPING
38    }
39
40    response = requests.put(
41        url,
42        headers={"Content-Type": "application/json"},
43        data=json.dumps(index_body)
44    )
45
46    if response.status_code in [200, 201]:
47        print("Index created successfully!")
48        print("\nIndex configuration:")
49        print(json.dumps(index_body, indent=2))
50    else:
51        print(f"Error creating index: {response.status_code}")
52        print(response.text)
53        return
54
55    # Verify index
56    print(f"\n[3] Verifying index...")
57    response = requests.get(url)
58
59    if response.status_code == 200:
```

```

60     print("Index verified successfully!")
61
62
63 def test_connection():
64     """Test Elasticsearch connection"""
65     config = ElasticsearchConfig()
66
67     print("Testing Elasticsearch connection...")
68     url = f"http://[{config.ES_HOSTS[0]}:{config.ES_PORT}]"
69
70     try:
71         response = requests.get(url)
72         if response.status_code == 200:
73             info = response.json()
74             print(f"Connected to Elasticsearch {info['version']['number']}!")
75             return True
76         else:
77             print(f"Failed to connect: {response.status_code}")
78             return False
79     except Exception as e:
80         print(f"Connection error: {e}")
81         print("\nMake sure Elasticsearch is running:")
82         print("  docker-compose up -d elasticsearch")
83         return False
84
85
86 if __name__ == "__main__":
87     if test_connection():
88         create_index()

```

Script này chuẩn bị ES index trước khi chạy Spark Streaming. Test Connection function verify ES đang chạy bằng cách GET root endpoint, hiển thị version nếu thành công, báo lỗi và hướng dẫn start container nếu fail.

Create Index Workflow thực hiện theo 3 bước: check index có tồn tại chưa (nếu có thì hỏi user xóa và recreate), PUT request với body chứa settings và mappings từ config để tạo index mới, và cuối cùng verify bằng GET request confirm index created thành công. Script này bắt buộc chạy trước Spark Streaming để đảm bảo ES có sẵn index với đúng mapping, tránh ES auto-infer types sai gây lỗi khi xử lý data.

Tiếp theo chúng ta sẽ cần truy cập "http://localhost:19001" (UI của MinIO được cấu hình trong Docker Compose) và tiến hành tạo một bucket có tên là "ml-data" (Đăng nhập với username và password: minioadmin).

### III.2.2. Thiết lập luồng stream và xử lí data

Ở bước này chúng ta sẽ cấu hình Kafka producer:

**kafka\_config.py**

```

1 class KafkaConfig:
2     # Kafka broker settings
3     BOOTSTRAP_SERVERS = ['localhost:19092']
4
5     # Topic settings
6     TOPIC_NAME = 'heart-disease-stream'
7     NUM_PARTITIONS = 1
8     REPLICATION_FACTOR = 1
9
10    # Producer settings
11    PRODUCER_CONFIG = {
12        'bootstrap_servers': BOOTSTRAP_SERVERS,
13        'value_serializer': lambda v: v.encode('utf-8'), # JSON string serialization
14        'key_serializer': lambda k: k.encode('utf-8') if k else None,
15        'acks': 'all', # Wait for all replicas to acknowledge
16        'retries': 3,
17        'max_in_flight_requests_per_connection': 1,
18        'compression_type': 'gzip',
19        'api_version': (2, 8, 0),
20    }

```

Code trên định nghĩa toàn bộ cấu hình cho Kafka producer. Broker Settings kết nối đến Kafka tại localhost:19092 (external port từ docker-compose). Topic Settings tạo topic heart-disease-stream với 1 partition và replication factor 1 phù hợp cho single broker.

Ngoài ra, Producer Config cấu hình chi tiết cho Kafka producer: value\_serializer và key\_serializer encode messages thành UTF-8, acks: 'all' đảm bảo message được acknowledge bởi tất cả replicas trước khi confirm, retries: 3 tự động retry nếu gửi fail, max\_in\_flight\_requests\_per\_connection: 1 đảm bảo message ordering (gửi tuần tự), compression\_type: 'gzip' nén data giảm network bandwidth, và api\_version: (2, 8, 0) chỉ định Kafka API version tương thích.

Bước tiếp theo sẽ tiến hành mô phỏng luồng streaming data:

**kafka\_producer.py**

```

1 import os
2 os.environ['KMP_DUPLICATE_LIB_OK'] = 'TRUE'
3 import pandas as pd
4 from sdv.single_table import GaussianCopulaSynthesizer
5 from sdv.metadata import SingleTableMetadata
6 import time
7 import json
8 from datetime import datetime
9 from kafka import KafkaProducer, KafkaAdminClient
10 from kafka.admin import NewTopic
11 from kafka.errors import TopicAlreadyExistsError, KafkaError
12 from kafka_config import KafkaConfig
13
14 class KafkaStreamProducer:

```

```
15     def __init__(self, config_module=None):
16         """Initialize Kafka Producer with stream generator"""
17         self.config = config_module or KafkaConfig()
18         self.producer = None
19         self.synthesizer = None
20         self.original_data = None
21         self.total_sent = 0
22
23     def connect_kafka(self):
24         """Connect to Kafka and create topic if not exists"""
25         print("\n" + "=" * 70)
26         print("KAFKA CONNECTION")
27         print("=" * 70)
28
29     try:
30         # Create Kafka producer
31         print(f"\n[1] Connecting to Kafka at {self.config.BOOTSTRAP_SERVERS}...")
32         self.producer = KafkaProducer(**self.config.PRODUCER_CONFIG)
33         print("Successfully connected to Kafka broker")
34
35         # Create topic manually if not exists
36         print(f"\n[2] Checking topic '{self.config.TOPIC_NAME}'...")
37         try:
38             admin_client = KafkaAdminClient(
39                 bootstrap_servers=self.config.BOOTSTRAP_SERVERS,
40                 client_id='producer_admin',
41                 api_version=(2, 8, 0)
42             )
43
44             topic = NewTopic(
45                 name=self.config.TOPIC_NAME,
46                 num_partitions=self.config.NUM_PARTITIONS,
47                 replication_factor=self.config.REPLICATION_FACTOR
48             )
49
50             try:
51                 admin_client.create_topics([topic], validate_only=False)
52                 print(f"Created topic '{self.config.TOPIC_NAME}'")
53             except TopicAlreadyExistsError:
54                 print(f"Topic '{self.config.TOPIC_NAME}' already exists")
55             except Exception as e:
56                 print(f"Note: {e}")
57                 print("Topic will be auto-created on first message")
58
59             admin_client.close()
60         except Exception as e:
61             print(f"Could not create topic manually: {e}")
62             print("Topic will be auto-created on first message (auto-create enabled")
63
64         return True
65
66     except Exception as e:
67         print(f"Error connecting to Kafka: {e}")
```

```
68     print("\nMake sure Kafka is running:")
69     print(" docker-compose up -d")
70     return False
71
72 def load_and_train_model(self, data_path, categorical_columns):
73     """Load data and train SDV model"""
74     print("\n" + "=" * 70)
75     print("SDV MODEL TRAINING")
76     print("=" * 70)
77
78     print(f"\n[1] Loading dataset from: {data_path}")
79     if not os.path.exists(data_path):
80         raise FileNotFoundError(f"Data file not found: {data_path}")
81
82     self.original_data = pd.read_csv(data_path)
83     print(f"Loaded {len(self.original_data)} records with {len(self.original_data.columns)} columns")
84
85     # Create metadata
86     print("\n[2] Creating metadata...")
87     metadata = SingleTableMetadata()
88     metadata.detect_from_dataframe(self.original_data)
89
90     for col in categorical_columns:
91         if col in self.original_data.columns:
92             metadata.update_column(column_name=col, sdtype='categorical')
93
94     print(f"Configured {len(categorical_columns)} categorical columns")
95
96     # Train model
97     print("\n[3] Training GaussianCopulaSynthesizer...")
98     self.synthesizer = GaussianCopulaSynthesizer(
99         metadata=metadata,
100        enforce_min_max_values=True,
101        enforce_rounding=True
102    )
103
104     self.synthesizer.fit(self.original_data)
105     print("Model training completed!")
106
107 def generate_batch(self, batch_size):
108     """Generate synthetic data batch"""
109     synthetic_data = self.synthesizer.sample(num_rows=batch_size)
110     return synthetic_data
111
112 def send_to_kafka(self, data, batch_number):
113     """Send data batch to Kafka topic"""
114     records_sent = 0
115
116     for index, row in data.iterrows():
117         try:
118             # Create message
119             message = {
120                 'batch_number': batch_number,
```

