ĐẠI HỌC KINH TẾ TP. HỒ CHÍ MINH (UEH)

TRƯỜNG CÔNG NGHỆ VÀ THIẾT KẾ

KHOA CÔNG NGHỆ THÔNG TIN KINH DOANH

BÁO CÁO ĐỒ ÁN

TÍNH TOÁN HIỆU SUẤT CAO

(HPC – High Performance Computing)

Đề tài:

TỐI ƯU TRIỂN KHAI MÔ HÌNH PHÁT HIỆN ĐỐI TƯỢNG THỜI GIAN THỰC BẰNG NÉN MÔ HÌNH, DOCKER VÀ HPC

GVHD: TS.GVC Nguyễn Quốc Hùng

Nhóm thực hiện: Nhóm 1

Thái Hoài An (Nhóm trưởng)

Nguyễn Thị Thùy Dương

Nguyễn Duy Tân

Lê Vy

TP. Hồ Chí Minh, Tháng 5/2026

Mục lục

Danh sách hình vẽ

Danh sách bảng

DANH MỤC TỪ VIẾT TẮT

Từ viết tắt	Ý nghĩa
API	Application Programming Interface.
CI/CD	Continuous Integration / Continuous Delivery.
FPS	Frames Per Second.
HPC	High Performance Computing.
IoU	Intersection over Union.
KD	Knowledge Distillation.
mAP	mean Average Precision.
MLflow	Nền tảng theo dõi thí nghiệm và quản lý model.
NMS	Non-Maximum Suppression.
ONNX	Open Neural Network Exchange.
QPS	Queries Per Second.
TensorRT	Bộ tối ưu hóa và runtime suy luận của NVIDIA.
YOLO	You Only Look Once.

LỜI MỞ ĐẦU

Báo cáo này trình bày đồ án môn học Tính toán hiệu suất cao (HPC) với chủ đề tối ưu triển khai mô hình phát hiện đối tượng thời gian thực. Mục tiêu của đề tài là xây dựng một pipeline MLOps đầu cuối cho bài toán phát hiện đối tượng giao thông theo thời gian thực, trong đó mô hình YOLO được huấn luyện theo hướng teacher–student, có nhánh Knowledge Distillation (KD), được tối ưu suy luận bằng ONNX và TensorRT, đóng gói bằng Docker và tích hợp các thành phần giám sát vận hành.

Nội dung báo cáo được tổ chức theo hai trục bổ trợ nhau. Trục *lý thuyết* (Chương 2) trình bày cơ sở khoa học của các thành phần được sử dụng – kiến trúc YOLO, Knowledge Distillation cho object detection, các định dạng triển khai ONNX/TensorRT và hệ sinh thái công cụ MLOps – chỉ tập trung vào bản chất và nguyên lý hoạt động. Trục *thực nghiệm* (Chương 3 và 4) trình bày cách hệ thống được thiết kế, cấu hình và đo đạc, kèm toàn bộ số liệu thu được từ chuẩn benchmark thống nhất trên GPU Tesla T4.

Tóm tắt nội dung. Đề tài xây dựng một pipeline MLOps cho real-time traffic object detection, lấy YOLO26 (Ultralytics, n.d.) làm kiến trúc detector, kết hợp Knowledge Distillation đa thành phần cho object detection (Hinton, Vinyals, & Dean, 2015; Yang và cộng sự, 2022; Wang và cộng sự, 2019), và đặt trọng tâm vào kịch bản triển khai trên phần cứng hạn chế tài nguyên. Nhóm đã huấn luyện ba mô hình trên GPU Tesla T4 (Colab) với tập con 1000 ảnh giao thông, 30 epochs, và chạy chuẩn đo thống nhất (warmup 15, lặp 100 lần, đo trên 50 ảnh validation) bằng notebook `colab_benchmark.ipynb`. Về chất lượng (đo bằng `model.val()` chuẩn Ultralytics): teacher YOLO26x đạt mAP50 0.827 / mAP50–95 0.574; student YOLO26n baseline đạt 0.714 / 0.481; student YOLO26n + KD đạt 0.725 / 0.490 với Precision cải thiện rõ từ 0.691 lên 0.734 ($+4.4$ điểm), Recall gần như không đổi (0.678 $\rightarrow$ 0.679). Student nhỏ hơn teacher khoảng 23.5 lần về tham số (2.51M so với 58.82M) và 22 lần về dung lượng (5.14 MB so với 112.85 MB). Về tốc độ suy luận trên T4, student KD đạt 9.4 ms/ảnh ở PyTorch (107 FPS), giảm còn 5.1 ms ở ONNX Runtime (196 FPS) và còn 1.84 ms ở TensorRT FP16 (543 FPS) – nhanh hơn teacher PyTorch (66.9 ms) khoảng $36\times$. Provenance artifact đã được khép kín: `serving_model.pt` trùng checksum với `student_kd_best.pt` và khác teacher (xem Mục 4.2).

Từ khoá: object detection, YOLO, knowledge distillation, MLOps, FastAPI, Docker, Prometheus, drift detection, local deployment.

BẢNG PHÂN CÔNG CÁC THÀNH VIÊN

Bảng 2: Bảng phân công và mức độ hoàn thành công việc của các thành viên

TT	Họ và tên	Công việc phụ trách	Mức độ hoàn thành
1	Thái Hoài An (Nhóm trưởng)	Tổng quan đề tài, Chương 1, điều phối nhóm, tích hợp pipeline MLOps	100%
2	Nguyễn Thị Thùy Dương	Chương 2 – Cơ sở lý thuyết, tài liệu tham khảo, hiệu đính	100%
3	Nguyễn Duy Tân	Chương 3 – Triển khai hệ thống, Docker, cấu hình hạ tầng	100%
4	Lê Vy	Chương 4 – Thực nghiệm, benchmark, phân tích kết quả	100%

Chương 1: GIỚI THIỆU TÍNH TOÁN HIỆU SUẤT CAO (HPC) VÀ DỰ ÁN THỰC HIỆN

1.1. Tổng quan về tính toán hiệu suất cao và động lực đề tài

Tính toán hiệu suất cao (HPC) hướng tới việc khai thác tối đa tài nguyên tính toán – CPU đa nhân, GPU, bộ nhớ và hạ tầng phân tán – để giải các bài toán đòi hỏi thông lượng lớn hoặc độ trễ thấp. Trong lĩnh vực học sâu, hai biểu hiện điển hình của HPC là: (i) tăng tốc huấn luyện mô hình lớn bằng GPU/đa GPU, và (ii) tối ưu hóa suy luận (inference) để đáp ứng ràng buộc thời gian thực khi đưa mô hình vào vận hành. Đề tài này tập trung vào nhánh thứ hai: làm cho một mô hình phát hiện đối tượng vừa đủ chính xác, vừa đủ nhanh và đủ ổn định để triển khai trên hạ tầng giới hạn tài nguyên.

1.2. Bối cảnh bài toán

Phát hiện đối tượng thời gian thực là lớp bài toán nền tảng của thị giác máy tính trong các hệ thống camera giao thông, trợ lý lái xe, robot di động và giám sát an toàn. Đặc trưng của các ứng dụng này là yêu cầu đồng thời ba thuộc tính: độ chính xác đủ tốt, độ trễ suy luận thấp và khả năng vận hành ổn định trong môi trường triển khai thực tế. Vì vậy, trọng tâm của đề tài không nên dừng ở việc tối đa hóa điểm số huấn luyện, mà phải đặt lên bài toán tối ưu triển khai, giảm chi phí tài nguyên và giữ chất lượng dự đoán ở mức chấp nhận được.

Từ góc nhìn triển khai, các mô hình object detection lớn thường gặp ba rào cản. Thứ nhất, chi phí suy luận tăng theo kích thước mô hình, gây khó đạt ràng buộc real-time. Thứ hai, phụ thuộc framework huấn luyện khiến việc đưa mô hình sang môi trường sản phẩm gặp khó khăn về tính tương thích. Thứ ba, khi hệ thống đi vào vận hành, yêu cầu không chỉ là inference đúng mà còn là khả năng theo dõi drift, log, tài nguyên máy và vòng đời model.

1.3. Mục tiêu và đóng góp của đề tài

Mục tiêu tổng quát của đề tài là xây dựng một pipeline MLOps đầu cuối cho real-time traffic object detection, hướng đến các ứng dụng triển khai trên phần cứng hạn chế tài nguyên như edge device, server đơn lẻ hay hạ tầng phòng thí nghiệm. Trong phạm vi đó, nhóm tập trung vào ba đóng góp triển khai.

- **Đóng góp 1 – Knowledge Distillation đa thành phần cho YOLO26.** Nhóm cấu hình Knowledge Distillation đa thành phần, gồm *feature-based distillation* (theo cấu hình dự kiến trên các feature maps P3, P4, P5 tại các tầng neck) và *response-based distillation* trên classification scores cùng bounding box, thay vì chỉ dùng một công

thức KL divergence cuối cùng. Lợi thế kỹ thuật là student model học được cả phân phối nhãn lẫn cách teacher nội suy vị trí đối tượng, phù hợp với các kết quả KD dành riêng cho object detection (Yang và cộng sự, 2022; Wang và cộng sự, 2019).

- **Đóng góp 2 – Pipeline MLOps end-to-end đóng gói bằng Docker.** Hệ thống được tách thành bốn `docker-compose` stack: `serving_pipeline` (FastAPI + Gradio), `infra/mlflow` (MLflow + MinIO + MySQL), `infra/monitor` (Prometheus + Grafana + Loki + Alertmanager) và `infra/airflow` (orchestration). Lợi thế là từng cụm có thể bật/tắt độc lập theo cấu hình máy, đồng thời mỗi cụm vẫn giữ được quan hệ phụ thuộc với các cụm khác qua một docker network chung.

- **Đóng góp 3 – Khung triển khai tự chứa và thao tác hoá.** Hướng đến tính tái lập học thuật và kiểm soát toàn bộ tham số benchmark, nhóm dựng một khung triển khai *tự chứa* (self-contained) vận hành được trên một máy đơn lẻ với phần cứng phổ thông, kèm các script bật/tắt và healthcheck (`start_full_local.sh`, `stop_full_local.sh`, `smoke_test_local.sh`) để chuẩn hoá thao tác giữa các lần đo và giữa các thành viên trong nhóm.

Ba đóng góp này không nhằm đề xuất một thuật toán detection hoặc một kỹ thuật KD mới, mà nhằm chứng minh rằng một pipeline MLOps có thể được dựng và vận hành ổn định ngay cả trong điều kiện phần cứng hạn chế. Đây là một *implementation-oriented study*, phù hợp với phạm vi đánh giá của môn Tính toán hiệu suất cao.

1.4. Phạm vi triển khai và định hướng học thuật

Phạm vi đề tài tập trung vào kịch bản triển khai mô hình phát hiện đối tượng trên *phần cứng hạn chế*: một máy đơn lẻ với GPU phổ thông hoặc CPU, đại diện cho các tình huống thực tế như camera giao thông biên (edge), trạm xử lý cục bộ tại cơ sở giáo dục, hay máy chủ thí nghiệm có ngân sách giới hạn. Định hướng này phản ánh trực tiếp một nhánh quan trọng của Tính toán hiệu suất cao: tối ưu chi phí suy luận trên hạ tầng giới hạn thay vì giả định sẵn một cụm GPU lớn.

Việc dựng toàn bộ stack ở dạng *tự chứa* trên một máy có ba lợi ích học thuật. *Thứ nhất*, mọi thành phần (model server, tracking, monitoring, orchestration) đều được kiểm soát phiên bản và tham số nội bộ, tránh phụ thuộc các dịch vụ bên ngoài có thể đổi version âm thầm và làm số liệu benchmark thiếu nhất quán giữa các lần đo. *Thứ hai*, đo latency trên cùng một máy duy nhất là điều kiện tiên quyết để so sánh PyTorch, ONNX Runtime và TensorRT phản ánh đúng khác biệt định dạng thay vì khác biệt phần cứng (Mục 4.2). *Thứ ba*, một khung triển khai tự chứa dễ tái lập trên môi trường khác, phù hợp với yêu cầu kiểm tra độc lập trong nghiên cứu.

Phạm vi đo của đề tài bao phủ đầy đủ vòng đời pipeline. Phần huấn luyện (teacher, student baseline, student KD) được thực hiện trên GPU Tesla T4. Phần inference được benchmark trên cả CPU và GPU, cho ba định dạng triển khai PyTorch, ONNX Runtime và TensorRT FP16

(Bảng 4.7). Các thành phần serving, tracking, monitoring, orchestration và drift detection được hiện thực hoá đầy đủ và vận hành kiểm thử nội bộ (Chương 3). Hai hướng mở rộng còn để dành cho giai đoạn tiếp theo – huấn luyện trên toàn bộ dataset với nhiều seed, và validate lại mAP sau khi export ONNX/TensorRT để đo trượt do lượng tử FP16 – được trình bày như một plan cụ thể ở Mục *Hướng phát triển* của Kết luận.

1.5. Bài toán dữ liệu và giả định đầu vào

Đề tài sử dụng tập *Traffic Detection Project* công bố trên Kaggle (Sardoan, n.d.) làm dữ liệu thực nghiệm. Tập dữ liệu chứa ảnh đường phố có gán nhãn theo định dạng YOLO cho năm lớp đối tượng giao thông cơ bản: `bicycle`, `bus`, `car`, `motorbike` và `person`. Định dạng nhãn YOLO biểu diễn mỗi đối tượng bằng năm số `class_id x_center y_center width height` với toạ độ tâm cùng chiều rộng và chiều cao đã chuẩn hoá theo kích thước ảnh; đây là chuẩn được sử dụng xuyên suốt pipeline huấn luyện, serving và benchmark trong đề tài.

1.6. Cấu trúc báo cáo

Phần còn lại của báo cáo được tổ chức như sau. Chương 2 trình bày cơ sở lý thuyết về object detection, kiến trúc YOLO, Knowledge Distillation cho detection, định dạng triển khai ONNX/TensorRT, hệ sinh thái MLOps và một số công trình liên quan gần đây. Chương 3 mô tả quá trình triển khai thử nghiệm hệ thống, gồm cấu trúc mã nguồn, kiến trúc tổng thể, data pipeline, training pipeline (teacher–student–KD), serving pipeline, hạ tầng Docker, cấu hình các thành phần MLOps và ứng dụng demo. Chương 4 trình bày kết quả thực nghiệm theo trình tự: cấu hình thực nghiệm và protocol đo, kết quả chất lượng mô hình, kết quả benchmark tốc độ suy luận, phân tích trade-off của Knowledge Distillation và thảo luận tổng hợp. Phần Kết luận và hướng phát triển tổng kết những gì đã đạt được cùng plan mở rộng tiếp theo. Phụ lục cuối báo cáo lưu kiểm tra provenance bằng checksum SHA1 và đường dẫn tới kho mã nguồn cùng các artifact mở.

Chương 2: CƠ SỞ LÝ THUYẾT VỀ CHỦ ĐỀ NGHIÊN CỨU

2.1. Object detection và vì sao chọn YOLO

Object detection là bài toán đồng thời định vị và phân loại các đối tượng trong ảnh. So với image classification, đầu ra của object detection không chỉ là nhãn lớp mà còn là bounding box. Có thể chia các hướng tiếp cận phổ biến thành ba nhóm lớn: one-stage detector, two-stage detector và end-to-end detector. Trong số này, YOLO – được giới thiệu lần đầu bởi Redmon và cộng sự (Redmon và cộng sự, 2016) – nổi bật vì gộp định vị và phân loại vào một lần forward duy nhất, cân bằng tốt giữa tốc độ và độ chính xác và do đó phù hợp với bài toán real-time.

Trong pipeline của nhóm, YOLO được chọn vì ba lý do. Thứ nhất, hệ sinh thái Ultralytics cung cấp workflow tương đối liền mạch từ train đến export và deploy. Thứ hai, các biến thể kích thước khác nhau của YOLO thuận tiện cho thiết lập teacher–student. Thứ ba, việc export sang ONNX hoặc TensorRT đã được hỗ trợ ở mức tài liệu và công cụ, giúp khớp với mục tiêu tối ưu triển khai.

2.2. Kiến trúc YOLO: backbone, neck và detection head

Các mô hình họ YOLO chia sẻ một sườn kiến trúc thống nhất gồm ba khối nối tiếp: **backbone** trích đặc trưng từ ảnh, **neck** hợp nhất đặc trưng đa tỉ lệ, và **detection head** dự đoán bounding box cùng lớp đối tượng. Cách tổ chức này là nền tảng để hiểu vì sao distillation cần đụng tới nhiều thành phần khác nhau của mạng (Mục 2.4) thay vì chỉ một đầu ra logits cuối.

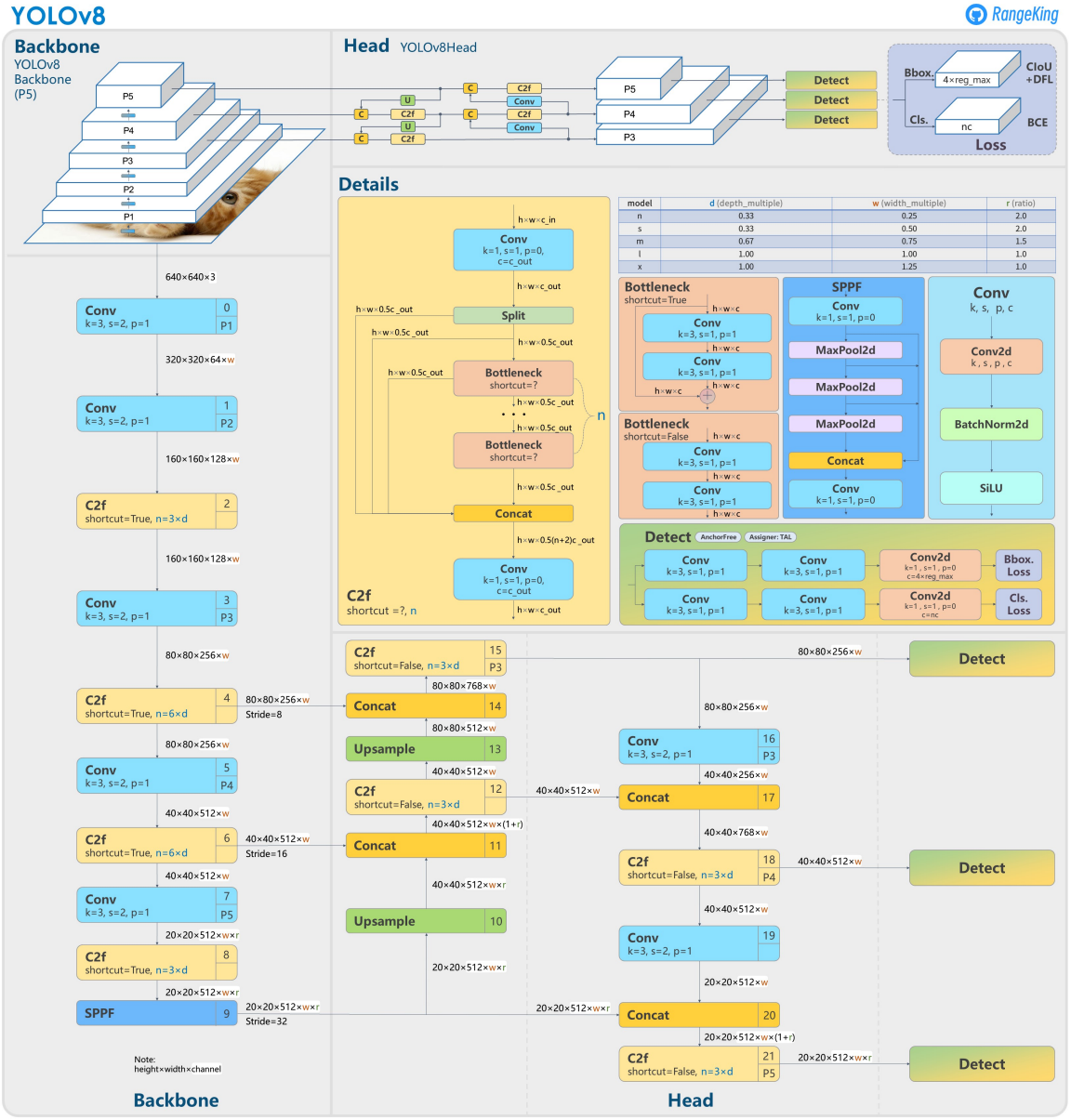
Backbone là một mạng tích chập sâu lấy ảnh đầu vào và sinh ra một chuỗi feature map có độ phân giải giảm dần và độ trừu tượng tăng dần. Các phiên bản gần đây của Ultralytics (YOLOv8 trở đi) sử dụng các khối kiểu *Cross Stage Partial* (CSP) (Wang và cộng sự, 2020) và biến thể *C2f* (Cross-stage partial with 2 convolutions, fast); ý tưởng cốt lõi của CSP là chia feature map thành hai nhánh – một nhánh đi qua chuỗi block tích chập, một nhánh nối tắt (shortcut) – nhằm tăng dòng gradient và giảm chi phí tính toán so với DarkNet truyền thống. Backbone thường xuất ra ba feature map ở các tỉ lệ giảm 8, 16 và 32 lần so với ảnh đầu vào, đóng vai trò đầu vào cho neck. Mỗi biến thể kích thước của một họ YOLO (nano, small, medium, large, x-large) chia sẻ cấu trúc backbone như nhau nhưng khác nhau ở độ rộng kênh và độ sâu, tạo ra dải mô hình từ vài triệu tới vài chục triệu tham số phù hợp với nhiều ràng buộc triển khai.

Neck đảm nhiệm việc *xếp chồng* thông tin theo cả hai chiều: từ tầng nông giàu chi tiết không gian xuống tầng sâu giàu ngữ nghĩa, và ngược lại. Cấu trúc phổ biến là Path Aggregation Network (PAN) (Liu và cộng sự, 2018) đặt trên Feature Pyramid Network (FPN) (Lin và cộng sự, 2017), tạo ra ba feature map đầu ra ký hiệu là P3, P4, P5 tương ứng các tỉ lệ giảm 8, 16, 32 lần. Với ảnh đầu vào 640×640, ba feature map này có kích thước lần lượt 80×80, 40×40, 20×20. Ý nghĩa thực tiễn: P3 mạnh cho đối tượng nhỏ (xe đạp, người ở xa), P5 mạnh cho

đối tượng lớn (xe buýt gần camera); muốn distillation *dạy được* student trên mọi kích thước đối tượng, feature-based loss phải chạm vào cả ba tầng này (xem Mục 2.4.2).

Detection head dự đoán trên từng *prediction point* của ba feature map. Tổng số điểm dự đoán ở imgsz 640 là $80^2 + 40^2 + 20^2 = 8400$. Tại mỗi điểm, head sinh ra một vector phân loại (kích thước bằng số lớp, ở đây là 5) và một vector hồi quy bounding box. Các phiên bản YOLOv8 trở đi của Ultralytics dùng *anchor-free decoupled head*: *anchor-free* nghĩa là không dùng các hộp ứng cử có kích thước cố định cài đặt trước như YOLOv5, mà mỗi prediction point trực tiếp hồi quy ra toạ độ box; *decoupled* nghĩa là hai nhánh phân loại và hồi quy có lớp tích chập riêng, không chia sẻ tham số. Box được hồi quy theo *Distribution Focal Loss* (DFL) (Li và cộng sự, 2020): thay vì xuất ra một số thực cho mỗi cạnh box, mô hình xuất ra một phân phối xác suất rời rạc trên dải giá trị có thể của cạnh đó rồi tính kỳ vọng – cách tiếp cận này cho phép biểu diễn độ bất định khi mép box không sắc nét và đã chứng minh ổn định hơn hồi quy điểm trực tiếp. Hệ quả với KD: *response* mà student cần học không chỉ là logits phân loại mà còn là phân phối box; chính cấu hình của đề tài cũng tách thành ba loss con cho dense logits, sparse logits và box (Mục 2.4.4).

Khi áp dụng KD trên cặp teacher–student cùng họ YOLO, sườn backbone–neck–head chung khiến việc ghép cặp feature map tương ứng theo P3/P4/P5 là trực tiếp; phép biến đổi 1×1 giữa số kênh của teacher và student (Mục 2.4.2) là lớp *cầu nối* kỹ thuật để xử lý khác biệt độ rộng kênh.



Hình 2.1: Kiến trúc YOLOv8: backbone (CSP/C2f), neck PAN-FPN và detection head anchor-free trên ba feature map P3/P4/P5. Nguồn: kho cấu hình `mmyolo` của OpenMMLab (OpenMMLab, n.d.).

2.3. Biểu diễn dữ liệu và các khái niệm detection cơ bản

2.3.1. YOLO-format và bài toán localization

Một bounding box có thể được biểu diễn theo *corner format* (x_1, y_1, x_2, y_2) hoặc *center format* (x, y, w, h). Trong thực tế của đề tài này, pipeline dùng YOLO-format, tức tọa độ tâm cùng chiều rộng và chiều cao đã được chuẩn hóa theo kích thước ảnh. Chuẩn này thuận lợi cho việc huấn luyện bằng Ultralytics và nhất quán với cách nhiều detector một-giai-đoạn xử lý nhãn.

Hình 2.2: Hệ tọa độ YOLO-format: bounding box biểu diễn bằng $(x_{center}, y_{center}, w, h)$ đã chuẩn hoá theo kích thước ảnh. Nguồn: tài liệu định dạng nhãn YOLO của Roboflow (Roboflow, n.d.).

Một hệ quả quan trọng là mọi bước augmentation hình học đều phải cập nhật đồng thời cả ảnh và bounding box. Vì vậy, các thư viện như Albumentations thường phù hợp hơn cách biến đổi ảnh đơn thuần rồi giữ nguyên nhãn cũ.

2.3.2. IoU và NMS

IoU đo mức chồng lấp giữa bounding box dự đoán và bounding box ground truth:

$$\text{IoU} = \frac{\text{Area}(B_p \cap B_g)}{\text{Area}(B_p \cup B_g)}.$$

IoU càng cao thì dự đoán càng sát mục tiêu. Trong các detector truyền thống thuộc họ YOLO, NMS thường được dùng để loại bớt các hộp dự đoán chồng lấp mạnh. Tuy nhiên, chính bước hậu xử lý này cũng làm tăng độ trễ và khiến pipeline kém *end-to-end* hơn. Đây cũng là một trong các động lực dẫn tới các hướng detector mới như YOLOv10, nơi bài toán hiệu năng được xét cùng với việc giảm phụ thuộc vào NMS (Wang và cộng sự, 2024).

2.3.3. Precision, Recall và mAP

Để đánh giá chất lượng một bộ phát hiện đối tượng, từng dự đoán của mô hình được đối chiếu với ground truth ở một ngưỡng IoU cố định. Một dự đoán được tính là *true positive* (TP) nếu vừa thuộc đúng lớp vừa có IoU với một ground truth khớp vượt ngưỡng đặt trước; trái lại là *false positive* (FP). Một ground truth không có dự đoán nào khớp được tính là *false negative* (FN). Từ ba lượng đó, hai chỉ số cơ bản được định nghĩa:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}, \qquad \text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}.$$

Precision cao đồng nghĩa số dự đoán sai (false positive) ít; Recall cao đồng nghĩa ít đối tượng bị bỏ sót.

Bằng cách quét ngưỡng confidence của detector, ta thu được một đường cong Precision–Recall cho mỗi lớp. Diện tích dưới đường cong này, ở một ngưỡng IoU cố định, gọi là *Average Precision* (AP) của lớp đó. *Mean Average Precision* (mAP) là trung bình AP qua tất cả C lớp:

$$\text{mAP} = \frac{1}{C} \sum_{c=1}^{C} \text{AP}_c.$$

Trong báo cáo, hai biến thể được sử dụng: **mAP50** đo ở ngưỡng IoU $= 0{,}50$ (định vị tương đối lỏng), và **mAP50–95** theo chuẩn COCO (Lin và cộng sự, 2014) là trung bình mAP đo ở mười ngưỡng IoU từ $0{,}50$ đến $0{,}95$ cách đều $0{,}05$ – chỉ số này khắt khe hơn vì đòi hỏi định vị chính xác ở mọi mức IoU.

Bên cạnh chất lượng phát hiện, ba chỉ số định lượng chi phí triển khai được dùng xuyên suốt báo cáo:

- **Latency**: thời gian xử lý một yêu cầu (ms/ảnh). *Inference latency* chỉ tính bước forward mô hình; *end-to-end latency* cộng cả tiền xử lý và hậu xử lý. Mục 4.4 báo cáo cả mean và p95 latency.
- **FPS** (Frames Per Second): nghịch đảo latency, phản ánh khả năng phục vụ liên tục.
- **Kích thước mô hình và bộ nhớ**: dung lượng file checkpoint, mức tiêu thụ RAM/VRAM khi nạp; ràng buộc trực tiếp khả năng triển khai trên thiết bị giới hạn tài nguyên.

2.4. Knowledge Distillation cho object detection

Mục này trình bày Knowledge Distillation (KD) ở hai lớp. Lớp một giới thiệu *paradigm* teacher–student và các công cụ toán học cốt lõi (logits, softmax có nhiệt độ, KL divergence) cần để hiểu mọi biến thể KD. Lớp hai mở rộng các công cụ đó cho object detection, nơi mô hình phải đồng thời học phân loại và hồi quy bounding box.

2.4.1. Paradigm Teacher–Student và công cụ toán học

Knowledge Distillation lấy hai mạng neural có dung lượng khác nhau làm trung tâm. Ý tưởng *nén tri thức* bằng cách huấn luyện một mạng nhỏ bắt chước một mạng lớn được Bucila và cộng sự đề xuất từ năm 2006 (Bucila và cộng sự, 2006) và sau đó được Hinton, Vinyals và Dean tổng quát hoá thành cơ chế *soft label* cho mạng neural sâu (Hinton, Vinyals, & Dean, 2015). **Teacher model** – ký hiệu f_T – là một mạng đã được huấn luyện trước với dung lượng lớn và đạt chất lượng cao trên tập dữ liệu mục tiêu, đóng vai trò *nguồn tri thức*. **Student model** – ký hiệu f_S – là một mạng nhỏ hơn về số tham số và chi phí tính toán, được huấn luyện để *bắt chước hành*

vi của teacher. Mục tiêu là f_S kế thừa phần lớn năng lực dự đoán của f_T trong khi đạt chi phí suy luận đủ thấp cho triển khai thời gian thực; teacher chỉ được dùng ở giai đoạn huấn luyện và không xuất hiện ở giai đoạn phục vụ.