```
121         'record_id': self.total_sent + records_sent,
122         'timestamp': datetime.now().isoformat(),
123         'data': row.to_dict()
124     }
125
126     # Serialize to JSON
127     message_json = json.dumps(message)
128
129     # Send to Kafka
130     future = self.producer.send(
131         self.config.TOPIC_NAME,
132         value=message_json,
133         key=str(message['record_id'])
134     )
135
136     # Wait for confirmation (optional, can be async)
137     future.get(timeout=10)
138     records_sent += 1
139
140     except KafkaError as e:
141         print(f"Error sending record {records_sent}: {e}")
142     except Exception as e:
143         print(f"Unexpected error: {e}")
144
145     # Ensure all messages are sent
146     self.producer.flush()
147     self.total_sent += records_sent
148
149     return records_sent
150
151 def stream_continuous(self, batch_size=10, interval_seconds=2):
152     """Stream data continuously"""
153     print("\n" + "=" * 70)
154     print("CONTINUOUS STREAMING TO KAFKA")
155     print("=" * 70)
156     print(f"\nTopic: {self.config.TOPIC_NAME}")
157     print(f"Batch size: {batch_size} records")
158     print(f"Interval: {interval_seconds} seconds")
159     print("\nPress Ctrl+C to stop...")
160     print("=" * 70)
161
162     batch_number = 0
163
164     try:
165         while True:
166             batch_number += 1
167             start_time = time.time()
168
169             # Generate synthetic data
170             data = self.generate_batch(batch_size)
171
172             # Send to Kafka
173             sent = self.send_to_kafka(data, batch_number)
174
```

```
175         elapsed = time.time() - start_time
176
177         print(f"\n[Batch {batch_number}] Sent {sent} records to Kafka")
178         print(f"Total sent: {self.total_sent}")
179         print(f"Time: {elapsed:.2f}s")
180         print(f"Timestamp: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}")
181
182     # Sample data
183     if batch_number == 1:
184         print("\nSample record:")
185         print(data.iloc[0].to_dict())
186
187     # Wait for next batch
188     sleep_time = max(0, interval_seconds - elapsed)
189     if sleep_time > 0:
190         time.sleep(sleep_time)
191
192 except KeyboardInterrupt:
193     print("\n\n" + "=" * 70)
194     print("STREAM STOPPED BY USER")
195     print("=" * 70)
196     print(f"Total batches: {batch_number}")
197     print(f"Total records sent: {self.total_sent}")
198     print(f"Topic: {self.config.TOPIC_NAME}")
199
200
201 def close(self):
202     """Close Kafka producer"""
203     if self.producer:
204         self.producer.close()
205         print("\nKafka producer closed")
206
207
208 def main():
209     # Configuration
210     DATA_PATH = "raw-data/Heart_disease_cleveland_new.csv"
211     CATEGORICAL_COLUMNS = ['sex', 'cp', 'fbs', 'restecg', 'exang', 'slope', 'ca', 'thal',
212                           'target']
213
214     # Streaming settings - Continuous mode only
215     BATCH_SIZE = 10          # Records per batch
216     INTERVAL_SECONDS = 2      # Seconds between batches
217
218     # Initialize producer
219     producer = KafkaStreamProducer()
220
221     try:
222         # Connect to Kafka
223         if not producer.connect_kafka():
224             return
225
226         # Load and train model
227         producer.load_and_train_model(DATA_PATH, CATEGORICAL_COLUMNS)
```

```

228     # Start continuous streaming
229     producer.stream_continuous(
230         batch_size=BATCH_SIZE,
231         interval_seconds=INTERVAL_SECONDS
232     )
233
234     except Exception as e:
235         print(f"\nError: {e}")
236         import traceback
237         traceback.print_exc()
238     finally:
239         producer.close()
240
241
242 if __name__ == "__main__":
243     main()

```

Script trên tạo synthetic data từ dataset gốc và stream vào Kafka. Initialization load KafkaConfig, khởi tạo producer và SDV synthesizer để generate fake data. Connect Kafka method kết nối đến broker, tạo topic bằng KafkaAdminClient nếu chưa tồn tại (hoặc rely vào auto-create), và verify connection thành công.

Hàm "load\_and\_train\_model" đọc file CSV Heart Disease Cleveland, tạo metadata với SDV SingleTableMetadata, set categorical columns (sex, cp, fbs...), và train GaussianCopulaSynthesizer để học distribution của data gốc. Từ đó hàm "stream\_continuous" sẽ chạy infinite loop: mỗi 2 giây generate 1 batch 10 records synthetic data, wrap mỗi record với metadata (batch\_number, record\_id, timestamp), serialize thành JSON và send vào Kafka topic. Producer flush sau mỗi batch đảm bảo messages được gửi ngay, print statistics (total sent, elapsed time), và cho phép dừng bằng Ctrl+C. Đây chính là phần SDV → Kafka trong pipeline diagram, mô phỏng continuous streaming data từ medical devices.

Sau khi đã thành công mô phỏng luồng dữ liệu stream, tiếp theo chúng ta sẽ tiến hành xử lý dữ liệu đó với Spark Streaming, cụ thể như sau:

### spark\_streaming\_dual\_sink.py

```

1 import os
2 os.environ['KMP_DUPLICATE_LIB_OK'] = 'TRUE'
3 from pyspark.sql import SparkSession
4 from pyspark.sql.functions import *
5 from pyspark.sql.types import *
6 from datetime import datetime
7
8 class DualSinkStreaming:
9     def __init__(self):
10         # Kafka config
11         self.kafka_bootstrap = os.getenv('KAFKA_BOOTSTRAP_SERVERS', 'stream-project-kafka:9092')
12         self.kafka_topic = os.getenv('KAFKA_TOPIC', 'heart-disease-stream')
13
14         # Elasticsearch config
15         self.es_nodes = os.getenv('ES_NODES', 'stream-project-elasticsearch')

```

```

16     self.es_port = os.getenv('ES_PORT', '9200')
17     self.es_index = os.getenv('ES_INDEX', 'heart-disease-stream')
18
19     # MinIO config
20     self.minio_endpoint = os.getenv('MINIO_ENDPOINT', 'http://stream-project-minio:
21                                     9000')
21     self.minio_access_key = os.getenv('MINIO_ACCESS_KEY', 'minioadmin')
22     self.minio_secret_key = os.getenv('MINIO_SECRET_KEY', 'minioadmin')
23     self.minio_bucket = os.getenv('MINIO_BUCKET', 'ml-data')
24
25     self.checkpoint_location = "/app/spark/checkpoint"
26     self.spark = None
27
28 def create_spark_session(self):
29     """Create Spark session with all required configurations"""
30     print("\n" + "=" * 70)
31     print("INITIALIZING SPARK SESSION (DUAL SINK: ES + MinIO)")
32     print("=" * 70)
33
34     print("\n[1] Configuring Spark session...")
35     print(f"Kafka: {self.kafka_bootstrap}")
36     print(f"Elasticsearch: {self.es_nodes}:{self.es_port}")
37     print(f"MinIO: {self.minio_endpoint}")
38
39     self.spark = SparkSession.builder \
40         .appName("HeartDisease-DualSink-ES-MinIO") \
41         .config("spark.jars.packages",
42                 "org.apache.spark:spark-sql-kafka-0-10_2.12:3.4.4,"
43                 "org.elasticsearch:elasticsearch-spark-30_2.12:8.11.0,"
44                 "org.apache.hadoop:hadoop-aws:3.3.4") \
45         .config("spark.hadoop.fs.s3a.endpoint", self.minio_endpoint) \
46         .config("spark.hadoop.fs.s3a.access.key", self.minio_access_key) \
47         .config("spark.hadoop.fs.s3a.secret.key", self.minio_secret_key) \
48         .config("spark.hadoop.fs.s3a.path.style.access", "true") \
49         .config("spark.hadoop.fs.s3a.impl", "org.apache.hadoop.fs.s3a.S3AFileSystem
50                                         ") \
51         .config("spark.hadoop.fs.s3a.connection.ssl.enabled", "false") \
52         .config("spark.sql.streaming.checkpointLocation", self.checkpoint_location)
53
54         \
55         .config("spark.es.nodes", self.es_nodes) \
56         .config("spark.es.port", self.es_port) \
57         .config("spark.es.nodes.wan.only", "true") \
58         .config("spark.es.index.auto.create", "true") \
59         .getOrCreate()
60
61
62     self.spark.sparkContext.setLogLevel("WARN")
63
64     print(f"\nSpark version: {self.spark.version}")
65     print("Spark session created successfully!")
66
67 def define_schema(self):
68     """Define message schema from Kafka"""
69     data_schema = StructType([
70         StructField("age", FloatType(), True),
71         StructField("sex", IntegerType(), True),
72         StructField("cp", IntegerType(), True),
73         StructField("trestbps", IntegerType(), True),
74         StructField("chol", IntegerType(), True),
75         StructField("fbs", IntegerType(), True),
76         StructField("restecg", IntegerType(), True),
77         StructField("thalach", IntegerType(), True),
78         StructField("exang", IntegerType(), True),
79         StructField("oldpeak", DoubleType(), True),
80         StructField("slope", IntegerType(), True),
81         StructField("thal", IntegerType(), True),
82         StructField("target", IntegerType(), True),
83     ])
84
85     return data_schema

```