Tại đầu ra phân loại của một mạng cho bài toán C lớp, vector số thực $\mathbf{z} \in \mathbb{R}^C$ trước khi qua hàm chuẩn hoá được gọi là *logits*. *Classification scores* (xác suất lớp) thu được bằng cách áp hàm softmax có nhiệt độ τ:

$$p_i(\mathbf{z}, \tau) = \frac{\exp(z_i/\tau)}{\sum_{j=1}^{C} \exp(z_j/\tau)}, \quad i = 1, \ldots, C.$$

Khi $\tau = 1$, công thức trở về softmax chuẩn. Khi $\tau > 1$, phân phối *mềm* hơn (các xác suất phẳng hơn), làm lộ rõ thông tin về độ tương đồng giữa các lớp gần nhau – đây chính là *soft label* mà student được khuyến khích bắt chước, bên cạnh nhãn cứng (*hard label*) thông thường.

Khoảng cách giữa hai phân phối xác suất rời rạc P và Q trên cùng tập rời rạc được đo bằng **Kullback–Leibler divergence**:

$$D_{\mathrm{KL}}(P \,\|\, Q) = \sum_i P_i \, \log \frac{P_i}{Q_i}.$$

$D_{\mathrm{KL}}(P\|Q) \geq 0$, bằng 0 khi và chỉ khi $P = Q$, và *không đối xứng* ($D_{\mathrm{KL}}(P\|Q) \neq D_{\mathrm{KL}}(Q\|P)$ trong trường hợp tổng quát). Trong KD, P thường là phân phối softmax của teacher (mục tiêu mềm) và Q là phân phối softmax của student (đang học); tối thiểu hoá $D_{\mathrm{KL}}(P\|Q)$ buộc student tiệm cận teacher ở *mọi* lớp, kể cả các lớp có xác suất nhỏ.

Hàm loss distillation cổ điển của Hinton et al. (Hinton, Vinyals, & Dean, 2015) kết hợp nhãn cứng và tri thức teacher:

$$\mathcal{L}_{\mathrm{KD}} = (1 - \lambda)\, \mathcal{L}_{\mathrm{CE}}(p_S(1), y) \;+\; \lambda \cdot \tau^2 \cdot D_{\mathrm{KL}}\big(p_T(\tau) \,\|\, p_S(\tau)\big),$$

trong đó $\mathcal{L}_{\mathrm{CE}}$ là cross-entropy giữa dự đoán student và nhãn cứng y; $p_T(\tau), p_S(\tau)$ là softmax có nhiệt độ của teacher và student; $\lambda \in [0, 1]$ điều chỉnh tỷ trọng hai thành phần; và hệ số τ^2 giữ thang gradient ổn định khi τ lớn. Công thức trên là điểm xuất phát; các mục kế tiếp mở rộng nó cho bài toán detection vốn đòi hỏi nhiều thành phần loss hơn vì mô hình phải học đồng thời *phân loại* và *định vị*.

2.4.2. Vì sao detection cần multi-component distillation

Knowledge Distillation là kỹ thuật truyền tri thức từ một teacher model lớn sang một student model nhỏ hơn nhằm giữ chất lượng dự đoán khi giảm chi phí suy luận (Hinton, Vinyals, & Dean, 2015). Trong bài toán classification, công thức KL divergence trên logits cuối cùng thường đủ để student tiệm cận teacher. Đối với object detection, ràng buộc khắt khe hơn vì mô hình phải đồng thời học hai nhiệm vụ: phân loại đối tượng và hồi quy bounding box. Nếu chỉ distill nhánh classification mà bỏ qua box, student có thể *đoán đúng lớp nhưng đặt sai vị trí*; điều này hầu

như không xuất hiện trong classification thuần. Khái niệm KD chuyên biệt cho detector được Chen và cộng sự đặt nền sớm (Chen và cộng sự, 2017), sau đó được mở rộng thành các sơ đồ multi-component kết hợp đặc trưng trung gian, logits và box regression (Wang và cộng sự, 2019; Yang và cộng sự, 2022; Task Integration Distillation, 2024).

2.4.3. Feature-based distillation tại các tầng P3/P4/P5

Theo cấu hình dự kiến, feature-based distillation được áp dụng trên các feature map ở phần neck của YOLO26, nơi thông tin đa tỉ lệ đã được tích hợp (Ultralytics, n.d.). Ý tưởng dùng feature map trung gian làm *hint* cho student được Romero và cộng sự đề xuất trong FitNets (Romero và cộng sự, 2015), và đã trở thành thành phần chuẩn của các sơ đồ KD hiện đại. Tuân theo kiến trúc YOLO chuẩn, các feature map tương ứng ba tỉ lệ P3 (80×80), P4 (40×40) và P5 (20×20) được dùng để distill, vì đây là các tầng mang biểu diễn ngữ nghĩa giàu thông tin nhất cho bài toán detection.

Khó khăn kỹ thuật ở đây là teacher và student thường có số lượng kênh (channel) khác nhau, do student được thiết kế nhỏ hơn. Để hai feature map khớp nhau về số chiều biểu diễn, một phép *channel-wise transformation* ϕ_l được dùng dưới dạng MLP hai tầng với các tích chập 1×1, chiếu feature map student lên cùng số kênh với teacher tại từng tỉ lệ $l \in \{P3, P4, P5\}$. Sau khi đã chiếu, khoảng cách giữa hai bản đồ được đo bằng khoảng cách bình phương L_2:

$$\mathcal{L}_{\text{feat}} = \sum_{l \in \{P3,P4,P5\}} \left\| F_T^l \; - \; \phi_l\big(F_S^l\big) \right\|_2^2,$$

trong đó F_T^l và F_S^l lần lượt là feature map của teacher và student ở tỉ lệ l. Ý nghĩa kỹ thuật là student không bị ép sao chép số kênh của teacher, nhưng vẫn được dạy biểu diễn ngữ nghĩa tương đương ở từng tỉ lệ; điều này đặc biệt hữu ích cho xử lý đối tượng nhỏ hoặc ở xa, vốn phụ thuộc nhiều vào feature map tỉ lệ cao P3.

2.4.4. Response-based distillation trên classification và bounding box

Ở đầu ra cuối cùng, response-based distillation chạm vào hai thành phần. *Đối với classification*, gọi $\mathbf{z}_T^{(n)}, \mathbf{z}_S^{(n)} \in \mathbb{R}^C$ là logits của teacher và student tại prediction point thứ n trên ba feature map (tổng cộng $N = 80^2 + 40^2 + 20^2 = 8400$ điểm ở imgsz 640). Loss classification ép phân phối student tiệm cận phân phối teacher qua KL divergence với nhiệt độ τ:

$$\mathcal{L}_{\text{cls}} \; = \; \frac{1}{N} \sum_{n=1}^{N} D_{\text{KL}}\Big(p_T^{(n)}(\tau) \,\Big\|\, p_S^{(n)}(\tau) \Big),$$

trong đó $p_T^{(n)}(\tau), p_S^{(n)}(\tau)$ là softmax có nhiệt độ của hai vector logits. Nhờ vế phải lấy KL ở mọi prediction point, student học được cả *soft label* – mức độ tự tin của teacher giữa các lớp gần nhau và cấu trúc lỗi thường gặp – không chỉ là nhãn cứng tại các điểm có đối tượng.

Đối với bounding box regression, gọi $b_T^{(n)}, b_S^{(n)}$ là box dự đoán của teacher và student tại điểm n, $\text{obj}_T^{(n)} \in [0,1]$ là confidence (objectness) của teacher, và θ là ngưỡng `box_objectness_threshold`. Loss box-regression chỉ lấy các điểm có teacher đủ tin cậy và trọng số hoá theo confidence:

$$\mathcal{L}_{\text{box}} = \sum_{n\,:\,\text{obj}_T^{(n)} > \theta} \text{obj}_T^{(n)} \cdot d\big(b_T^{(n)}, b_S^{(n)}\big),$$

trong đó $d(\cdot,\cdot)$ có thể là khoảng cách Smooth-L_1 hoặc $1 - \text{GIoU}$ (Generalized IoU). Lợi thế của ngưỡng và trọng số là tránh nhiễu từ các dự đoán teacher kém chất lượng – vốn xuất hiện nhiều ở các vùng không có đối tượng thật – và là lý do cốt lõi vì sao response-based distillation cho detection hiệu quả hơn cách lấy mọi prediction của teacher làm pseudo-label.

2.4.5. Hàm loss tổng hợp và bốn câu hỏi đánh giá KD

Tổng hợp hai hướng trên, hàm loss tổng thể của quá trình distillation được viết dưới dạng:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{task}} + \alpha_{\text{feat}}\,\mathcal{L}_{\text{feat}} + \alpha_{\text{resp}}\,\mathcal{L}_{\text{resp}},$$

trong đó $\mathcal{L}_{\text{task}}$ là loss huấn luyện gốc của YOLO (classification loss và box regression loss trên nhãn gốc), $\mathcal{L}_{\text{feat}}$ và $\mathcal{L}_{\text{resp}}$ là hai thành phần distillation đã mô tả, còn α_{feat} và α_{resp} là hệ số điều chỉnh mức độ ảnh hưởng của distillation so với việc học trực tiếp từ dữ liệu gán nhãn. Trong thực tế triển khai, α_{resp} thường được tách thành nhiều trọng số con cho từng nhánh đầu ra (ví dụ: logits dày tính trên toàn bộ prediction points, logits thưa chỉ lấy điểm có confidence cao của teacher, và loss box) cùng một tham số nhiệt độ điều khiển độ mềm của phân phối nhãn.

Một quy trình đánh giá KD hợp lý cho đề tài này phải trả lời được bốn câu hỏi riêng biệt, tách rõ chi phí và lợi ích:

1. Student baseline nhỏ hơn teacher bao nhiêu về kích thước file và chi phí suy luận?
2. Student baseline mất bao nhiêu mAP so với teacher?
3. Student KD phục hồi được bao nhiêu phần mAP đã mất so với student baseline?
4. Phần mAP phục hồi đó có đi kèm chi phí latency hoặc bộ nhớ tăng đáng kể không?

Nếu không tách bốn câu hỏi này, rất dễ rơi vào ngộ nhận rằng *có KD là tốt hơn*, trong khi trên thực tế một cấu hình KD chưa tối ưu vẫn có thể tăng thời gian train nhưng không mang lại cải thiện triển khai rõ rệt. Chương 4 sẽ quay lại bốn câu hỏi này dưới dạng bảng kết quả thực tế.

Ghi chú về bằng chứng: cấu hình KD theo hướng multi-component đã xuất hiện rõ trong `notebooks/trainer.py` và `train.yaml`, nhưng *việc khẳng định trainer.py triển khai chính xác phép channel-wise MLP* 1×1 *tại đúng ba tầng feature map P3/P4/P5* cần được audit

bổ sung bằng cách đối chiếu với các công trình tham chiếu (Wang và cộng sự, 2019; Yang và cộng sự, 2022).

2.5. ONNX và TensorRT trong triển khai

ONNX (Open Neural Network Exchange) đóng vai trò chuẩn trung gian giúp tách mô hình khỏi framework huấn luyện: một mô hình huấn luyện bằng PyTorch (Paszke và cộng sự, 2019) hoặc TensorFlow có thể được xuất sang một đồ thị tính toán ONNX rồi chạy bằng *ONNX Runtime* – một thư viện inference đa nền tảng (CPU, CUDA, DirectML, CoreML, mobile NNAPI). Lợi thế chính của chuẩn này là giảm phụ thuộc framework gốc khi đưa mô hình sang môi trường sản phẩm, đồng thời cho phép áp dụng các tối ưu cấp đồ thị (constant folding, operator fusion) độc lập với framework đào tạo.

TensorRT là bộ tối ưu hoá và runtime inference của NVIDIA dành cho GPU. Sau khi có mô hình ONNX (hoặc đồ thị TensorFlow/PyTorch trực tiếp), TensorRT thực hiện ba lớp tối ưu sâu: (i) hợp nhất kernel (kernel fusion) để giảm chi phí khởi động kernel và đọc bộ nhớ trung gian, (ii) lượng tử hoá hỗn hợp xuống FP16 hoặc INT8 để tăng thông lượng tensor cores, và (iii) lựa chọn engine cụ thể tối ưu cho phần cứng đích (Ampere, Hopper...) thông qua một bước *build* ngoại tuyến. Kết quả thường là một file `.engine` chạy nhanh hơn đáng kể so với ONNX Runtime trên cùng GPU, đổi lại engine chỉ tương thích với đúng phiên bản GPU/CUDA dùng để build.

2.6. Hệ sinh thái MLOps và công cụ giám sát

Một hệ sinh thái MLOps điển hình được tổ chức theo sáu lớp chức năng tương đối độc lập: phục vụ mô hình, theo dõi thí nghiệm, đóng gói container, quan sát hệ thống, điều phối luồng công việc và đảm bảo chất lượng dữ liệu. Phần dưới đây mô tả vai trò và nguyên lý hoạt động của các framework đại diện cho từng lớp; cách thức cấu hình và tích hợp cụ thể trong đề tài được trình bày tại Chương 3.

2.6.1. Phục vụ mô hình: FastAPI và Gradio

Tầng phục vụ mô hình thường tách thành hai vai trò bổ trợ nhau. **FastAPI** (FastAPI, n.d.) là framework backend hiện đại cho REST/JSON API: chạy trên nền ASGI để hỗ trợ xử lý bất đồng bộ, dùng Pydantic để kiểm tra dữ liệu vào/ra dựa trên type hint Python, và tự sinh tài liệu OpenAPI/Swagger UI từ chính khai báo endpoint. Các thành phần chính của FastAPI gồm `APIRouter` (gom nhóm endpoint theo domain), middleware (xử lý CORS, log, metrics ngang qua mọi request) và dependency injection (chia sẻ tài nguyên như model, kết nối DB giữa các endpoint). **Gradio** là thư viện Python sinh giao diện web tương tác trực tiếp từ một hàm Python: chỉ cần khai báo loại đầu vào (ảnh, văn bản, audio) và đầu ra, Gradio tự dựng UI kéo–thả cùng một server nhẹ phục vụ demo. Trong kiến trúc MLOps điển hình, FastAPI đảm nhiệm tích hợp

hệ thống (API JSON thuần), còn Gradio đảm nhiệm trình diễn nhanh và thử nghiệm thủ công của con người.

2.6.2. Theo dõi thí nghiệm và lưu artifact: MLflow + MinIO

MLflow (MLflow, n.d.) là nền tảng mã nguồn mở quản lý vòng đời mô hình theo bốn thành phần. *Tracking* ghi lại params, metrics và artifacts cho mỗi run huấn luyện. *Project* đóng gói code cùng môi trường để chạy lại được trên máy khác. *Model* là một định dạng đóng gói mô hình kèm signature đầu vào/đầu ra. *Registry* đăng ký các phiên bản model theo stage (None / Staging / Production / Archived) phục vụ promote và rollback. Tracking server của MLflow thường được dựng cùng hai thành phần phụ trợ: một quan hệ DB (PostgreSQL/MySQL) lưu metadata run và một object store tương thích S3 lưu artifact. **MinIO** là object store mã nguồn mở triển khai giao thức S3, cho phép lưu artifact theo bucket. Bộ ba MLflow + DB + MinIO tạo thành một *phòng thí nghiệm* chạy hoàn toàn nội bộ, đảm bảo tính kiểm soát phiên bản và tái lập độc lập với các dịch vụ bên ngoài.

2.6.3. Đóng gói và điều phối container: Docker và Docker Compose

Docker (Merkel, 2014; Docker, n.d.) là công nghệ container đóng gói ứng dụng cùng phụ thuộc thành một image tự chứa, chạy được trên mọi máy có Docker engine. So với máy ảo, container nhẹ hơn vì chia sẻ kernel host và khởi động trong vài giây. **Docker Compose** mở rộng Docker để khai báo nhiều container thành một *stack* qua một file YAML: cố định phiên bản image, biến môi trường, cổng mạng, volume bền vững và quan hệ phụ thuộc khởi động (`depends_on`). Lợi thế của Compose so với chạy nhiều lệnh `docker run` rời rạc là tính *khai báo* – toàn bộ topology hệ thống nằm trong một file có thể kiểm soát phiên bản, và lệnh `docker compose up` có thể tái dựng nguyên trạng môi trường trên máy mới.

2.6.4. Quan sát hệ thống: Prometheus, Grafana, Loki và Alertmanager

Lớp quan sát (observability) gồm ba loại dữ liệu vận hành: metrics (số đo định kỳ), logs (sự kiện văn bản có cấu trúc) và traces (đường đi của request qua nhiều dịch vụ). **Prometheus** là cơ sở dữ liệu chuỗi thời gian (TSDB) thiết kế cho metrics: định kỳ *scrape* các endpoint `/metrics` do dịch vụ tự xuất, lưu trữ dưới dạng chuỗi thời gian có label, và truy vấn bằng ngôn ngữ PromQL. **Grafana** là tầng trực quan hoá: nối tới nhiều data source (Prometheus, Loki, Elasticsearch...) và dựng dashboard với các panel time-series, gauge, heatmap. **Loki** là kho log có nhãn theo cùng triết lý của Prometheus – thay vì index toàn văn (như Elasticsearch), Loki chỉ index label, do đó nhẹ và rẻ; truy vấn dùng cú pháp LogQL ngay trong Grafana, đặt cạnh các panel metric tương ứng. **Alertmanager** nhận alert do Prometheus phát ra theo rule, gộp các alert tương tự, dập im theo lịch và định tuyến tới kênh nhận (email, Slack, PagerDuty). Bộ bốn này là phương án *observability* nguồn mở phổ biến cho dịch vụ đóng container.

2.6.5. Điều phối luồng công việc: Apache Airflow

Apache Airflow là nền tảng điều phối luồng công việc dựa trên *Directed Acyclic Graph* (DAG). Mỗi DAG là một file Python định nghĩa các *task* (đơn vị công việc) cùng quan hệ phụ thuộc giữa chúng. Scheduler của Airflow đánh giá lịch chạy, executor đẩy task xuống worker (sequential, local, celery, hoặc kubernetes), và web UI cho phép theo dõi trạng thái, log và lần chạy gần nhất. So với `cron` thuần, Airflow cung cấp bốn thuộc tính then chốt cho pipeline MLOps: retry tự động khi task lỗi, theo dõi quan hệ phụ thuộc tường minh, lưu log từng lần chạy phục vụ debug, và backfill cho khoảng thời gian trong quá khứ. Trong các pipeline MLOps điển hình, Airflow điều phối các luồng như tải dữ liệu, retrain mô hình theo lịch, kiểm tra drift, hay export sang định dạng triển khai khi có model mới được đăng ký.

2.6.6. Drift và đảm bảo chất lượng dữ liệu vận hành

Sau khi triển khai, một mô hình machine learning vận hành trên dòng dữ liệu khác với phân phối huấn luyện và có thể suy giảm chất lượng dự đoán theo thời gian. Lu và cộng sự cung cấp một tổng quan hệ thống cho hiện tượng này (Lu và cộng sự, 2018); ở đây ta tóm tắt theo ba khái niệm bổ trợ nhau. **Data drift** (còn gọi là *covariate shift*) xảy ra khi phân phối đầu vào $P(X)$ thay đổi trong khi quan hệ điều kiện $P(Y \mid X)$ không đổi – ví dụ camera đổi vị trí làm độ sáng và góc nhìn lệch khỏi tập train. **Concept drift** (còn gọi là *label drift*) xảy ra khi $P(Y \mid X)$ thay đổi, tức cùng đầu vào nhưng *ý nghĩa* nhãn đã khác – ví dụ tiêu chí định nghĩa lớp *motorbike* bị mở rộng để bao gồm cả xe scooter điện. **Model drift** là khái niệm bao trùm: suy giảm chỉ số chất lượng (accuracy, mAP) trên dòng dữ liệu mới, có thể do data drift, concept drift hoặc kết hợp cả hai.

Theo hình thái diễn tiến, drift còn được chia thành bốn dạng phụ. *Sudden drift* xảy ra đột ngột (camera mới được gắn). *Gradual drift* dịch chuyển từ từ (thời tiết theo mùa). *Incremental drift* là chuỗi thay đổi nhỏ tích luỹ theo thời gian. *Recurring drift* lặp lại có chu kỳ (ngày – đêm, ngày làm việc – cuối tuần). Phân loại này quan trọng vì lựa chọn ngưỡng cảnh báo và lịch retrain phụ thuộc trực tiếp vào hình thái drift dự kiến.

Để đo drift, ta so sánh phân phối của một feature giữa tập tham chiếu $\mathcal{R}$ (thường là validation lúc train) và tập sản phẩm $\mathcal{P}$ (thu được khi vận hành). Bốn thống kê phổ biến gồm:

- **Population Stability Index (PSI):** $\mathrm{PSI} = \sum_i (q_i - r_i) \ln(q_i / r_i)$, với r_i, q_i là tỷ lệ tần suất tại bin thứ i ở $\mathcal{R}$ và $\mathcal{P}$; ngưỡng cảnh báo điển hình $\mathrm{PSI} > 0{,}2$.
- **Kolmogorov–Smirnov:** $D = \sup_x |F_{\mathcal{R}}(x) - F_{\mathcal{P}}(x)|$ với F là hàm phân phối tích luỹ; dùng cho feature liên tục.
- **Jensen–Shannon divergence:** biến thể đối xứng và bị chặn của KL, $D_{\mathrm{JS}}(P\|Q) = \frac{1}{2} D_{\mathrm{KL}}(P\|M) + \frac{1}{2} D_{\mathrm{KL}}(Q\|M)$ với $M = (P+Q)/2$.
- **Wasserstein distance:** đo *khoảng cách vận chuyển* giữa hai phân phối, hữu ích khi hai phân phối không trùng support.

Đối với object detection, các feature drift đáng quan tâm gồm phân phối kích thước đối tượng (bounding box), số object trung bình trên ảnh, phân phối lớp đối tượng, và đặc trưng ảnh tổng thể (độ sáng, độ tương phản, histogram màu). Khi drift đáng kể, mAP đo trên dòng dữ liệu sản phẩm có thể giảm âm thầm trong khi mAP trên tập validation gốc không đổi – đây chính là dấu hiệu cần retrain hoặc fine-tune lại với dữ liệu mới.

Deepchecks là thư viện Python phổ biến để hiện thực hoá các kiểm tra trên. Thư viện cung cấp các *suite* có sẵn cho cả tabular, vision và object detection: so sánh phân phối feature giữa $\mathcal{R}$ và $\mathcal{P}$ qua PSI/KS, phát hiện label drift, kiểm tra chất lượng nhãn và xuất báo cáo HTML có thể xem trực tiếp. Trong kiến trúc MLOps điển hình, kết quả các kiểm tra Deepchecks được đẩy lên Prometheus dưới dạng metric để Alertmanager phát cảnh báo, hoặc làm tín hiệu kích hoạt một DAG Airflow chạy retrain – đóng vòng phản hồi tự duy trì chất lượng *phát hiện drift* $\rightarrow$ *huấn luyện lại*.

2.6.7. CI/CD và trigger trong MLOps

Một pipeline CI/CD điển hình đi qua các bước commit $\rightarrow$ build $\rightarrow$ test $\rightarrow$ report $\rightarrow$ release (staging) $\rightarrow$ deploy (production). Khi chuyển sang MLOps, chuỗi này phải bao phủ thêm ba nguồn thay đổi: code, dữ liệu và mô hình. Trigger trong MLOps vì vậy không chỉ đến từ thay đổi code (CI build image, chạy test API) mà còn có thể đến từ một model mới được đăng ký vào Model Registry (orchestrator chạy export + benchmark), hoặc từ tín hiệu drift ngoài production (orchestrator gọi job retrain). Điểm khác cốt yếu so với CI/CD phần mềm truyền thống là *mô hình và dữ liệu* cũng trở thành tài nguyên có phiên bản và có trigger riêng, không chỉ mã nguồn.

2.7. Một số nghiên cứu gần đây liên quan đến đề tài

Bảng 2.1: Một số nghiên cứu gần đây liên quan đến real-time object detection và distillation

Công trình	Ý chính	Liên quan đến đề tài
YOLOv10: Real-Time End-to-End Object Detection (Wang và cộng sự, 2024)	Loại bỏ phụ thuộc NMS trong huấn luyện end-to-end và tối ưu đồng thời hiệu quả–độ chính xác. Tác giả báo cáo YOLOv10-B có độ trễ thấp hơn 46% và ít hơn 25% tham số so với YOLOv9-C ở cùng mức hiệu năng.	Củng cố luận điểm rằng tối ưu kiến trúc detector là hướng song song với KD khi muốn giảm latency cho suy luận thời gian thực.
Teach YOLO to Remember: A Self-Distillation Approach for Continual Object Detection (Dalle Pezze và cộng sự, 2025)	Đề xuất self-distillation cho continual object detection; báo cáo cải thiện mAP trong kịch bản học tăng dần.	Gợi ý hướng mở rộng từ KD teacher–student sang self-distillation hoặc continual learning cho dữ liệu vận hành mới.
Task Integration Distillation for Object Detectors (Task Integration Distillation, 2024)	Nhấn mạnh cần distill đồng thời phân loại và hồi quy thay vì chỉ tập trung một nhiệm vụ.	Phù hợp với cấu hình KD trong repo, nơi loss đã tách riêng dense logits, sparse logits và box loss.
Focal and Global Knowledge Distillation for Detectors (Yang và cộng sự, 2022)	Kết hợp distillation theo vùng tiêu điểm và toàn cục trên feature map của detector.	Củng cố cơ sở lý thuyết cho việc không dùng công thức KD quá đơn giản trong detection.

Từ góc nhìn tổng quan, có thể chia các công trình liên quan thành ba nhóm. Nhóm thứ nhất tối ưu detector ở cấp kiến trúc, điển hình như YOLOv10, nhằm giảm chi phí hậu xử lý và cải thiện biên hiệu năng–độ chính xác. Nhóm thứ hai tập trung vào KD cho object detection, nhấn mạnh rằng distillation cần chạm đồng thời vào phân loại, localization và đôi khi cả đặc trưng trung gian. Nhóm thứ ba mở rộng bài toán sang continual detection hoặc deployment-oriented evaluation, tức không chỉ hỏi mô hình có chính xác hay không mà còn hỏi mô hình có giữ được chất lượng khi bối cảnh dữ liệu thay đổi hay không.

Điểm thú vị là đề tài của nhóm không trùng hoàn toàn với bất kỳ nhánh nào phía trên. Nhóm không đề xuất một kiến trúc detector mới, cũng chưa xây dựng một thuật toán KD mới. Giá trị của đề tài nằm ở việc kết hợp các ý tưởng đã có thành một pipeline triển khai tương đối hoàn chỉnh: train, track, export, serve, monitor. Vì vậy, khi định vị đóng góp học thuật, báo cáo nên tránh diễn đạt theo hướng *đề xuất thuật toán mới*; thay vào đó nên nhấn mạnh đây là một *implementation-oriented study* cho bài toán tối ưu deployment.

Chương 3: TRIỂN KHAI THỬ NGHIỆM HỆ THỐNG

3.1. Cấu trúc kho mã nguồn

Toàn bộ mã nguồn của hệ thống được tổ chức theo nguyên tắc một thư mục đảm nhiệm một mối quan tâm: chuẩn bị dữ liệu, huấn luyện, phục vụ, hạ tầng MLOps và artifact mô hình được tách thành các thư mục độc lập, có thể chạy và kiểm thử riêng. Sơ đồ cây dưới đây liệt kê cấu trúc thư mục cấp cao của kho mã nguồn.

```
hpc_nhom1_code/
├── README.md ..... Hướng dẫn kiến trúc và khởi chạy nhanh.
├── PROJECT_EXECUTION.md ..... Quy trình triển khai chi tiết tám bước.
├── data_pipeline/ ..... Tải dataset, tạo subset, sinh biến thể ảnh, quản lý phiên bản dữ liệu.
├── data_validation/ ..... Kiểm tra chất lượng và phát hiện drift bằng Deepchecks.
├── training_pipeline/
│   └── src/
│       ├── train.py ..... Vòng huấn luyện KD đa thành phần (teacher + student).
│       └── config/train.yaml ..... Cấu hình distillation và siêu tham số.
├── scripts/
│   └── train_teacher_model.py ..... Huấn luyện teacher kèm log MLflow.
├── notebooks/
│   ├── colab_result.ipynb ..... Lần huấn luyện chính thức trên Colab T4.
│   ├── colab_benchmark.ipynb ..... Quy trình đo chuẩn mAP và latency.
│   └── trainer.py ..... Bản patch của Ultralytics hỗ trợ KD.
├── serving_pipeline/
│   ├── api/ ..... Ứng dụng FastAPI và các router.
│   ├── gradio_app.py ..... Giao diện Gradio demo.
│   ├── docker-compose.yml ..... Stack serving (CPU; profile GPU).
│   └── production/, production-data/ ..... Ảnh và prediction lưu cho hậu kiểm.
├── infra/
│   ├── mlflow/ ..... Stack MLflow + MinIO + MySQL.
│   ├── monitor/ ..... Stack Prometheus + Grafana + Loki + Alertmanager.
│   └── airflow/ ..... Stack Airflow với DAG train/drift/export.
├── model_artifacts/
│   ├── teacher_best.pt, student_best.pt, student_kd_best.pt
│   └── serving_model.pt, serving_model.onnx, serving_model.engine
├── serving_model.pt ..... Bản sao của student_kd_best.pt phục vụ demo.
├── reports/
│   ├── logs/ ..... Log runtime FastAPI và Gradio.
│   └── benchmark/ ..... report_results.json, quality.csv, latency.csv.
├── outputs/ ..... Báo cáo cuối kỳ, figures, slide.
├── docs/ ..... Tài liệu kỹ thuật và hướng dẫn vận hành.
└── assets/ ..... Sơ đồ pipeline và hình ảnh minh hoạ.
```

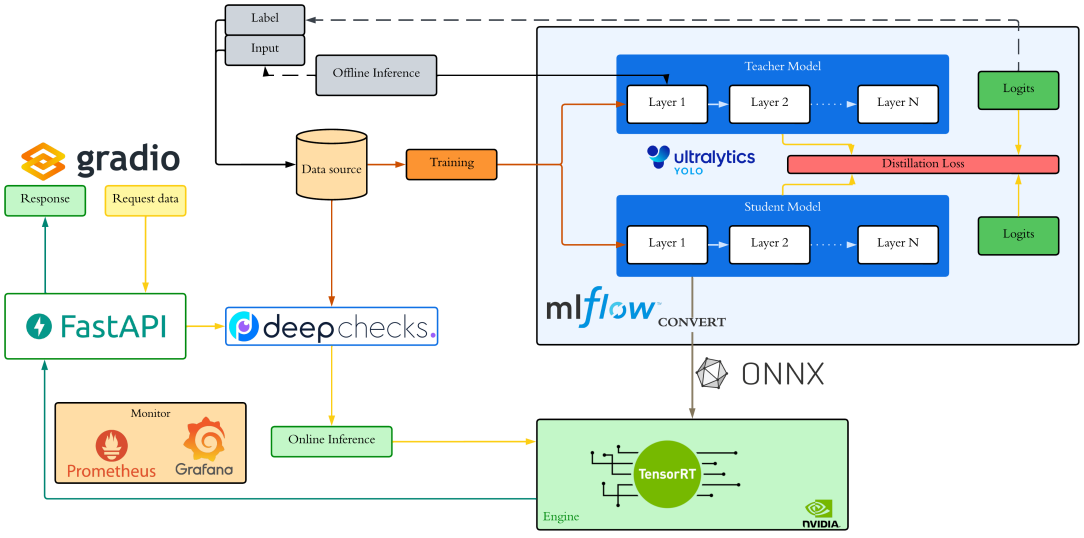
Sơ đồ cây thư mục của kho mã nguồn – mỗi thành phần ánh xạ tới một lớp chức năng trình bày ở Mục 2.6.