```
67     StructField("sex", FloatType(), True),
68     StructField("cp", FloatType(), True),
69     StructField("trestbps", FloatType(), True),
70     StructField("chol", FloatType(), True),
71     StructField("fbs", FloatType(), True),
72     StructField("restecg", FloatType(), True),
73     StructField("thalach", FloatType(), True),
74     StructField("exang", FloatType(), True),
75     StructField("oldpeak", FloatType(), True),
76     StructField("slope", FloatType(), True),
77     StructField("ca", FloatType(), True),
78     StructField("thal", FloatType(), True),
79     StructField("target", FloatType(), True)
80   ])
81
82   message_schema = StructType([
83     StructField("batch_number", IntegerType(), True),
84     StructField("record_id", LongType(), True),
85     StructField("timestamp", StringType(), True),
86     StructField("data", data_schema, True)
87   ])
88
89   return message_schema
90
91 def read_from_kafka(self):
92   """Read streaming data from Kafka"""
93   print("\n" + "=" * 70)
94   print("READING FROM KAFKA")
95   print("=" * 70)
96
97   print(f"\n[2] Connecting to Kafka...")
98   print(f"Bootstrap servers: {self.kafka_bootstrap}")
99   print(f"Topic: {self.kafka_topic}")
100
101   df = self.spark.readStream \
102     .format("kafka") \
103     .option("kafka.bootstrap.servers", self.kafka_bootstrap) \
104     .option("subscribe", self.kafka_topic) \
105     .option("startingOffsets", "latest") \
106     .option("failOnDataLoss", "false") \
107     .load()
108
109   print("Connected to Kafka successfully!")
110   return df
111
112 def process_data(self, df):
113   """Parse and transform data"""
114   print("\n" + "=" * 70)
115   print("PROCESSING DATA")
116   print("=" * 70)
117
118   print("\n[3] Parsing JSON messages...")
119
120   message_schema = self.define_schema()
```

```
121
122     # Parse JSON from Kafka
123     parsed_df = df.selectExpr("CAST(value AS STRING) as json_str") \
124         .select(from_json(col("json_str"), message_schema).alias("data")) \
125         .select("data.*")
126
127     # Flatten and add metadata
128     processed_df = parsed_df.select(
129         col("batch_number"),
130         col("record_id"),
131         to_timestamp(col("timestamp")).alias("event_timestamp"),
132         col("data.age").cast("integer").alias("age"),
133         col("data.sex").cast("integer").alias("sex"),
134         col("data.cp").cast("integer").alias("cp"),
135         col("data.trestbps").cast("integer").alias("trestbps"),
136         col("data.chol").cast("integer").alias("chol"),
137         col("data.fbs").cast("integer").alias("fbs"),
138         col("data.restecg").cast("integer").alias("restecg"),
139         col("data.thalach").cast("integer").alias("thalach"),
140         col("data.exang").cast("integer").alias("exang"),
141         col("data.oldpeak").alias("oldpeak"),
142         col("data.slope").cast("integer").alias("slope"),
143         col("data.ca").cast("integer").alias("ca"),
144         col("data.thal").cast("integer").alias("thal"),
145         col("data.target").cast("integer").alias("target")
146     ) \
147         .withColumn("processed_at", current_timestamp()) \
148         .withColumn("date", to_date(col("event_timestamp"))) \
149         .withColumn("hour", hour(col("event_timestamp")))
150
151     print("Data processing pipeline configured!")
152     print("\nOutput schema:")
153     processed_df.printSchema()
154
155     return processed_df
156
157 def write_dual_sink(self, df):
158     """Write to BOTH Elasticsearch AND MinIO using foreachBatch"""
159     print("\n" + "=" * 70)
160     print("CONFIGURING DUAL SINK (ES + MinIO)")
161     print("=" * 70)
162
163     def foreach_batch_function(batch_df, batch_id):
164         """Process each micro-batch - write to both sinks"""
165
166         record_count = batch_df.count()
167
168         if record_count == 0:
169             print(f"\n[Batch {batch_id}] Empty batch, skipping...")
170             return
171
172         print(f"\n{'=' * 70}")
173         print(f"PROCESSING BATCH {batch_id}")
174         print(f"{'=' * 70}")
```

```

175     print(f"Records: {record_count}")
176     print(f"Time: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}")
177
178     # Show sample record
179     print("\nSample record:")
180     batch_df.select("record_id", "age", "sex", "target").show(1, truncate=False)
181
182     # Elasticsearch (Real-time query)
183     print(f"\n[Batch {batch_id}] Writing to Elasticsearch...")
184     try:
185         batch_df.write \
186             .format("org.elasticsearch.spark.sql") \
187             .option("es.resource", self.es_index) \
188             .option("es.nodes", self.es_nodes) \
189             .option("es.port", self.es_port) \
190             .option("es.nodes.wan.only", "true") \
191             .mode("append") \
192             .save()
193         print(f"Elasticsearch: {record_count} records written")
194     except Exception as e:
195         print(f"Elasticsearch write failed: {e}")
196
197     # MinIO (ML training data)
198     print(f"\n[Batch {batch_id}] Writing to MinIO (Parquet)...")
199     try:
200         # Output path with partitioning
201         output_path = f"s3a://{{self.minio_bucket}}/raw/heart-disease"
202
203         # Write as Parquet with partitioning by date and hour
204         batch_df \
205             .repartition(1) \
206             .write \
207             .format("parquet") \
208             .partitionBy("date", "hour") \
209             .mode("append") \
210             .option("compression", "snappy") \
211             .save(output_path)
212
213         print(f" MinIO: {record_count} records written")
214         print(f"  Path: {output_path}/date=.../hour=.../")
215         print(f"  Format: Parquet (Snappy compressed)")
216     except Exception as e:
217         print(f" MinIO write failed: {e}")
218         import traceback
219         traceback.print_exc()
220
221     print(f"\n[Batch {batch_id}] Completed successfully!")
222     print("=" * 70)
223
224     # Start streaming with foreachBatch
225     query = df.writeStream \
226         .foreachBatch(foreach_batch_function) \
227         .trigger(processingTime="5 seconds") \

```

```

228     .option("checkpointLocation", self.checkpoint_location) \
229     .start()
230 
231     return query
232 
233 def run(self):
234     try:
235         self.create_spark_session()
236         kafka_df = self.read_from_kafka()
237         processed_df = self.process_data(kafka_df)
238         query = self.write_dual_sink(processed_df)
239         query.awaitTermination()
240     except KeyboardInterrupt:
241         print("\n\nStopping streaming...")
242         if self.spark:
243             self.spark.stop()
244         print("Spark session stopped.")
245     except Exception as e:
246         print(f"\n\nError: {e}")
247         import traceback
248         traceback.print_exc()
249         if self.spark:
250             self.spark.stop()
251 
252 if __name__ == "__main__":
253     app = DualSinkStreaming()
254     app.run()

```

Script trên sẽ đọc data từ Kafka, xử lý và ghi đồng thời vào Elasticsearch và MinIO. Initialization load config từ environment variables: Kafka bootstrap servers, ES endpoint, MinIO credentials. Checkpoint location /app/spark/checkpoint lưu streaming state để recover khi restart.

Đầu tiên hàm "create\_spark\_session" cấu hình 3 connectors: spark-sql-kafka để đọc Kafka, elasticsearch-spark để ghi ES, hadoop-aws để ghi MinIO qua S3A protocol. Config S3A với MinIO endpoint, credentials, path-style access (MinIO requirement), và disable SSL. Read from Kafka tạo streaming DataFrame với startingOffsets: latest (chỉ đọc data mới), failOnDataLoss: false để tránh job fail khi Kafka xóa old data.

Tiếp theo sẽ tiến hành parse JSON messages từ Kafka value qua hàm "process\_data", flatten nested "data" object thành flat columns, cast types (integer cho age/sex/target, float cho oldpeak), và add metadata columns: processed\_at (current timestamp), date và hour (extract từ event\_timestamp) để partition data trong MinIO. Hàm "write\_dual\_sink" dùng foreachBatch để process từng micro-batch (trigger mỗi 5 giây): với mỗi batch, ghi vào ES dạng append mode cho realtime query, sau đó ghi vào MinIO format Parquet với Snappy compression, partition theo date/hour để optimize query performance khi train model. Hàm sẽ in statistics mỗi batch (record count, sample data, write status) để monitor streaming progress. Checkpoint đảm bảo exactly-once semantics khi restart.

Cuối cùng chúng ta sẽ chạy file: kafka\_producer.py và file: run\_dual\_sink.sh (Qua Container với câu lệnh "docker exec stream-project-spark /bin/bash /app/spark/run\_dual\_sink.sh") để tiến

hành tạo luồng dữ liệu stream và execute Spark Streaming job trong container để xử lí dữ liệu.

### run\_dual\_sink.sh