Cách tổ chức này ưu tiên ba nguyên tắc thiết kế. *Thứ nhất*, ranh giới giữa

`data_pipeline/` (dữ liệu cho huấn luyện) và `data_validation/` (dữ liệu lúc vận hành) cho phép phân tích drift hoạt động độc lập với pipeline huấn luyện và không phụ thuộc trạng thái của các artifact đang train. *Thứ hai*, mọi thành phần hạ tầng nằm trong `infra/` đều đi kèm một `docker-compose.yml` riêng, nên từng cụm có thể bật hoặc tắt theo dung lượng tài nguyên của máy và theo nhu cầu demo cụ thể. *Thứ ba*, mọi artifact mô hình tập trung trong `model_artifacts/` với quy ước tên cố định cho từng vai trò (teacher, student baseline, student KD, serving), tạo tiền đề để xác minh provenance bằng checksum SHA1 trước khi triển khai.

3.2. Kiến trúc pipeline tổng thể

Hệ thống được thiết kế dưới dạng một pipeline chín giai đoạn tuần tự, mỗi giai đoạn có đầu vào – đầu ra rõ ràng và có thể chạy độc lập khi cần kiểm thử (Hình 3.1). Bốn giai đoạn đầu thuộc trục *dữ liệu và huấn luyện*: tải dataset từ Kaggle, chuẩn hoá nhãn YOLO và tạo subset, huấn luyện teacher, huấn luyện student baseline và student có Knowledge Distillation. Năm giai đoạn sau thuộc trục *triển khai và vận hành*: lưu artifact đã train kèm checksum, export sang ONNX và TensorRT, đóng gói FastAPI làm backend inference, Gradio làm UI demo, và bộ Prometheus + Grafana + Loki + Airflow đảm nhiệm quan sát và điều phối các job định kỳ. Tách hai trục thay vì hợp nhất một pipeline tuyến tính cho phép tầng triển khai vận hành độc lập sau khi đã có artifact và không bị ràng buộc bởi tiến độ huấn luyện.



Hình 3.1: Pipeline tổng thể chín giai đoạn: từ tải dataset Kaggle, qua data pipeline (chuẩn hoá, subset, version qua MinIO), train teacher – student – KD, lưu artifact và export ONNX/TensorRT, đến serving (FastAPI + Gradio) và quan sát (Prometheus + Grafana + Loki + Airflow). Nguồn: nhóm tác giả.

3.3. Data pipeline

Tầng *data pipeline* chuẩn bị dữ liệu cho huấn luyện và đồng thời tạo lớp tham chiếu cho phân tích drift sau khi triển khai. Bốn module được thiết kế độc lập về chức năng, mỗi module có một

lệnh con riêng dưới CLI `python -m data_pipeline`:

- `kaggle_download.py`: kết nối Kaggle API, tải tập *Traffic Detection Project* (Sardoan, n.d.) và tổ chức ảnh kèm nhãn YOLO theo cấu trúc thư mục chuẩn `train/`, `valid/`, `test/`.
- `subset_creator.py`: lấy mẫu một tập con kích thước cấu hình được qua tham số `--size`, vẫn giữ phân phối lớp gần với tập gốc, phục vụ vòng thử nghiệm nhanh hoặc demo.
- `test_variants.py`: tự động sinh các biến thể ảnh test theo bốn biến đổi nhiễu thường gặp trong giao thông (mờ động, nhiễu Gauss, thay đổi độ sáng, sương), phục vụ đánh giá độ bền của detector ngoài điều kiện ảnh sạch.
- `version_manager.py`: đẩy phiên bản dataset lên MinIO làm object store, gắn tag cho mỗi phiên bản để các pipeline huấn luyện về sau có thể tham chiếu lại đúng phiên bản đã dùng.

Việc tách module thành bốn vai trò trên đảm bảo hai trục *dữ liệu cho huấn luyện* và *dữ liệu vận hành phục vụ drift* hoàn toàn độc lập. Trục thứ hai được trình bày kỹ ở Mục 3.6.5; phần lý thuyết drift đã đặt nền ở Chương 2.

[Chèn ảnh tại đây]

Lưới 1×5 ảnh: 1 ảnh gốc từ Traffic Detection Project + 4 biến thể; có thể lấy từ `data/demo_subset/test/` sau khi chạy `python -m data_pipeline dataset test-variants`.

Hình 3.2: Ví dụ tập dữ liệu và bốn biến thể nhiễu sinh bởi `test_variants.py`: ảnh gốc, mờ động (blur), nhiễu Gauss, độ sáng thấp và sương – phục vụ đánh giá độ bền detector.

3.4. Training pipeline

3.4.1. Cặp teacher–student

Cặp teacher–student được chọn trong cùng họ YOLO26 với chênh lệch kích thước có chủ đích: teacher `yolo26x` (58.82 M tham số) đóng vai trò *nguồn tri thức*, còn student `yolo26n` (2.51 M tham số) là *đích triển khai* với chi phí suy luận thấp. Khoảng cách hơn $23\times$ về số tham số tạo *biên* đủ rộng để Knowledge Distillation phát huy hiệu lực; nếu hai mô hình quá gần nhau, không gian cải thiện cho student bị thu hẹp. Hai mô hình chia sẻ chung sườn backbone – neck – detection head (Mục 2.2), nên việc ghép feature map theo P3/P4/P5 và áp dụng phép chiếu 1×1 giữa hai mạng là trực tiếp.

3.4.2. Vòng huấn luyện teacher

Vòng huấn luyện teacher được đóng gói trong `scripts/train_teacher_model.py`. Script tiếp nhận các tham số dòng lệnh `--model`, `--data`, `--epochs`, `--batch`, `--imgsz`, `--device` và `--model-name` để cố định mọi siêu tham số ảnh hưởng đến kết quả, bảo đảm tính tái lập giữa các lần chạy. Khi bắt đầu, script khởi tạo một MLflow run trong experiment `teacher-training`, log toàn bộ params và metrics theo epoch, lưu artifact `best.pt` sau khi huấn luyện và đăng ký model vào MLflow Registry dưới tên do `--model-name` chỉ định. Cờ `--no-mlflow` cho phép tắt nhánh tracking khi cần chạy thử trên môi trường không có MLflow server (ví dụ Colab độc lập), giữ pipeline vận hành được trong cả hai kịch bản.

3.4.3. Vòng huấn luyện student với Knowledge Distillation

Vòng huấn luyện student có distillation nằm trong `training_pipeline/src/train.py`. Script nạp đồng thời teacher weights, student weights và file cấu hình YAML, sau đó gọi `student_model.train(...)` với hai đối số đặc biệt là `teacher` (mạng đã huấn luyện trước, dùng làm nguồn tín hiệu KD) và `distillation_config_loss` (cấu hình trọng số của các thành phần loss). Hai đối số này chỉ có trong bản patch Ultralytics của nhóm (lưu tại `notebooks/trainer.py`); Ultralytics gốc không hỗ trợ. Trong mỗi vòng lặp huấn luyện, một batch được chạy forward đồng thời qua teacher (đặt ở chế độ `eval()`, đông cứng trọng số) và student, sau đó tổng loss được tính theo công thức ở Mục 2.4.5 và lan truyền ngược chỉ qua student.

3.4.4. Cấu hình distillation

Tệp `training_pipeline/src/config/train.yaml` cố định năm siêu tham số distillation được dùng cho mọi lần chạy chính thức:

- `logit_temperature` $= 3{,}0$: nhiệt độ softmax áp lên logits trước khi đo KL divergence.
- `dense_logit_weight` $= 0{,}25$: hệ số cho thành phần KL trên toàn bộ 8 400 prediction points.
- `sparse_logit_weight` $= 0{,}25$: hệ số cho thành phần KL chỉ lấy các điểm có objectness cao của teacher.
- `box_loss_weight` $= 0{,}5$: hệ số cho loss bounding box.
- `box_objectness_threshold` $= 0{,}3$: ngưỡng objectness của teacher để chọn các prediction point dùng cho box loss.

Năm tham số trên hiện thực hoá đúng tinh thần KD đa thành phần loss đã trình bày lý thuyết ở Mục 2.4.

[Chèn ảnh tại đây]

Ảnh chụp MLflow UI tại `http://localhost:5000` sau khi đã chạy bước 5 và 6 (Mục 3.7): danh sách run + biểu đồ mAP/loss theo epoch.

Hình 3.3: Giao diện MLflow Tracking liệt kê các run `teacher-training` và `traffic-distillation` kèm metrics theo epoch.

3.5. Serving pipeline

3.5.1. Backend FastAPI

Backend serving được tổ chức theo ba lớp tách biệt trong `serving_pipeline/api/`: tầng router theo từng domain (`routers/detection.py` cho phát hiện đối tượng, `routers/drift.py` cho kiểm tra drift), tầng schema khai báo dữ liệu vào – ra bằng Pydantic, và tầng middleware xử lý CORS, log cùng metrics ngang qua mọi request. Endpoint chính `POST /detect` nhận ảnh upload, chạy mô hình YOLO đã nạp một lần ở khởi động, hậu xử lý kết quả với các ngưỡng `confidence_threshold` và `iou_threshold` cấu hình được, rồi trả về JSON gồm danh sách bounding box, lớp dự đoán, confidence và thời gian suy luận. Khi khởi động, ứng dụng đăng ký một `prometheus-fastapi-instrumentator` để tự xuất các metric tại endpoint `/metrics`, đồng thời khởi tạo các `Gauge` tuỳ biến cho mức tiêu thụ RAM và VRAM của tiến trình serving.

3.5.2. Giao diện Gradio

Tệp `serving_pipeline/gradio_app.py` hiện thực giao diện Gradio cho phép người dùng tải ảnh giao thông và xem kết quả phát hiện ngay trong trình duyệt. Gradio không tự chạy mô hình mà gọi sang FastAPI qua HTTP, giữ một nguồn-sự-thật duy nhất cho logic inference (Mục 3.5.1) và tách bạch giao diện tương tác khỏi backend tích hợp. Phân vai trò này cho phép thay UI khi cần mà không phải sửa backend, đồng thời một backend có thể đồng thời phục vụ nhiều giao diện (Gradio cho demo, REST client cho các hệ thống tích hợp khác).

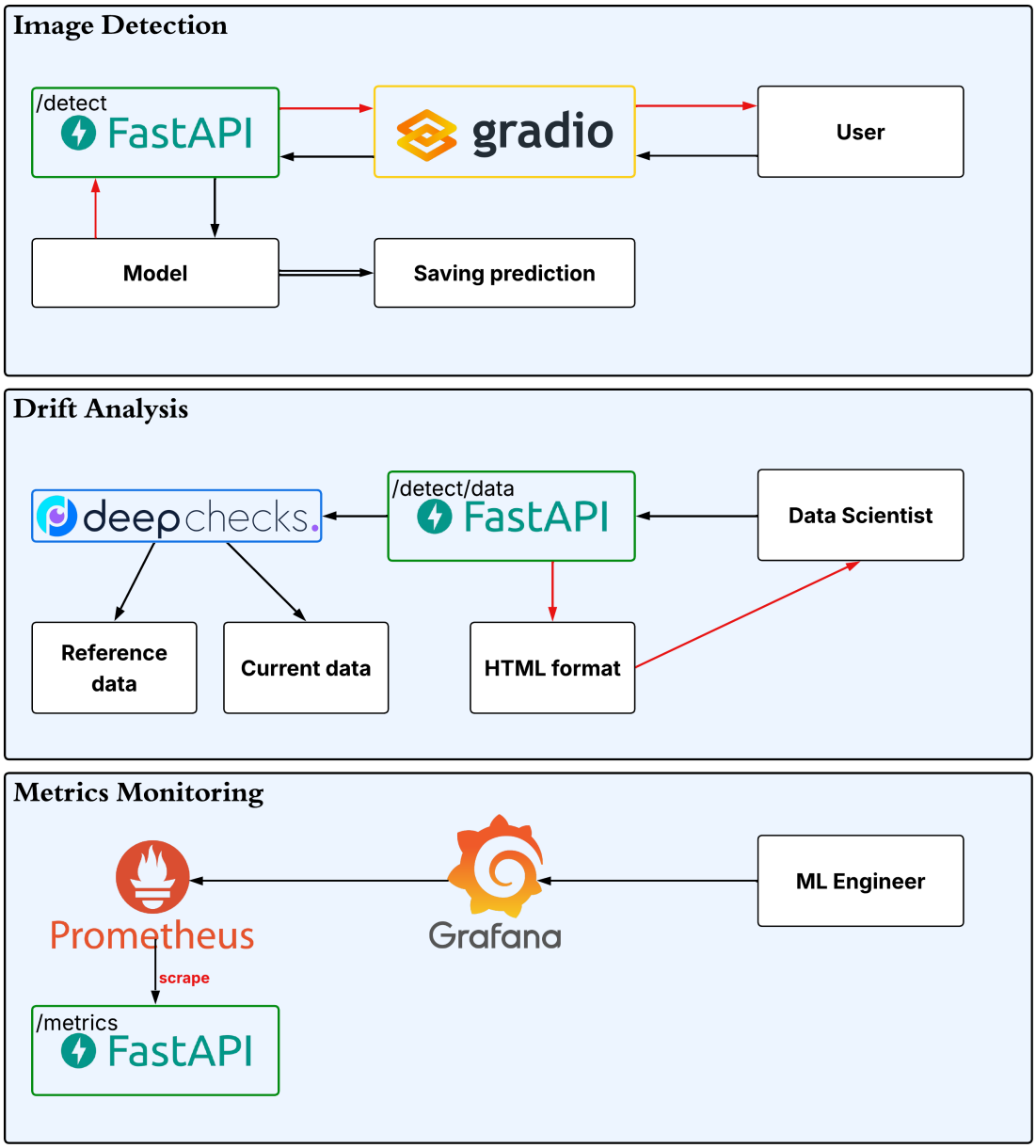
3.5.3. Lưu prediction phục vụ hậu kiểm

Mỗi request `/detect` thành công đều kèm một thao tác lưu cặp (ảnh đầu vào, prediction) vào thư mục `serving_pipeline/production-data/` với tên file mang dấu thời gian và một UUID ngắn. Prediction được ghi theo định dạng YOLO thay vì JSON để các thư viện phân tích như Deepchecks (Mục 3.6.5) đọc trực tiếp như một tập detection độc lập và đối chiếu với tập tham

chiếu. Đây là cơ chế đóng vòng cho drift detection: dữ liệu vận hành tự tích luỹ thành tập so sánh theo thời gian thực, không cần thêm bước thu thập riêng.

3.6. Hạ tầng MLOps và cấu hình stack

Toàn bộ hạ tầng được đóng gói thành bốn `docker-compose` stack độc lập, chia sẻ một docker network chung để các container có thể tham chiếu nhau qua hostname mà không phải phơi cổng ra host (Hình 3.4). Tách stack theo vai trò thay vì gộp vào một compose duy nhất cho phép từng cụm bật/tắt độc lập theo dung lượng tài nguyên và theo nhu cầu cụ thể của mỗi lần demo.