```

1 export SPARK_HOME=${SPARK_HOME:-/opt/spark}
2
3 # Define Spark packages
4 SPARK_PACKAGES="org.apache.spark:spark-sql-kafka-0-10_2.12:3.4.4,org.elasticsearch:
5                           elasticsearch-spark-30_2.12:8.11.0,org.
6                           apache.hadoop:hadoop-aws:3.3.4"
7
8 # Execute the Spark job
9 exec "${SPARK_HOME}/bin/spark-submit" \
10   --packages "${SPARK_PACKAGES}" \
11   --master local[*] \
12   /app/spark/spark_streaming_dual_sink.py

```

Sau khi chạy hai file trên chúng ta sẽ có được luồng dữ liệu liên tục được chạy qua Kafka và được xử lí với Spark Streaming:

The image displays two terminal windows side-by-side, both titled "Windows PowerShell".

**Left Terminal Window (Kafka Processing):**

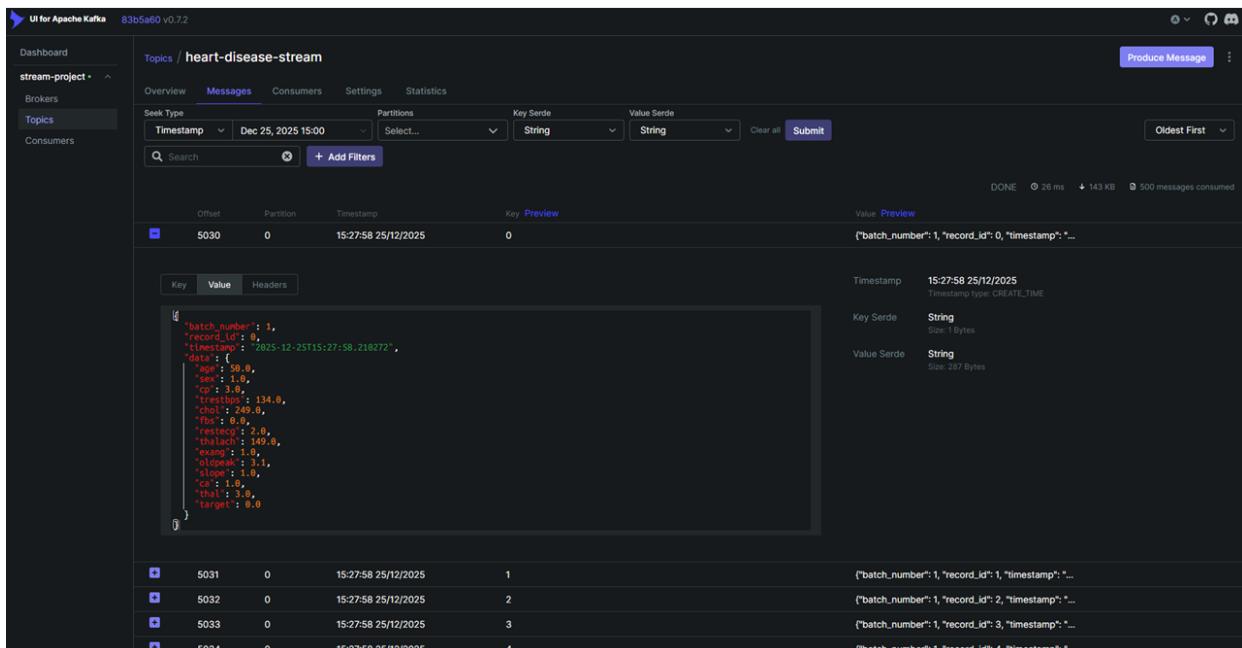
- Logs show batches being sent to Kafka at intervals of approximately 1 second.
- Each batch contains 10 records.
- Sample records are shown for each batch, including fields like age, sex, cp, chol, fbs, restecg, thal, and target.
- Timestamps range from 2025-12-25 15:27:58 to 15:28:08.

**Right Terminal Window (Data Writing):**

- Logs show batches being processed and written to Elasticsearch and MinIO.
- Batch 0 completed successfully, taking 0.095 seconds.
- Batch 1 took 8.275 milliseconds due to falling behind.
- Logs for Elasticsearch and MinIO show 30 records written per batch.
- Sample records for Parquet format are shown, indicating a schema with record\_id, age, sex, and target.
- Timestamps range from 2025-12-25 08:28:12 to 08:28:15.

Hình 27: Terminals hiển thị luồng streaming data

Mọi người có thể truy cập vào Kafka UI để xem chi tiết thông tin về luồng chạy hiện tại



Hình 28: Thông tin về Topic và Messages trên Kafka UI

Để kiểm chứng, chúng ta sẽ thử chạy file test\_es.py để kiểm tra dữ liệu đã được ghi vào trong Elasticsearch hay chưa, và xem thông tin về thông tin dữ liệu:

The CLI output is divided into several sections:

- INDEX INFORMATION**: Shows the index 'heart-disease-stream' exists with 1 shard, 0 replicas, and 21 fields.
- DOCUMENT COUNT**: Shows a total of 1300 documents.
- LATEST 5 DOCUMENTS**: Lists the first 5 documents with their IDs, timestamps, ages, sexes, and targets.
- SEARCH: TARGET = 0**: Shows 705 documents with target=0.
- SEARCH: TARGET = 1**: Shows 595 documents with target=1.
- AGGREGATIONS**: Shows target distribution (Target 0: 705 documents, Target 1: 595 documents), sex distribution (Female: 908 documents, Male: 392 documents), and average age (54.53).

Hình 29: Kiến trúc Event-Driven Workflow của Github Actions

Ở đây, chúng ta sẽ kiểm tra thêm truy vấn dữ liệu với Elasticsearch sử dụng URL với thông tin cần truy vấn là "age=63" và kết quả trả về tìm thấy 41 instances tương ứng (Hình số 26).

The screenshot shows a terminal window with the URL `localhost:19200/heart-disease-stream/_search?pretty&q=age:63` in the address bar. The response is a JSON object representing the search results. The object has fields like "took", "timed\_out", "\_shards", "hits" (with total value 45), and "max\_score". The "hits" field contains two document objects, each with "\_index", "\_id", "\_score", and "\_source" fields. The "\_source" field for the first hit includes various heart disease features: batch\_number, record\_id, event\_timestamp, age (highlighted in orange), sex, cp, trestbps, chol, fbs, restecg, thalach, exang, oldpeak, slope, ca, thal, target, processed\_at, date, and hour.

```
{"took": 41,
 "timed_out": false,
 "_shards": {
   "total": 1,
   "successful": 1,
   "skipped": 0,
   "failed": 0
 },
 "hits": {
   "total": {
     "value": 45,
     "relation": "eq"
   },
   "max_score": 1,
   "hits": [
     {
       "_index": "heart-disease-stream",
       "_id": "jZKgVJsBepf9000kFq3V",
       "_score": 1,
       "_source": {
         "batch_number": 29,
         "record_id": 285,
         "event_timestamp": 1766676534460,
         "age": 63,
         "sex": 1,
         "cp": 3,
         "trestbps": 136,
         "chol": 372,
         "fbs": 0,
         "restecg": 0,
         "thalach": 114,
         "exang": 1,
         "oldpeak": 5.1,
         "slope": 1,
         "ca": 1,
         "thal": 3,
         "target": 1,
         "processed_at": 1766651335006,
         "date": 1766620800000,
         "hour": 15
       }
     },
     {
       "_index": "heart-disease-stream",
       "_id": "GJKfvJsBepf9000kyK2r",
       "_score": 1,
       "_source": {
         "batch_number": 17,
         "record_id": 168,
         "event_timestamp": 1766676510389,
         "age": 63,
         "sex": 1,
         "cp": 0,
         "trestbps": 145,
         "chol": 299,
         "fbs": 0,
         "restecg": 0,
         "thalach": 147,
         "exang": 0,
         "oldpeak": 1.4,
         "slope": 1,
         "ca": 1,
         "thal": 1
       }
     }
   ]
 }
```

Hình 30: Tìm kiếm dữ liệu với ES

### III.2.3. Cấu hình Kubeflow pipeline cho quá trình huấn luyện và triển khai mô hình với KServe

Ở phần này, chúng ta sẽ tận dụng dữ liệu được lưu trữ ở MinIO và thực hiện huấn luyện mô hình ML đơn giản qua việc sử dụng Kubeflow Pipeline và triển khai mô hình với KServe:

#### heart\_disease\_pipeline.py

```

1 from kfp import dsl, compiler
2 from kfp.dsl import Artifact, Input, Output
3 from typing import NamedTuple
4
5 # COMPONENT 1: Data Preparation
6 @dsl.component(
7     base_image="python:3.9",
8     packages_to_install=["boto3", "pandas", "pyarrow", "numpy", "scikit-learn"]
9 )
10 def prepare_data(
11     minio_endpoint: str,
12     minio_access_key: str,
13     minio_secret_key: str,
14     bucket_name: str,
15     data_prefix: str,
16     output_data: Output[Artifact]
17 ) -> NamedTuple('Outputs', [('num_records', int), ('num_features', int)]):
18     """Collect + Preprocess + Feature Engineering"""
19     import boto3
20     import pandas as pd
21     import numpy as np
22     import tempfile
23     import os
24     from sklearn.preprocessing import StandardScaler
25     from collections import namedtuple
26
27     print("=" * 70)
28     print("COMPONENT 1: DATA PREPARATION")
29     print("=" * 70)
30
31     # Collect from MinIO
32     s3_client = boto3.client(
33         's3',
34         endpoint_url=minio_endpoint,
35         aws_access_key_id=minio_access_key,
36         aws_secret_access_key=minio_secret_key,
37         verify=False
38     )
39
40     response = s3_client.list_objects_v2(Bucket=bucket_name, Prefix=data_prefix)
41     parquet_files = [obj['Key'] for obj in response.get('Contents', []) if obj['Key'].endswith('.parquet')]
42
43     print(f"Found {len(parquet_files)} parquet files")
44
45     dfs = []
46

```

```

47     for file_key in parquet_files[:10]:
48         tmp_file = tempfile.NamedTemporaryFile(delete=False, suffix='.parquet')
49         tmp_file_path = tmp_file.name
50         print(f"Downloading file: {file_key} to {tmp_file_path}")
51         tmp_file.close()
52         s3_client.download_file(bucket_name, file_key, tmp_file_path)
53         dfs.append(pd.read_parquet(tmp_file_path))
54         os.unlink(tmp_file_path)
55
56     if not dfs:
57         raise Exception("No data found in MinIO!")
58
59     combined_df = pd.concat(dfs, ignore_index=True)
60     print(f"Combined data: {combined_df.shape}")
61
62     # Preprocess
63     feature_cols = ['age', 'sex', 'cp', 'trestbps', 'chol', 'fbs',
64                      'restecg', 'thalach', 'exang', 'oldpeak', 'slope',
65                      'ca', 'thal', 'target']
66     df_clean = combined_df[feature_cols].copy()
67     df_clean = df_clean.fillna(df_clean.median())
68     df_clean = df_clean.drop_duplicates()
69
70     # Remove outliers
71     for col in ['age', 'trestbps', 'chol', 'thalach', 'oldpeak']:
72         Q1, Q3 = df_clean[col].quantile([0.25, 0.75])
73         IQR = Q3 - Q1
74         df_clean[col] = df_clean[col].clip(Q1 - 3*IQR, Q3 + 3*IQR)
75
76     # Feature engineering
77     df_clean['age_x_thalach'] = df_clean['age'] * df_clean['thalach']
78     df_clean['bmi_proxy'] = df_clean['chol'] / (df_clean['thalach'] + 1)
79
80     target = df_clean['target']
81     features = df_clean.drop('target', axis=1)
82
83     # Standardization
84     scaler = StandardScaler()
85     features_scaled = scaler.fit_transform(features)
86     features_df = pd.DataFrame(features_scaled, columns=features.columns)
87
88     df_final = features_df.copy()
89     df_final['target'] = target.values
90
91     df_final.to_csv(output_data.path, index=False)
92
93     print(f"[OK] Complete! {len(df_final)} records, {len(features.columns)} features")
94
95     Outputs = namedtuple('Outputs', ['num_records', 'num_features'])
96     return Outputs(len(df_final), len(features.columns))
97
98     # COMPONENT 2: Validate Data
99
100    @dsl.component(

```

```
101     base_image="python:3.9",
102     packages_to_install=["pandas", "numpy"]
103 )
104 def validate_data(
105     input_data: Input[Artifact],
106     min_records: int = 100,
107     min_features: int = 10
108 ) -> str:
109     """Validate data quality"""
110     import pandas as pd
111     import json
112
113     print("=" * 70)
114     print("COMPONENT 2: DATA VALIDATION")
115     print("=" * 70)
116
117     df = pd.read_csv(input_data.path)
118
119     validation_checks = {
120         'min_records': bool(len(df) >= min_records),
121         'min_features': bool((len(df.columns) - 1) >= min_features),
122         'no_missing': bool(df.isnull().sum().sum() == 0),
123         'balanced_target': bool(df['target'].value_counts().min() >= 30)
124     }
125
126     is_valid = all(validation_checks.values())
127
128     validation_report = json.dumps({
129         'total_records': int(len(df)),
130         'total_features': int(len(df.columns) - 1),
131         'checks': validation_checks,
132         'is_valid': bool(is_valid)
133     }, indent=2)
134
135     print(f"Total records: {len(df)}")
136     print(f"Total features: {len(df.columns) - 1}")
137     print(f"Result: {'[OK] VALID' if is_valid else '[ERROR] INVALID'}")
138
139     if not is_valid:
140         raise Exception(f"Validation failed: {validation_report}")
141
142     return validation_report
143
144 # COMPONENT 3: Train and Evaluate Model
145 @dsl.component(
146     base_image="python:3.9",
147     packages_to_install=["pandas", "scikit-learn", "joblib", "numpy"]
148 )
149 def train_model(
150     input_data: Input[Artifact],
151     model_output: Output[Artifact],
152     n_estimators: int = 100,
153     max_depth: int = 10,
154     min_samples_split: int = 5,
```

```
155     min_samples_leaf: int = 2
156 ) -> NamedTuple('Outputs', [
157     ('train_accuracy', float),
158     ('test_accuracy', float),
159     ('eval_accuracy', float),
160     ('eval_f1_score', float)
161 ]):
162     import pandas as pd
163     from sklearn.model_selection import train_test_split
164     from sklearn.ensemble import RandomForestClassifier
165     from sklearn.metrics import accuracy_score, f1_score
166     import joblib
167     from collections import namedtuple
168     import time
169
170     print("==" * 70)
171     print("COMPONENT 3: TRAINING AND EVALUATION")
172     print("==" * 70)
173
174     # Load data
175     df = pd.read_csv(input_data.path)
176     X = df.drop('target', axis=1)
177     y = df['target']
178
179     print(f"Dataset: {X.shape[0]} samples, {X.shape[1]} features")
180     print(f"Target distribution: {y.value_counts().to_dict()}")
181
182     # Split data
183     X_train, X_test, y_train, y_test = train_test_split(
184         X, y, test_size=0.2, random_state=42, stratify=y
185     )
186
187     print(f"\nTrain set: {X_train.shape[0]} samples")
188     print(f"Test set: {X_test.shape[0]} samples")
189
190     # Hyperparameters
191     print(f"\nHyperparameters:")
192     print(f" n_estimators: {n_estimators}")
193     print(f" max_depth: {max_depth}")
194     print(f" min_samples_split: {min_samples_split}")
195     print(f" min_samples_leaf: {min_samples_leaf}")
196
197     # Initialize model
198     model = RandomForestClassifier(
199         n_estimators=n_estimators,
200         max_depth=max_depth,
201         min_samples_split=min_samples_split,
202         min_samples_leaf=min_samples_leaf,
203         random_state=42,
204         n_jobs=-1,
205         verbose=1
206     )
207
208     # Train
```