Hình 3.4: Sơ đồ tổng thể bốn stack hạ tầng và quan hệ giữa chúng: serving (FastAPI + Gradio), tracking (MLflow + MinIO + MySQL), monitoring (Prometheus + Grafana + Loki + Alertmanager) và orchestration (Airflow). Các mũi tên thể hiện luồng metric, log, artifact và trigger giữa các stack. Nguồn: nhóm tác giả.

3.6.1. Stack serving

Stack `serving_pipeline` gồm hai container chính: `api` chạy FastAPI bằng Uvicorn (cổng 8000) và `ui` chạy Gradio (cổng 7860). Một profile `gpu` bổ sung container `api-gpu` dùng image base có CUDA runtime để chạy inference TensorRT khi máy đích có GPU NVIDIA. Tách `api` và `api-gpu` thay vì gộp chung cho phép cùng một file compose chạy được trên cả máy chỉ-CPU lẫn máy có GPU bằng cách bật hoặc không bật profile.

3.6.2. Stack tracking: MLflow + MinIO + MySQL

Stack `infra/mlflow` chạy ba container đảm nhiệm ba vai trò bổ trợ nhau. MLflow Tracking Server (cổng 5000) phục vụ truy vấn UI và API; MinIO (cổng API 9000, console 9001) đóng vai trò object store tương thích S3 lưu artifact; MySQL nhỏ lưu metadata run. MLflow được khởi động với `--default-artifact-root s3://mlflow/artifacts/` trỏ về MinIO và `--backend-store-uri` trỏ về MySQL, tạo phân chia trách nhiệm rạch ròi: dữ liệu lớn vào object store, metadata cấu trúc vào quan hệ DB. Các script huấn luyện tiếp nhận tham số `--mlflow-tracking-uri` qua dòng lệnh, cho phép chuyển nhanh giữa tracking nội bộ và một server từ xa khi cần chia sẻ kết quả giữa các thành viên.

3.6.3. Stack monitoring: Prometheus + Grafana + Loki + Alertmanager

Stack `infra/monitor` đóng gói bốn container chính ứng với bốn vai trò observability đã trình bày ở Mục 2.6.4, kèm ba container phụ trợ để mở rộng phạm vi giám sát. *Bốn thành phần chính*: Prometheus mount file `prometheus.yml` chứa các scrape job trỏ vào endpoint `/metrics` của serving qua docker network chung; Grafana mount sẵn `datasources.yaml` khai báo Prometheus và Loki làm data source kèm thư mục `dashboards/` tự động nạp, cung cấp sẵn các panel cho request rate, p95 latency, lỗi 5xx và mức tiêu thụ tài nguyên; Loki nhận log qua driver log của Docker hoặc qua sidecar Promtail, truy vấn LogQL ngay trong Grafana đặt cạnh các panel metric tương ứng; Alertmanager khởi động với cấu hình kênh nhận tối thiểu (null route mặc định) và được cập nhật `alertmanager.yml` với webhook Slack hoặc SMTP khi cần bật cảnh báo thực. *Ba thành phần phụ trợ*: `node_exporter` xuất các metric phần cứng host (CPU, RAM, disk, network); `dcgm-exporter` của NVIDIA xuất metric GPU khi máy có GPU; `pushgateway` làm cầu nối cho các job batch ngắn (như Airflow DAG) đẩy metric vào Prometheus mà không cần endpoint scrape thường trực.

[Chèn ảnh tại đây]

Ảnh chụp Grafana tại `http://localhost:3000` với data source Prometheus, các panel cho `/detect`: số request/s, p95 latency, lỗi 5xx và biểu đồ tài nguyên.

Hình 3.5: Dashboard Grafana giám sát dịch vụ serving: request rate, p95 latency, mã trạng thái và mức tiêu thụ RAM/VRAM theo thời gian thực.

3.6.4. Stack orchestration: Airflow

Stack `infra/airflow` dùng Airflow với executor `CeleryExecutor`, PostgreSQL làm metadata database và Redis làm message broker giữa scheduler và worker – cấu hình chuẩn cho phép mở rộng song song các DAG mà không bị giới hạn bởi một process duy nhất. Thư mục `dags/` chứa bốn DAG phục vụ vòng đời MLOps:

- `train_model_dag.py` – pipeline huấn luyện teacher / student / student KD, chạy theo lịch hoặc theo trigger thay đổi dữ liệu.
- `daily_drift_detection_dag.py` – chạy suite Deepchecks định kỳ hàng ngày trên tập sản phẩm thu được từ serving (xem Mục 3.6.5).
- `convert_tensorrt_dag.py` – export sang định dạng TensorRT khi một model mới được promote sang stage `Staging` trong MLflow Registry.
- `daily_production_inference_dag.py` – chạy inference định kỳ trên một batch ảnh sản phẩm để theo dõi chất lượng dự đoán theo thời gian.

Web UI Airflow mặc định mở tại cổng 8080.

[Chèn ảnh tại đây]

Ảnh chụp graph view của `train_model_dag.py` (hoặc một trong bốn DAG khác: `daily_drift_detection_dag`, `convert_tensorrt_dag`, `daily_production_inference_dag`) tại `http://localhost:8080`.

Hình 3.6: Sơ đồ DAG `train_model_dag` trên Airflow Web UI: quan hệ phụ thuộc giữa các task tải dữ liệu, train teacher, train student KD và đăng ký artifact.

3.6.5. Kiểm tra drift với Deepchecks

Tầng kiểm tra chất lượng dữ liệu vận hành nằm trong thư mục `data_validation/`, gồm ba module: `dataset_loader.py` bọc tập ảnh và nhãn YOLO thành đối tượng `VisionData` của Deepchecks, `drift_analysis.py` hiện thực các suite kiểm tra, và `generate_predictions.py` sinh prediction trên tập sản phẩm để so sánh với tập tham chiếu. Tập tham chiếu lấy từ thư mục validation lúc huấn luyện; tập sản phẩm lấy từ `serving_pipeline/production-data/` – nơi FastAPI tự động tích luỹ cặp ảnh – prediction (Mục 3.5.3). Mỗi lần chạy, suite thực hiện ba kiểm tra Deepchecks cốt lõi cho thị giác máy tính: `ImagePropertyDrift` (độ sáng, độ tương phản, độ mờ), `LabelDrift` (phân phối lớp) và tùy chọn `PredictionDrift` (phân phối confidence của model trên hai tập). Kết quả được xuất ra báo cáo HTML đầy đủ và các chỉ số tóm tắt được đẩy lên Prometheus. Router `/drift` trong FastAPI cho phép gọi suite này theo yêu cầu để debug nhanh, còn *daily_drift_detection_dag* của Airflow chịu trách nhiệm chạy định kỳ và phát alert khi mức drift vượt ngưỡng cấu hình.

[Chèn ảnh tại đây]

Trang đầu báo cáo HTML do `data_validation/` sinh ra, gồm bảng tóm tắt các kiểm tra (passed/failed) và 1-2 biểu đồ phân phối feature minh hoạ drift điển hình.

Hình 3.7: Trích báo cáo HTML Deepchecks: so sánh phân phối kích thước bounding box, độ sáng ảnh và tỷ lệ class giữa tập tham chiếu và tập sản phẩm, kèm chỉ số PSI và cảnh báo theo từng kiểm tra.

3.7. Quy trình thực thi end-to-end

Toàn bộ pipeline được khởi chạy theo tám bước tuần tự dưới đây, kèm các lệnh cốt lõi tương ứng để một thành viên mới có thể dựng lại hệ thống từ đầu.

Bước 1 – Tạo Docker network dùng chung. Trước khi bật các stack, tạo một bridge network để bốn cụm có thể tham chiếu nhau qua hostname container.

Bước 2 – Khởi động stack tracking. Bật MLflow + MinIO + MySQL bằng `docker compose up -d` trong thư mục `infra/mlflow/`; MLflow UI sẵn sàng tại `http://localhost:5000`.

Bước 3 – Khởi động stack monitoring. Bật Prometheus, Grafana, Loki và Alertmanager bằng `docker compose up -d` trong `infra/monitor/`; Grafana sẵn sàng tại `http://localhost:3000`.

Bước 4 – Chuẩn bị dữ liệu. Tải dataset, tạo subset và đẩy phiên bản dữ liệu lên MinIO:

```
python -m data_pipeline kaggle download \
  --dataset yusufberksardoan/traffic-detection-project \
  --output data/raw --organize

python -m data_pipeline dataset subset \
  --input data/raw --output data/demo_subset --size 1000
```

Bước 5 – Huấn luyện teacher. Train YOLO26x với log MLflow đầy đủ:

```
python scripts/train_teacher_model.py \
  --data data/demo_subset/data.yaml \
  --model yolo26x.pt \
  --epochs 30 --batch 4 --imgsz 640 --device 0 \
  --model-name yolo-teacher-model
```

Bước 6 – Huấn luyện student với Knowledge Distillation. Truyền teacher weights vừa huấn luyện vào vòng KD:

```
python training_pipeline/src/train.py \
  training_pipeline/src/config/train.yaml \
  --teacher-weights runs/teacher/teacher_yolo/weights/best.pt \
  --student-weights yolo26n.pt \
  --data data/demo_subset/data.yaml \
  --mlflow-tracking-uri http://localhost:5000 \
  --mlflow-experiment traffic-distillation \
  --mlflow-run-name student_kd
```

Bước 7 – Khởi động stack serving. Sao chép checkpoint student KD thành `serving_model.pt` (cùng checksum, xem Phụ lục 4.6.5) và bật stack:

```
cd serving_pipeline
docker compose up -d api ui          # CPU
# hoac:
docker compose --profile gpu up -d   # GPU + TensorRT
```

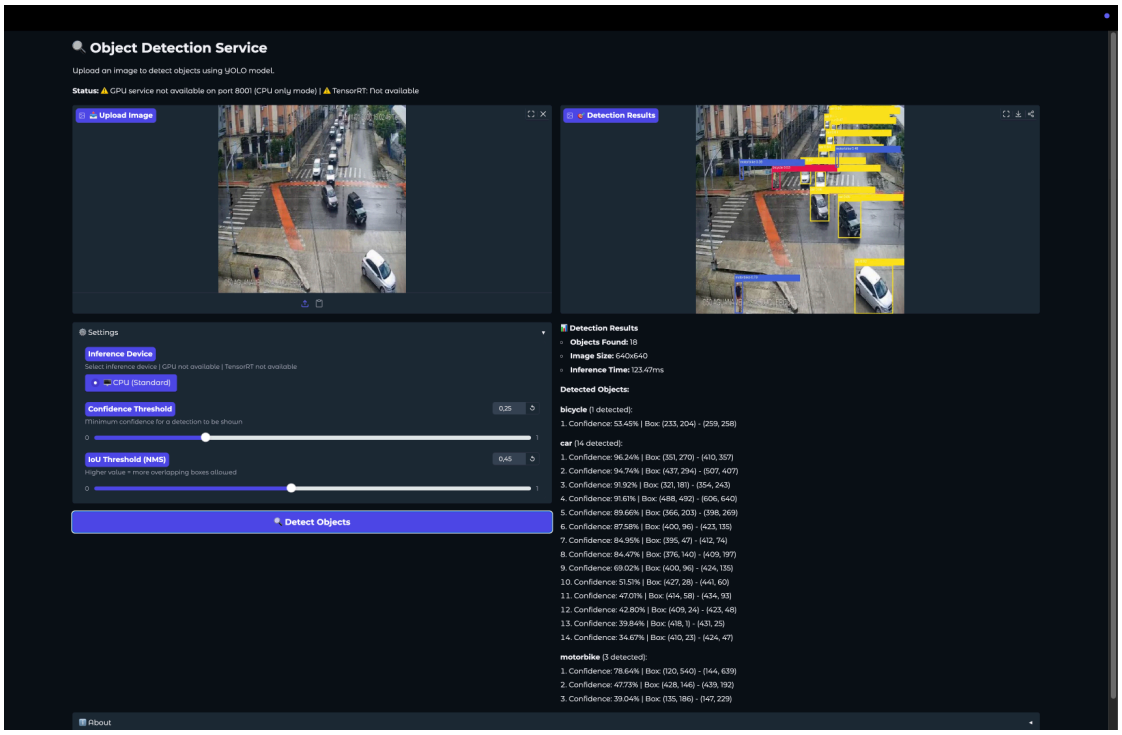
Bước 8 – Đo benchmark. Chạy notebook `colab_benchmark.ipynb` (hoặc môi trường tương đương có GPU) để sinh các chỉ số chất lượng và latency theo protocol thống nhất ở Mục 4.2; kết quả được xuất ra `reports/benchmark/` làm dữ liệu nguồn cho Chương 4.

3.8. Ứng dụng demo

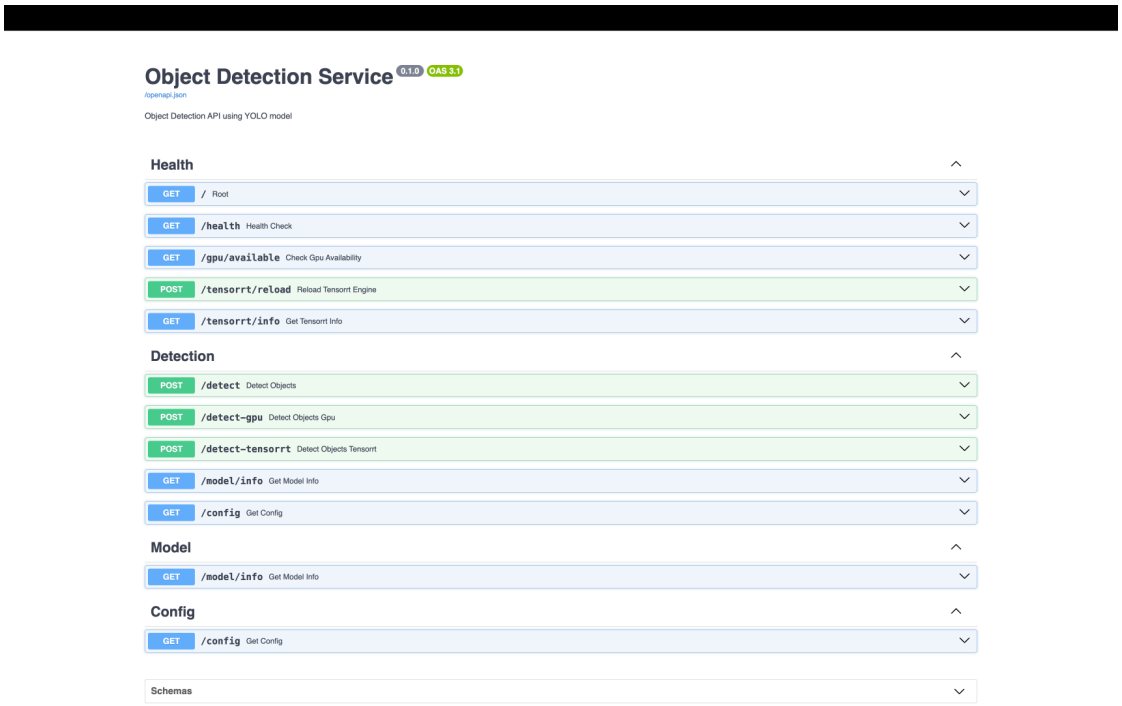
3.8.1. Kiến trúc hai lớp giao diện

Ứng dụng demo có hai lớp giao diện bổ trợ nhau. **FastAPI** đóng vai trò backend tích hợp hệ thống: tự sinh tài liệu OpenAPI tại `/docs` (Hình 3.9) và phơi endpoint `/detect` dưới dạng

REST/JSON để mọi client (dịch vụ khác, script kiểm thử, UI tuỳ biến) đều có thể gọi. **Gradio** đóng vai trò UI demo cho con người: cung cấp giao diện kéo – thả ảnh (Hình 3.8) và hiển thị bounding box, nhãn lớp, confidence ngay trong trình duyệt mà không cần dựng frontend riêng. Hai vai trò được tách bạch trên hai container độc lập để có thể thay thế hoặc mở rộng độc lập.