```
209     print("\nTraining model...")
210     start_time = time.time()
211
212     model.fit(X_train, y_train)
213
214     elapsed_time = time.time() - start_time
215     print(f"[OK] Training completed in {elapsed_time:.2f} seconds")
216
217     # Predictions
218     train_pred = model.predict(X_train)
219     test_pred = model.predict(X_test)
220
221     # Training metrics
222     train_acc = accuracy_score(y_train, train_pred)
223     test_acc = accuracy_score(y_test, test_pred)
224
225     # Evaluation metrics (on test set)
226     eval_acc = accuracy_score(y_test, test_pred)
227     eval_f1 = f1_score(y_test, test_pred, average='weighted', zero_division=0)
228
229     print(f"\nTraining Results:")
230     print(f"  Train Accuracy: {train_acc:.4f}")
231     print(f"  Test Accuracy: {test_acc:.4f}")
232
233     print(f"\nEvaluation Results:")
234     print(f"  Accuracy: {eval_acc:.4f}")
235     print(f"  F1-Score: {eval_f1:.4f}")
236
237     # Feature importance
238     feature_importance = pd.DataFrame({
239         'feature': X.columns,
240         'importance': model.feature_importances_
241     }).sort_values('importance', ascending=False)
242
243     print(f"\nTop 5 Important Features:")
244     for idx, row in feature_importance.head(5).iterrows():
245         print(f"  {row['feature']}: {row['importance']:.4f}")
246
247     # Save model
248     joblib.dump(model, model_output.path)
249     print(f"\n[OK] Model saved to: {model_output.path}")
250
251     Outputs = namedtuple('Outputs', ['train_accuracy', 'test_accuracy', 'eval_accuracy',
252                                     , 'eval_f1_score'])
253     return Outputs(float(train_acc), float(test_acc), float(eval_acc), float(eval_f1))
254
255     # COMPONENT 4: Register Model to MinIO
256     @dsl.component(
257         base_image="python:3.9",
258         packages_to_install=["boto3", "joblib"]
259     )
260     def register_model(
261         model: Input[Artifact],
262         train_accuracy: float,
```

```
262     test_accuracy: float,
263     eval_accuracy: float,
264     eval_f1_score: float,
265     minio_endpoint: str,
266     minio_access_key: str,
267     minio_secret_key: str,
268     bucket_name: str,
269     n_estimators: int,
270     max_depth: int
271 ) -> str:
272     """Register trained model to MinIO with metadata"""
273     import boto3
274     from datetime import datetime
275     import json
276
277     print("=" * 70)
278     print("COMPONENT 5: MODEL REGISTRY")
279     print("=" * 70)
280
281     s3_client = boto3.client(
282         's3',
283         endpoint_url=minio_endpoint,
284         aws_access_key_id=minio_access_key,
285         aws_secret_access_key=minio_secret_key,
286         verify=False
287     )
288
289     # Generate version
290     timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
291     model_version = f"v_{timestamp}_acc_{eval_accuracy:.4f}".replace(".", "_")
292
293     print(f"Model version: {model_version}")
294
295     # Upload model
296     model_key = f"models/heart-disease/{model_version}/model.joblib"
297     with open(model.path, 'rb') as f:
298         s3_client.upload_fileobj(f, bucket_name, model_key)
299     print(f"[OK] Model uploaded: s3://{bucket_name}/{model_key}")
300
301     # Create metadata
302     metadata = {
303         'model_version': model_version,
304         'timestamp': timestamp,
305         'hyperparameters': {
306             'n_estimators': n_estimators,
307             'max_depth': max_depth
308         },
309         'metrics': {
310             'train_accuracy': float(train_accuracy),
311             'test_accuracy': float(test_accuracy),
312             'eval_accuracy': float(eval_accuracy),
313             'eval_f1_score': float(eval_f1_score)
314         },
315         'model_type': 'RandomForestClassifier',
```

```
316     'framework': 'scikit-learn',
317     'dataset': 'heart-disease'
318   }
319
320   # Upload metadata
321   metadata_key = f"models/heart-disease/{model_version}/metadata.json"
322   s3_client.put_object(
323     Bucket=bucket_name,
324     Key=metadata_key,
325     Body=json.dumps(metadata, indent=2).encode()
326   )
327   print(f"[OK] Metadata uploaded: s3://{bucket_name}/{metadata_key}")
328
329   model_uri = f"s3://{bucket_name}/{model_key}"
330
331   print(f"\n[OK] Model registration complete!")
332   print(f"Model URI: {model_uri}")
333   print(f"Test Accuracy: {eval_accuracy:.4f}")
334   print(f"F1-Score: {eval_f1_score:.4f}")
335
336   return model_uri
337
338 # PIPELINE
339
340 @dsl.pipeline(
341   name="heart-disease-simple-no-tuning",
342   description="Simple pipeline: Prepare -> Validate -> Train and Evaluate -> Register"
343 )
344 def heart_disease_pipeline(
345   # MinIO configs
346   minio_endpoint: str = "http://192.168.2.4:19000",
347   minio_access_key: str = "minioadmin",
348   minio_secret_key: str = "minioadmin",
349   bucket_name: str = "ml-data",
350   data_prefix: str = "raw/heart-disease",
351
352   # Model hyperparameters
353   n_estimators: int = 100,
354   max_depth: int = 10,
355   min_samples_split: int = 5,
356   min_samples_leaf: int = 2
357 ):
358   # STEP 1: DATA PREPARATION
359   prepare_task = prepare_data(
360     minio_endpoint=minio_endpoint,
361     minio_access_key=minio_access_key,
362     minio_secret_key=minio_secret_key,
363     bucket_name=bucket_name,
364     data_prefix=data_prefix
365   )
366   prepare_task.set_display_name('Data Preparation')
367
368   # STEP 2: DATA VALIDATION
```

```

369     validate_task = validate_data(
370         input_data=prepare_task.outputs['output_data']
371     )
372     validate_task.set_display_name('Data Validation')
373     validate_task.after(prepare_task)
374
375     # STEP 3: TRAINING AND EVALUATION
376     train_task = train_model(
377         input_data=prepare_task.outputs['output_data'],
378         n_estimators=n_estimators,
379         max_depth=max_depth,
380         min_samples_split=min_samples_split,
381         min_samples_leaf=min_samples_leaf
382     )
383     train_task.set_display_name('Training and Evaluation')
384     train_task.after(validate_task)
385
386     # STEP 4: MODEL REGISTRY
387     register_task = register_model(
388         model=train_task.outputs['model_output'],
389         train_accuracy=train_task.outputs['train_accuracy'],
390         test_accuracy=train_task.outputs['test_accuracy'],
391         eval_accuracy=train_task.outputs['eval_accuracy'],
392         eval_f1_score=train_task.outputs['eval_f1_score'],
393         minio_endpoint=minio_endpoint,
394         minio_access_key=minio_access_key,
395         minio_secret_key=minio_secret_key,
396         bucket_name=bucket_name,
397         n_estimators=n_estimators,
398         max_depth=max_depth
399     )
400     register_task.set_display_name('Model Registry')
401     register_task.after(train_task)
402 # COMPILE PIPELINE
403 if __name__ == "__main__":
404     output_file = "heart_disease_pipeline_simple.yaml"
405
406     print("=" * 70)
407     print("COMPILING SIMPLE PIPELINE (NO HYPERPARAMETER TUNING)")
408     print("=" * 70)
409
410     compiler.Compiler().compile(
411         pipeline_func=heart_disease_pipeline,
412         package_path=output_file
413     )
414
415     print(f"\n[OK] Pipeline compiled successfully!")
416     print(f"Output file: {output_file}")