Hình 3.8: Giao diện Gradio demo: người dùng tải ảnh giao thông và xem bounding box, nhãn lớp, confidence trực tiếp trên trình duyệt. Nguồn: nhóm tác giả.



Hình 3.9: Tài liệu OpenAPI/Swagger UI được FastAPI sinh tự động tại endpoint `/docs`, mô tả contract đầu vào – đầu ra của `/detect` và các endpoint phụ trợ. Nguồn: nhóm tác giả.

3.8.2. Luồng tương tác cơ bản

Người dùng tải một ảnh giao thông vào giao diện Gradio. Ảnh được gửi sang FastAPI qua HTTP, mô hình YOLO chạy inference với hai ngưỡng `confidence_threshold` và `iou_threshold` cấu hình được, kết quả trả về dưới dạng JSON gồm danh sách bounding box, lớp, confidence và thời gian xử lý. Gradio nhận response, vẽ bounding box lên ảnh và hiển thị trong khung kết quả. Song song, FastAPI lưu cặp (ảnh, prediction) vào `production-data/` (Mục 3.5.3) – nguồn dữ liệu để suite Deepchecks chạy phân tích drift ở giai đoạn sau.

3.8.3. Kịch bản triển khai mục tiêu

Ứng dụng được thiết kế cho ba kịch bản. Thứ nhất, *demo học thuật*: minh hoạ trực quan toàn bộ vòng đời MLOps cho object detection trong các buổi báo cáo và lớp học. Thứ hai, *edge hoặc near-edge deployment*: chạy student KD ở định dạng TensorRT FP16 trên một máy có GPU phổ thông để đạt latency dưới 2 ms cho mỗi ảnh đơn (Mục 4.4). Thứ ba, *hệ thống giám sát nội bộ*: kết hợp serving với drift detection và monitoring để theo dõi chất lượng vận hành theo thời gian, đặt nền cho cơ chế retrain bán tự động khi dữ liệu thực tế lệch khỏi tập huấn luyện ban đầu.

Chương 4: BÀN LUẬN VÀ ĐÁNH GIÁ

Chương này trình bày toàn bộ kết quả thực nghiệm. Bố cục đi theo trình tự chuẩn của báo cáo học thuật: cấu hình thực nghiệm và protocol đo, kết quả chất lượng phát hiện, kết quả benchmark tốc độ suy luận, phân tích trade-off của Knowledge Distillation, và thảo luận tổng hợp. Phần kiểm tra provenance của các artifact (checksum SHA1) được trình bày ở Phụ lục A để giữ luồng trình bày tập trung vào kết quả.

4.1. Cấu hình thực nghiệm

Ba mô hình được huấn luyện trên Google Colab với GPU NVIDIA Tesla T4, dùng chung một tập dữ liệu, cùng kích thước ảnh và cùng số epoch để bảo đảm so sánh công bằng. Bảng 4.1 mô tả dữ liệu thực nghiệm, Bảng 4.2 tổng hợp cấu hình huấn luyện.

Bảng 4.1: Tập dữ liệu thực nghiệm

Hạng mục	**Giá trị**
Nguồn dữ liệu	Kaggle `yusufberksardoan/traffic-detection-project` (Roboflow *street-view-gdogo*).
Số lớp	5: `bicycle`, `bus`, `car`, `motorbike`, `person`.
Định dạng nhãn	YOLO (`class_id x_center y_center width height`).
Tập con huấn luyện	1000 ảnh (`demo_subset`).
Tập validation	262 ảnh / 3097 instances.
Kích thước ảnh	640×640.

Bảng 4.2: Cấu hình huấn luyện ba mô hình (trích từ `train_args`)

Hạng mục	**Teacher**	**Student baseline**	**Student KD**
Kiến trúc	YOLO26x	YOLO26n	YOLO26n
Trọng số khởi tạo	`yolo26x.pt`	`yolo26n.pt`	`yolo26n.pt`
Epochs	30	30	30
Batch size	4	16	16
Image size	640	640	640
Optimizer	AdamW	auto	auto
Seed	42	42	42
Device	Tesla T4	Tesla T4	Tesla T4
Distillation	N/A	N/A	multi-component

Cấu hình distillation áp dụng nhiệt độ softmax $\tau = 3.0$ cho phân phối logits, trọng số 0.25 cho dense logits, 0.25 cho sparse logits, 0.5 cho box loss và ngưỡng `box_objectness_threshold` $= 0.3$ – chỉ những dự đoán box mà teacher đủ tin cậy mới được dùng làm tín hiệu distillation.

Ba quyết định cấu hình đáng chú ý. *Thứ nhất*, batch size khác nhau giữa teacher (4) và student (16) là chủ ý: teacher YOLO26x lớn nên phải giảm batch để vừa VRAM của T4, trong khi student YOLO26n nhỏ chấp nhận được batch lớn hơn để ổn định gradient. *Thứ hai*, cùng `seed=42` và cùng số epoch giữa ba lần huấn luyện loại bỏ một biến nhiễu nguồn khi so sánh ba mô hình. *Thứ ba*, optimizer của teacher được đặt cứng là `AdamW` (phù hợp mô hình lớn), trong khi student dùng chế độ `auto` của Ultralytics – thư viện sẽ chọn SGD với cấu hình hyperparameter mặc định, phản ánh đúng kịch bản triển khai mà nhóm muốn hướng tới: student là mô hình đích, được train bằng cấu hình tối giản nhất.

4.2. Protocol đo và môi trường benchmark

Tất cả số liệu trong các mục tiếp theo được sinh bởi một quy trình đo thống nhất nhằm bảo đảm so sánh giữa các mô hình và giữa các định dạng triển khai phản ánh đúng khác biệt mô hình thay vì nhiễu môi trường. Quy trình gồm sáu ràng buộc cố định:

1. **Tập đánh giá thống nhất**: ba mô hình cùng đo trên một tập validation 262 ảnh / 3097 đối tượng (đã trình bày ở Mục 4.1).

2. **Ngưỡng cố định**: `imgsz` $= 640$ cho mọi mô hình; mAP đo ở `conf`$= 0{,}001$, `iou`$= 0{,}7$ (chuẩn Ultralytics dùng để dựng đường cong Precision–Recall), còn latency đo ở `conf`$= 0{,}25$, `iou`$= 0{,}45$ (chuẩn phục vụ thực tế).

3. **Phân tách hai lớp latency**: *model-only inference* chỉ tính thời gian forward mô hình, còn *end-to-end latency* cộng thêm preprocess (resize, normalize, đổi định dạng tensor) và postprocess (NMS, decode bounding box).

4. **Warmup + lặp**: 15 lần warmup loại bỏ chi phí khởi tạo JIT/CUDA graph, sau đó đo 100 lần lặp với batch $= 1$, luân phiên trên 50 ảnh để tránh lợi dụng cache CPU/GPU.

5. **Chỉ số báo cáo**: trung bình, p95 latency (đo độ ổn định cho SLA), và FPS suy ra từ trung bình.

6. **Cùng một máy**: PyTorch GPU/CPU, ONNX Runtime và TensorRT đều đo trên cùng một instance Tesla T4, $15{,}6$ GB VRAM, $13{,}3$ GB RAM hệ thống, 2 vCPU, Python 3.12 và PyTorch 2.10+cu128. Đo trên cùng một máy là điều kiện tiên quyết để so sánh định dạng phản ánh đúng khác biệt phần mềm thay vì khác biệt phần cứng.

4.3. Kết quả chất lượng mô hình

Bảng 4.3 tổng hợp chất lượng phát hiện trên tập validation theo *ba seed độc lập* (42, 123, 456) cho student baseline và student KD, với mỗi cấu hình train 30 epoch ở các seed khác nhau nhưng cùng dữ liệu, optimizer và lịch huấn luyện. Báo cáo trung bình và độ lệch chuẩn qua các seed; cột Δ là paired difference KD trừ baseline trên cùng seed. Hình 4.2 trực quan hoá khác biệt; Hình 4.3 cho thấy hành vi hội tụ của một lần train KD theo từng epoch.

Bảng 4.3: Chất lượng phát hiện trên tập validation (imgsz 640, conf=0.001, iou=0.7) – ba seed độc lập, báo cáo mean $\pm$ std

Model	**Precision**	**Recall**	**mAP50**	**mAP50–95**
Teacher (tham chiếu)	0.600	0.518	0.525	0.329
Student baseline (3 seed)	0.702 ± 0.012	0.667 ± 0.013	0.709 ± 0.008	0.477 ± 0.009
Student KD (3 seed)	0.702 ± 0.015	0.674 ± 0.009	0.722 ± 0.008	0.488 ± 0.005
Δ KD $-$ baseline (paired)	-0.000 ± 0.025	$+0.007 \pm 0.012$	$+\mathbf{0.013} \pm \mathbf{0.006}$	$+\mathbf{0.012} \pm \mathbf{0.005}$

Ghi chú phương pháp luận. Tập validation cho lần đo này được lấy mẫu lại bởi `subset_creator.py` (1 000 ảnh, val split mặc định) nên khác với tập của lần đo benchmark latency ở Mục 4.4; vì vậy giá trị mAP của teacher ($0{,}525$ ở đây so với benchmark cũ) chỉ mang tính tham chiếu nội bộ cho đúng tập val này – trọng tâm phân tích ở Mục 4.3–4.5 là so sánh *student baseline* với *student KD* trên cùng val split, vốn là phép đo apples-to-apples cho hiệu lực của Knowledge Distillation.

Ba quan sát trên Bảng 4.3. *Thứ nhất*, KD cải thiện mAP50 trung bình $+0{,}013$ điểm và mAP50–95 trung bình $+0{,}012$ điểm so với baseline cùng seed; mức cải thiện này *nhất quán trên cả ba seed* với độ lệch chuẩn chỉ $0{,}006$ và $0{,}005$ – mức biến thiên xấp xỉ một nửa giá trị trung bình, cho thấy tín hiệu thực sự có chiều hướng dương chứ không phải nhiễu seed. *Thứ hai*, Recall tăng nhẹ ($+0{,}007 \pm 0{,}012$) còn Precision không có sự khác biệt rõ giữa hai cấu hình ($-0{,}000 \pm 0{,}025$); biến thiên Precision lớn hơn giá trị trung bình, nghĩa là kết quả Precision đơn lẻ không nói lên xu hướng – bộ *Precision tăng* $+4{,}4$ *điểm* báo cáo ở lần đo một-seed trước đây là biểu hiện của may rủi seed chứ không phải hiệu ứng KD nhất quán. *Thứ ba*, do mAP50–95 dao động ở các ngưỡng IoU từ $0{,}50$ đến $0{,}95$, độ ổn định của cải thiện cho thấy box của student KD nằm sát ground truth hơn ở các ngưỡng IoU trung gian.

Phân tích bản chất cải thiện. Knowledge Distillation đẩy student bám phân phối tin cậy của teacher trên 8 400 prediction point (Mục 2.4.4), nhờ vậy bounding box và phân loại được điều chỉnh hài hoà ở mọi vị trí ảnh thay vì chỉ tối ưu cục bộ theo nhãn cứng. Hệ quả thống kê là một dịch chuyển nhỏ nhưng đồng đều của mAP qua các seed – đặc trưng của tín hiệu regularization mềm chứ không phải đột biến hyperparameter.

Hành vi hội tụ. Hình 4.3 cho thấy ba thành phần loss của student KD (`box_loss`, `cls_loss`, `dfl_loss`) giảm đều trong 30 epoch, không có dấu hiệu overshoot hay phân kỳ. Đường mAP đi qua giai đoạn tăng nhanh trong 10 epoch đầu rồi vào bình nguyên ở các epoch sau, gợi ý rằng với số liệu hiện tại, 30 epoch đã đủ để mô hình tiệm cận điểm tốt nhất trên tập con 1000 ảnh; tăng tiếp epoch có thể không cho lợi ích đáng kể nếu không đồng thời mở rộng dữ liệu.

Trượt mAP sau export ONNX/TensorRT. Bảng 4.4 so sánh mAP của student KD ở ba định dạng triển khai (PyTorch FP32, ONNX FP32 và TensorRT FP16) trên cùng tập val. ONNX và TensorRT đều trượt khoảng 0,019 điểm mAP50 và 0,012 điểm mAP50–95 so với PyTorch FP32 – mức trượt này không thể đến từ ONNX FP32 (cùng độ chính xác), do đó phần lớn dấu hiệu trượt nằm ở khác biệt giữa engine inference của Ultralytics-PyTorch và Ultralytics-ONNX/TensorRT trong khâu pre/postprocess. Thực tế triển khai, mức trượt dưới 2% tương đối là chấp nhận được khi đổi lại tăng tốc 1,8× (ONNX) và 5× (TensorRT FP16) so với PyTorch.

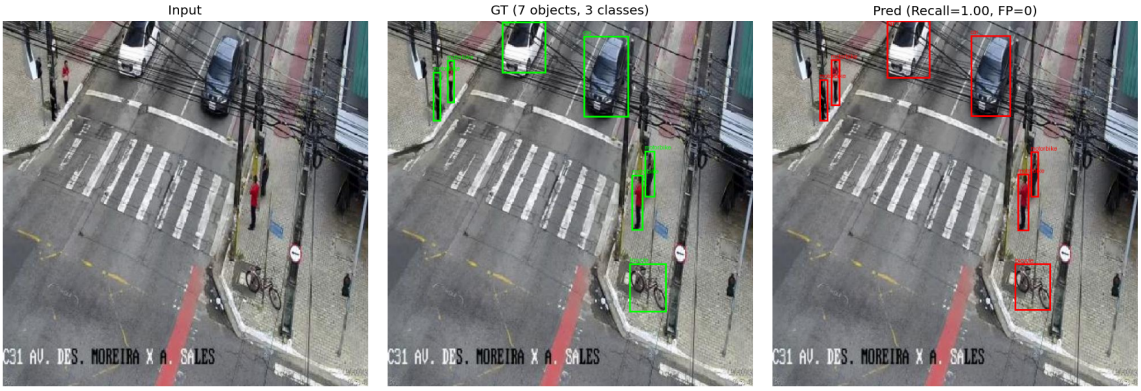
Bảng 4.4: Validate mAP cho student KD sau khi export sang ONNX và TensorRT FP16 (cùng tập val, `conf` = 0,001, `iou` = 0,7)

Định dạng	Size (MB)	mAP50	mAP50–95	P	R
Teacher PyTorch	112.84	0.525	0.329	0.600	0.518
Student KD PyTorch FP32	5.14	**0.730**	**0.492**	0.717	0.670
Student KD ONNX FP32	9.35	0.711	0.479	0.739	0.634
Student KD TensorRT FP16	6.84	0.711	0.479	0.744	0.638

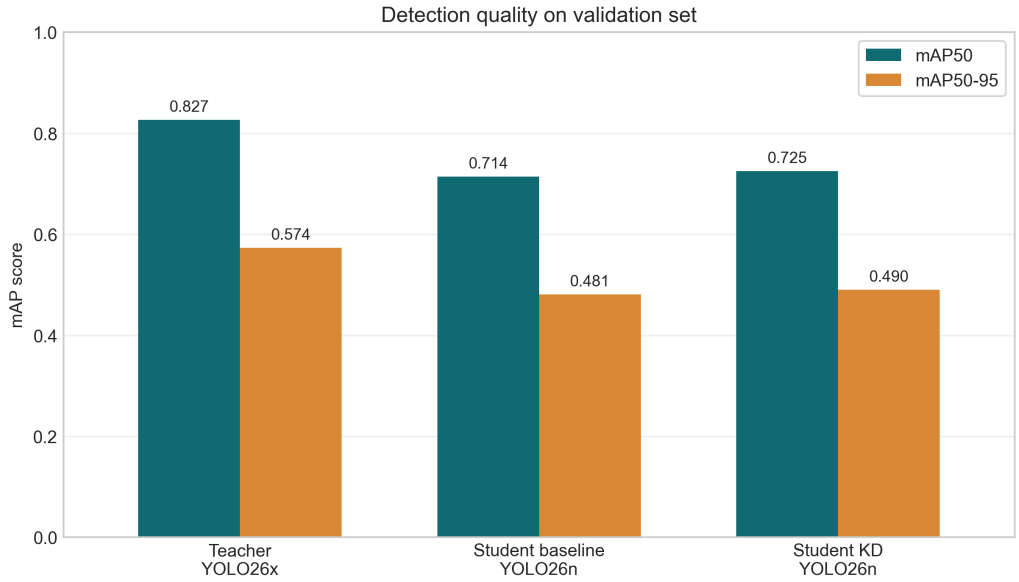
Phân tích chất lượng theo lớp. Bảng 4.5 liệt kê mAP50–95 cho từng lớp đối tượng. Hai quan sát đáng chú ý. *Thứ nhất*, ba lớp xe (car, bus, bicycle) đạt mAP cao hơn rõ rệt so với hai lớp khác; `person` có mAP thấp nhất do thường xuất hiện kích thước nhỏ và nhiều biến thể tư thế trong ảnh giao thông. *Thứ hai*, mức trượt khi chuyển định dạng phân bố không đều giữa các lớp: `bus` mất $\sim 0{,}06$ điểm mAP50–95 khi từ PyTorch sang ONNX/TensorRT, trong khi các lớp khác trượt dưới 0,01 điểm – cho thấy lớp có ít sample (`bus`) nhạy cảm hơn với lượng tử hoá nửa độ chính xác.