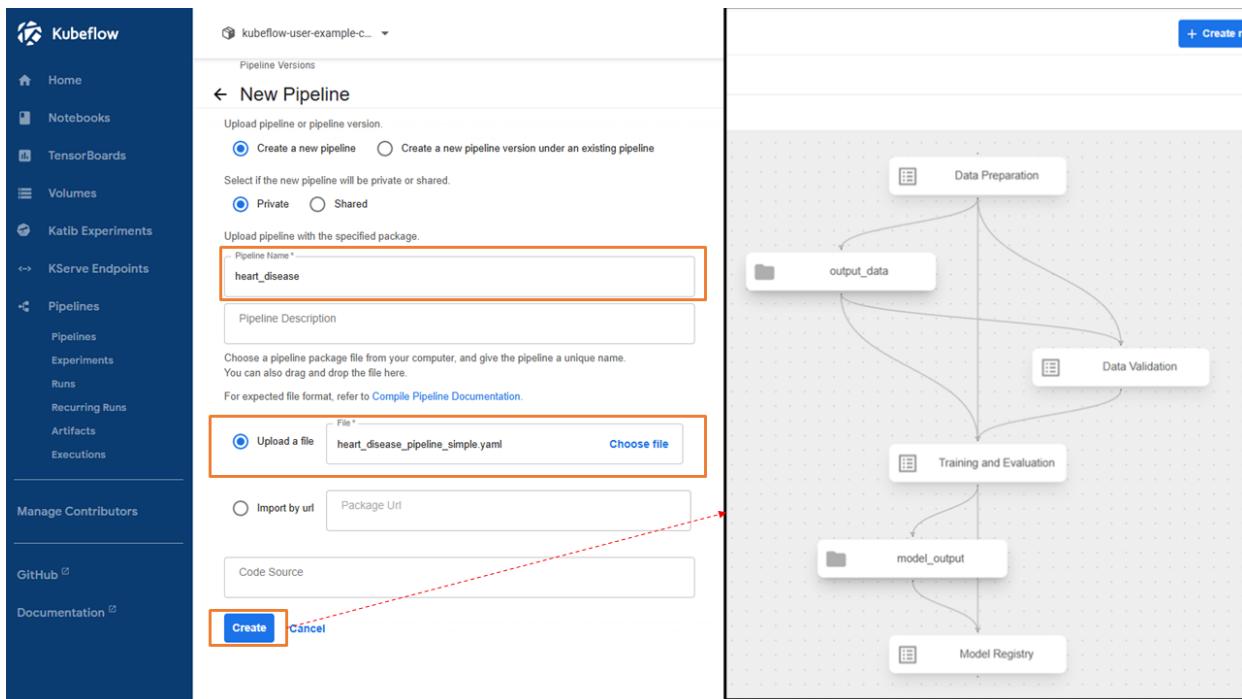
```

Pipeline gồm 4 bước tuần tự: Data Preparation → Data Validation → Training and Evaluation → Model Registry.

- **Component 1:** Prepare Data - Gộp 3 tasks (collect, preprocess, feature engineering).

Download Parquet files từ MinIO (limit 10 files đầu), concat thành single DataFrame. Preprocessing: fill missing values bằng median, drop duplicates, clip outliers dùng IQR method ( $3 \times \text{IQR}$  từ Q1/Q3). Feature engineering: tạo 2 features mới age\_x\_thalach (interaction term) và bmi\_proxy (cholesterol/heart rate ratio), standardize features bằng StandardScaler. Output CSV file và return metadata (num\_records, num\_features).

- **Component 2:** Validate Data - Quality checks trước khi training: verify minimum 100 records và 10 features, check no missing values (sau preprocessing phải = 0), verify balanced target (mỗi class ít nhất 30 samples tránh overfitting). Nếu fail bất kỳ check nào thì raise Exception stop pipeline. Output validation report dạng JSON.
- **Component 3:** Train and Evaluate Model - Thực hiện cả training và evaluation trong một component. Load data và split train/test (80/20 với stratify), huấn luyện mô Random Forest Classifier với hyperparameters cố định, tiếp theo sẽ tính toán training metrics (train accuracy, test accuracy). Sau đó evaluate model trên cùng test set: calculate eval\_accuracy (giống test\_accuracy) và eval\_f1\_score (weighted F1 cho multi-class). Lưu trained model và return 4 outputs: train\_accuracy, test\_accuracy, eval\_accuracy, eval\_f1\_score.
- **Component 4:** Register Model - Upload model.joblib và metadata.json vào MinIO. Metadata bao gồm model\_version (timestamp + accuracy), hyperparameters (n\_estimators, max\_depth), comprehensive metrics (train/test/eval accuracy, F1-score), model\_type, framework, dataset name. Trả về model S3 URI để phục vụ xác định mô hình triển khai với Kubeflow KServe.



Hình 31: Tạo pipeline trên Kubeflow Dashboard

Run details

Pipeline: heart\_disease

Pipeline Version: heart\_disease

Run name: Run of heart\_disease (68f06)

Description:

This run will be associated with the following experiment

Experiment: testing

This run will use the following Kubernetes service account.

Service Account:

Run Type

One-off  Recurring

Pipeline Root

Pipeline Root represents an artifact repository, refer to [Pipeline Root Documentation](#).

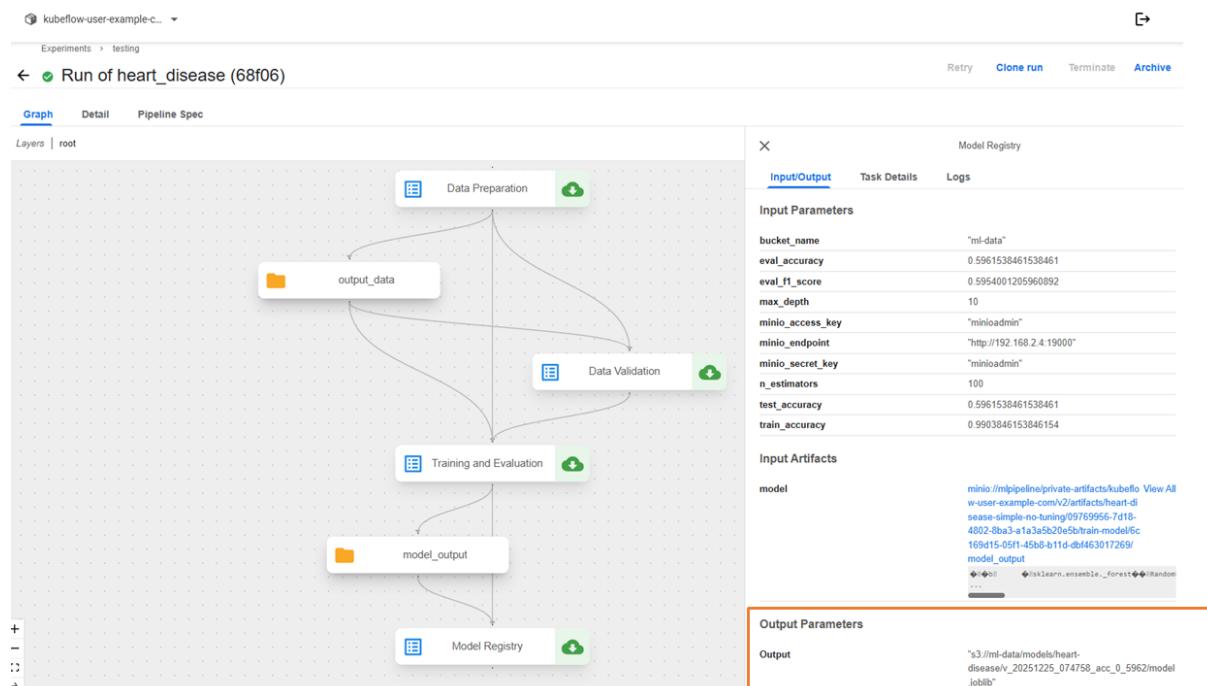
Custom Pipeline Root

Run parameters

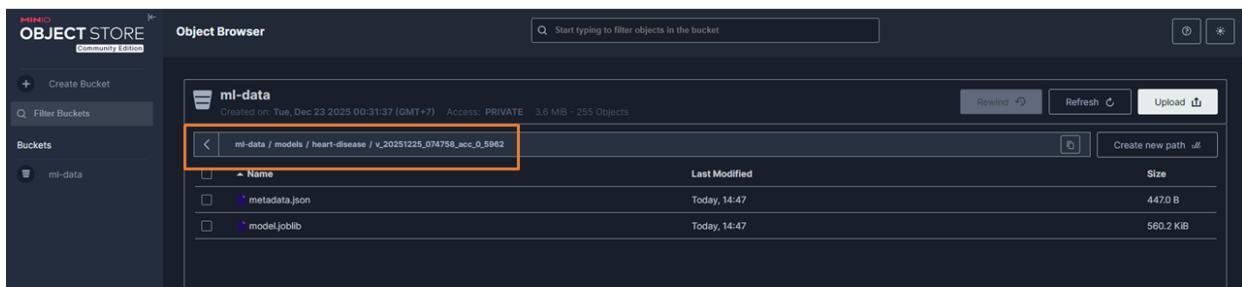
Specify parameters required by the pipeline

bucket_name - string	ml-data
data_prefix - string	rawheart-disease
max_depth - integer	10
min_samples_leaf - integer	2
min_samples_split - integer	5
minio_access_key - string	minioadmin
minio_endpoint - string	http://192.168.2.4:19000
minio_secret_key - string	minioadmin
n_estimators - integer	100

Hình 32: Tạo Run để tiến hành chạy pipeline



Hình 33: Kết quả sau khi chạy xong pipeline



Hình 34: MiniO lưu thông tin mô hình sau khi chạy xong pipeline

```
demo_kserve.py

1 from kubernetes import client, config
2 import boto3
3
4 def deploy_standalone(
5     model_uri: str = None,
6     service_name: str = "heart-disease-predictor",
7     namespace: str = "kubeflow-user-example-com",
8     minio_endpoint: str = "192.168.2.4:19000",
9     minio_access_key: str = "minioadmin",
10    minio_secret_key: str = "minioadmin",
11    bucket_name: str = "ml-data"
12):
13    # Load Kubernetes config
14    try:
15        config.load_incluster_config()
16    except:
17        config.load_kube_config()
18
19    # Find latest model if not specified
20    if model_uri is None:
21        print("\n[Finding latest model in MinIO]")
22        minio_endpoint_url = f"http://{{minio_endpoint}}" if not minio_endpoint.
23                                         startswith("http") else minio_endpoint
24
25        s3_client = boto3.client(
26            's3',
27            endpoint_url=minio_endpoint_url,
28            aws_access_key_id=minio_access_key,
29            aws_secret_access_key=minio_secret_key,
30            verify=False
31        )
32
33        try:
34            response = s3_client.list_objects_v2(
35                Bucket=bucket_name,
36                Prefix="models/heart-disease/"
37            )
38
39            if 'Contents' not in response:
40                print("[ERROR] No models found!")
```

```
40         return
41
42     model_paths = [obj['Key'] for obj in response['Contents'] if obj['Key'].endswith('/model.joblib')]
43
44     if not model_paths:
45         print("[ERROR] No model.joblib files found!")
46         return
47
48     # Sort by modified time (newest first)
49     model_paths_with_time = []
50     for path in model_paths:
51         obj_info = s3_client.head_object(Bucket=bucket_name, Key=path)
52         model_paths_with_time.append({
53             'path': path,
54             'modified': obj_info['LastModified']
55         })
56
57     model_paths_with_time.sort(key=lambda x: x['modified'], reverse=True)
58     latest_model = model_paths_with_time[0]
59     model_uri = f"s3://{bucket_name}/{latest_model['path']}"
60     print(f"[OK] Found latest model: {model_uri}")
61
62 except Exception as e:
63     print(f"[ERROR] Error finding model: {e}")
64     return
65
66 # Create MinIO secret
67 print("\n[1] Creating MinIO secret...")
68 secret_name = "minio-s3-secret"
69 secret = {
70     "apiVersion": "v1",
71     "kind": "Secret",
72     "metadata": {
73         "name": secret_name,
74         "namespace": namespace,
75         "annotations": {
76             "serving.kserve.io/s3-endpoint": minio_endpoint,
77             "serving.kserve.io/s3-usehttps": "0",
78             "serving.kserve.io/s3-region": "us-east-1",
79             "serving.kserve.io/s3-useanonicalcredential": "false"
80         }
81     },
82     "type": "Opaque",
83     "stringData": {
84         "AWS_ACCESS_KEY_ID": minio_access_key,
85         "AWS_SECRET_ACCESS_KEY": minio_secret_key
86     }
87 }
88
89 v1 = client.CoreV1Api()
90 try:
91     v1.create_namespaced_secret(namespace=namespace, body=secret)
92     print(f"[OK] Secret '{secret_name}' created")
```

```
93     except client.exceptions.ApiException as e:
94         if e.status == 409:
95             print(f"[OK] Secret '{secret_name}' already exists")
96         else:
97             raise
98
99 # Create ServiceAccount
100 print("\n[2] Creating ServiceAccount...")
101 sa_name = "kserve-sa"
102 service_account = {
103     "apiVersion": "v1",
104     "kind": "ServiceAccount",
105     "metadata": {
106         "name": sa_name,
107         "namespace": namespace
108     },
109     "secrets": [{"name": secret_name}]
110 }
111
112 try:
113     v1.create_namespaced_service_account(namespace=namespace, body=service_account)
114     print(f"[OK] ServiceAccount '{sa_name}' created")
115 except client.exceptions.ApiException as e:
116     if e.status == 409:
117         print(f"[OK] ServiceAccount '{sa_name}' already exists")
118     else:
119         raise
120
121 # Create InferenceService
122 print("\n[3] Creating InferenceService...")
123 inference_service = {
124     "apiVersion": "serving.kserve.io/v1beta1",
125     "kind": "InferenceService",
126     "metadata": {
127         "name": service_name,
128         "namespace": namespace
129     },
130     "spec": {
131         "predictor": {
132             "serviceAccountName": sa_name,
133             "model": {
134                 "modelFormat": {"name": "sklearn"},
135                 "storageUri": model_uri,
136                 "resources": {
137                     "requests": {"cpu": "100m", "memory": "256Mi"},
138                     "limits": {"cpu": "1", "memory": "1Gi"}
139                 }
140             }
141         }
142     }
143 }
144
145 api = client.CustomObjectsApi()
146 try:
```

```

147     api.create_namespaced_custom_object(
148         group="serving.kserve.io",
149         version="v1beta1",
150         namespace=namespace,
151         plural="inferenceservices",
152         body=inference_service
153     )
154     print(f"[OK] InferenceService '{service_name}' created")
155 except client.exceptions.ApiException as e:
156     if e.status == 409:
157         print(f"[WARN] InferenceService already exists - updating...")
158         api.patch_namespaced_custom_object(
159             group="serving.kserve.io",
160             version="v1beta1",
161             namespace=namespace,
162             plural="inferenceservices",
163             name=service_name,
164             body=inference_service
165         )
166         print(f"[OK] InferenceService updated")
167     else:
168         raise
169
170     print("\n" + "=" * 70)
171     print("[OK] DEPLOYMENT COMPLETE!")
172     print("=" * 70)
173     print(f"\nCheck status:")
174     print(f"  kubectl get inferenceservice {service_name} -n {namespace}")
175
176
177 if __name__ == "__main__":
178     import sys
179     if len(sys.argv) > 1 and sys.argv[1] == "deploy":
180         deploy_standalone()
181     else:
182         print("Usage:")
183         print("  python demo_kserve.py deploy")

```

Script demo\_kserve.py deploy trained model từ MinIO lên KServe để serving predictions. Auto-find Latest Model: Nếu không specify model\_uri, script tự động query MinIO list tất cả models trong models/heart-disease/, filter files kết thúc bằng /model.joblib, sort theo LastModified timestamp (newest first), và chọn model mới nhất để triển khai.

The screenshot shows the Kubeflow UI with the sidebar open. Under 'KServe Endpoints', the 'heart-disease-predictor' endpoint is selected. The 'OVERVIEW' tab is active. The 'URL internal' field contains the value `http://heart-disease-predictor.kubeflow-user-example-com.svc.cluster.local`, which is highlighted with an orange box.

Hình 35: Thông tin Endpoint của mô hình được triển khai

```
(base) jovyan@test-0:~$ curl -v http://heart-disease-predictor.kubeflow-user-example-com.svc.cluster.local/v1/models/heart-disease-predictor:predict \
-H "Content-Type: application/json" \
-d '{
  "instances": [
    [0.5, 1.0, 2.0, 0.3, -0.5, 0.0, 1.0, -0.8, 1.0, 1.2, 1.0, 0.0, 2.0, -0.4, 0.2],
    [-0.5, 0.0, 0.0, -0.3, -0.8, 0.0, 0.0, 0.8, 0.0, -0.5, 0.0, 0.0, 1.0, 0.3, -0.1],
    [0.0, 1.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.5, 1.0, 1.0, 2.0, 0.0, 0.0]
  ]
}'
```

Test with URL internal

Prediction

Hình 36: Kiểm tra dự đoán của mô hình với URL internal

## IV. Tài liệu tham khảo

- <https://kubernetes.io/vi/docs/concepts/overview/what-is-kubernetes/>
- <https://www.kubeflow.org/docs/started/>
- <https://kafka.apache.org/11/streams/architecture/>
- <https://spark.apache.org/docs/latest/streaming-programming-guide.html>
- <https://www.kaggle.com/datasets/ritwikb3/heart-disease-cleveland>

## V. Source code

- <https://github.com/ThuanNaN/aio2025-kafka-spark-kubeflow.git>