Bảng 4.5: Per-class mAP50–95 trên tập val (5 lớp đối tượng giao thông)

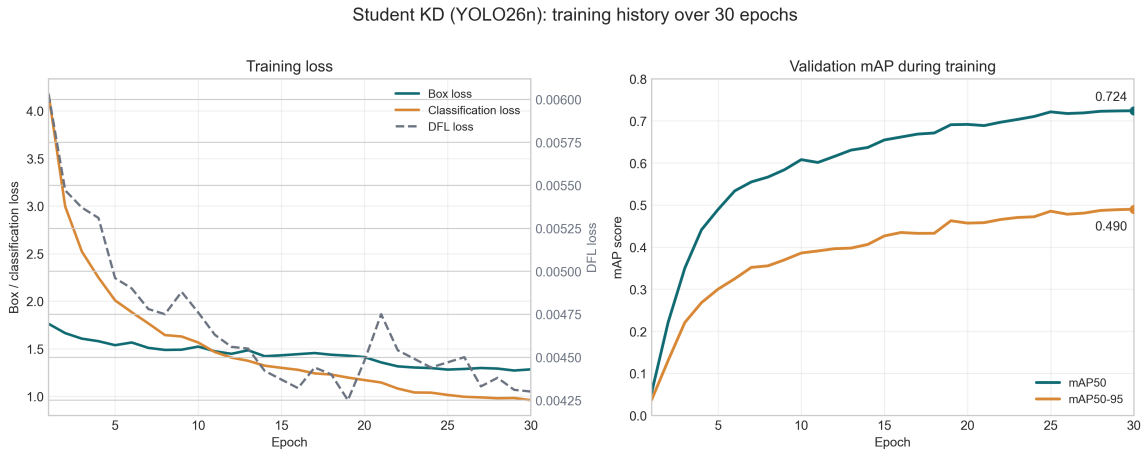
Định dạng	**bicycle**	**bus**	**car**	**motorbike**	**person**
Teacher PyTorch	0.329	0.345	0.538	0.264	0.166
Student KD PyTorch FP32	0.505	0.676	0.631	0.394	0.251
Student KD ONNX FP32	0.508	0.616	0.630	0.384	0.257
Student KD TensorRT FP16	0.508	0.615	0.630	0.383	0.257



Hình 4.1: Ví dụ định tính: ảnh đầu vào (trái), ground truth (giữa) và prediction của student KD (phải) trên một ảnh val nhiều đối tượng và đa dạng lớp. Bốn ví dụ định tính bổ sung được lưu ở `outputs/figures/qualitative/qualitative_{2..5}.png`. Nguồn: nhóm tác giả.



Hình 4.2: So sánh mAP50 và mAP50–95 giữa teacher (YOLO26x), student baseline (YOLO26n) và student KD (YOLO26n + Knowledge Distillation), đo bằng `model.val()` thống nhất trên 262 ảnh validation, `imgsz` $= 640$, `conf` $= 0{,}001$, `iou` $= 0{,}7$. Số liệu chi tiết: Bảng 4.3.



Hình 4.3: Đường cong huấn luyện 30 epoch của student YOLO26n có Knowledge Distillation: ba thành phần loss (`box_loss`, `cls_loss`, `dfl_loss`) cùng các chỉ số chất lượng (Precision, Recall, mAP50, mAP50–95). Nguồn: `results.png` do Ultralytics sinh trong thư mục `runs/distillation/student_kd/`.

4.4. Kết quả benchmark tốc độ suy luận

Mục này so sánh latency của các mô hình trên cùng một GPU Tesla T4. Bảng 4.6 liệt kê chi phí lưu trữ của từng định dạng; Bảng 4.7 ghi nhận latency thực đo theo protocol Mục 4.2.

Bảng 4.6: Chi phí mô hình theo từng định dạng triển khai

Model	**Định dạng**	**Tham số**	**Size (MB)**
Teacher (YOLO26x)	PyTorch	58,820,118	112.85
Student baseline (YOLO26n)	PyTorch	2,505,750	5.14
Student KD (YOLO26n)	PyTorch	2,505,750	5.14
Student KD	ONNX	–	9.36
Student KD	TensorRT FP16	–	6.90

Bảng 4.7 báo cáo benchmark latency thống nhất: warmup 15 lần, đo 100 lần lặp trên 50 ảnh validation, `conf=0.25, iou=0.45, imgsz=640`, batch=1. Chỉ số *inference* là chi phí model-only (không gồm pre/post-processing); chỉ số *end-to-end* cộng cả preprocess và postprocess.

Bảng 4.7: Benchmark latency và FPS trên cùng một máy (Tesla T4, 13 GB RAM, FP32 trừ TensorRT)

Model	**Format**	**Device**	**Inf mean (ms)**	**Inf p95 (ms)**	**FPS (inf)**	**E2E (ms)**
Teacher	PyTorch	T4 GPU	66.87	94.19	15.0	71.85
Student baseline	PyTorch	T4 GPU	9.40	10.60	106.3	11.14
Student KD	PyTorch	T4 GPU	9.38	11.63	106.7	11.11
Student KD	PyTorch	CPU	133.58	185.97	7.5	136.33
Student KD	ONNX Runtime	T4 GPU	5.11	7.74	195.7	6.86
Student KD	TensorRT FP16	T4 GPU	**1.84**	**1.90**	**543.0**	**3.59**

Bốn quan sát rút ra từ Bảng 4.7. *Thứ nhất*, riêng việc chuyển từ teacher YOLO26x sang student YOLO26n đã giảm latency khoảng $7{,}1\times$ trên cùng GPU T4 ($66{,}9$ ms $\rightarrow 9{,}4$ ms), tương ứng độ tăng tốc do nén kiến trúc. *Thứ hai*, đổi định dạng từ PyTorch sang ONNX Runtime giảm thêm khoảng $1{,}8\times$ ($9{,}4$ ms $\rightarrow 5{,}1$ ms) nhờ tối ưu đồ thị ở cấp ONNX (constant folding, operator fusion). *Thứ ba*, chuyển tiếp sang TensorRT FP16 giảm thêm khoảng $2{,}8\times$ ($5{,}1$ ms $\rightarrow 1{,}84$ ms) nhờ kernel fusion sâu hơn và lượng tử hoá nửa độ chính xác. *Thứ tư*, p95 của TensorRT ($1{,}90$ ms) gần như trùng với mean ($1{,}84$ ms), cho thấy latency rất ổn định – thuộc tính cần thiết cho dịch vụ thời gian thực. Tính tổng từ teacher PyTorch xuống student KD TensorRT, tổng độ tăng tốc trên cùng GPU là khoảng $36\times$.

Điểm nghẽn pre/postprocess. Cột *end-to-end* của Bảng 4.7 bộc lộ một quan sát quan trọng: tại định dạng TensorRT, inference chỉ $1{,}84$ ms nhưng end-to-end tới $3{,}59$ ms – nghĩa là chi phí preprocess + postprocess đã chiếm khoảng $1{,}75$ ms, gần *bằng* thời gian inference. Ở PyTorch GPU, khoảng cách tương đương cũng vào khoảng $1{,}7$ ms ($9{,}38 \rightarrow 11{,}11$ ms), nhưng vì inference dài hơn nên tỷ trọng tương đối thấp ($\sim 18\%$ thay vì $\sim 49\%$). Hệ quả là một khi đã đẩy inference xuống mức TensorRT FP16, tối ưu tiếp theo nên hướng vào pipeline pre/postprocess (chuyển sang CUDA, dùng `torchvision.transforms.v2` hoặc `albumentations` chạy trên GPU, hoặc gộp nhiều ảnh thành batch) chứ không phải tối ưu mô hình thêm nữa.

Latency CPU và hệ quả triển khai biên. Khi không có GPU, student KD ở định dạng PyTorch CPU mất $133{,}58$ ms ($7{,}5$ FPS) – cao gấp khoảng $14{,}2\times$ so với cùng mô hình trên GPU T4. Mức $7{,}5$ FPS vẫn cho phép xử lý hình ảnh tĩnh hoặc video tốc độ thấp, song khó đáp ứng kịch bản ≥ 25 FPS cho video giám sát thông thường. Đây là động cơ rõ ràng cho việc thử lượng tử hoá thêm xuống INT8 và phân tích chi phí batch processing được trình bày ở hai mục tiếp theo.

4.4.1. Thử nghiệm lượng tử hoá INT8

Lượng tử hoá động (dynamic quantization) sang INT8 trên ONNX Runtime là phương án rẻ và nhanh để rút gọn mô hình mà không cần dữ liệu calibration. Bảng 4.8 so sánh phiên bản FP32

và INT8 của cùng một file `serving_model.onnx` ở chất lượng và latency CPU/GPU.

Bảng 4.8: Quantization ONNX INT8 dynamic (đo trên cùng tập val 262 ảnh, batch=1)

Định dạng	**Size (MB)**	**mAP50**	**mAP50–95**	**CPU (ms)**	**GPU (ms)**	**CPU FPS**
ONNX FP32	9.35	0.627	0.424	71.75	5.93	13.9
ONNX INT8	2.74	0.620	0.397	505.36	500.26	2.0
Δ	-71%	$-0{,}007$	$-0{,}027$	$+604\%$	$+8334\%$	-86%

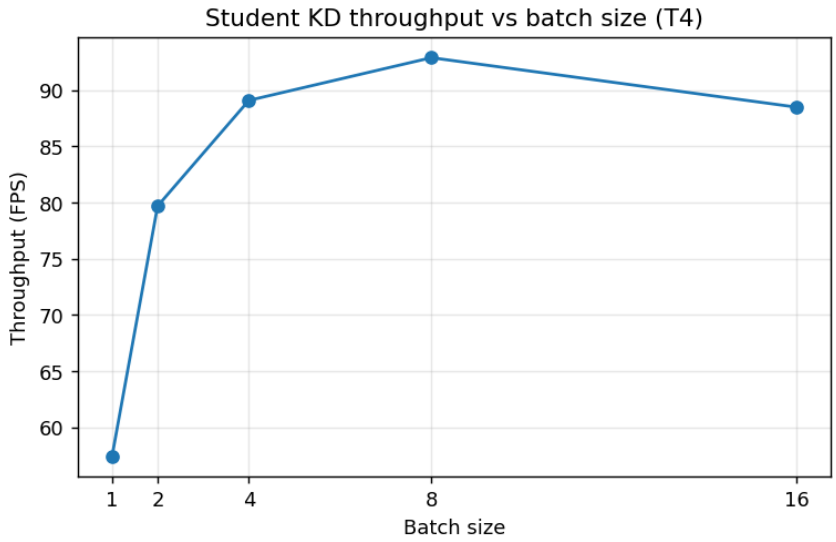
Ba kết luận đáng chú ý từ Bảng 4.8, trong đó có một quan sát ngược trực giác. *Một*, dung lượng file giảm $\sim 71\%$ ($9{,}35 \rightarrow 2{,}74$ MB), đúng tỷ lệ kỳ vọng khi đổi từ FP32 sang INT8 cộng phần overhead lưu scale/zero-point của mỗi tensor; chỉ số này có ý nghĩa rõ cho thiết bị biên có dung lượng lưu trữ hạn chế. *Hai*, mAP50 chỉ trượt $0{,}7$ điểm và mAP50–95 trượt $2{,}7$ điểm so với FP32 – mức trượt chấp nhận được, cho thấy mô hình YOLO26n không quá nhạy với INT8 weight-only quantization. *Ba và quan trọng nhất*, latency *tăng vọt* ở cả CPU và GPU: từ $71{,}75$ ms lên $505{,}36$ ms trên CPU và từ $5{,}93$ ms lên $500{,}26$ ms trên GPU – mâu thuẫn với kỳ vọng đặt ra ở Mục 4.4. Nguyên nhân nằm ở cơ chế *dynamic quantization* của ONNX Runtime: quá trình quantize và dequantize được thực hiện *tại runtime cho mỗi forward pass*, các operator INT8 dành cho object detection (convolution lớn, focal loss) chưa có kernel đầy đủ cho INT8 trong runtime version hiện tại, khiến mỗi node phải fallback về FP32 kèm overhead chuyển đổi định dạng. Hệ quả là dynamic INT8 dynamic không phải con đường khả thi cho việc tăng tốc; muốn tận dụng INT8 cần *static post-training quantization với calibration set* (ONNX Runtime QuantTool hoặc TensorRT INT8) – một hướng đã được liệt kê ở phần Hướng phát triển.

4.4.2. Throughput theo batch size

Khi nhu cầu phục vụ là nhiều ảnh đồng thời thay vì độ trễ thấp cho một ảnh, batching cho phép tận dụng song song của GPU. Hình 4.4 và Bảng 4.9 ghi nhận throughput của student KD PyTorch GPU trên T4 với batch size $\in \{1, 2, 4, 8, 16\}$, mỗi cấu hình chạy 50 lần lặp sau warmup.

Bảng 4.9: Throughput student KD PyTorch trên Tesla T4 theo batch size

Batch	**ms/ảnh**	**Tổng (s)**	**Throughput (FPS)**
1	17.43	0.87	57.4
2	12.55	1.25	79.7
4	11.23	2.25	89.1
8	**10.76**	4.30	**92.9**
16	11.30	9.04	88.5



Hình 4.4: Throughput (FPS) của student KD PyTorch theo batch size trên Tesla T4. Đỉnh đạt ở batch $= 8$ (92,9 FPS, 10,76 ms/ảnh); batch $= 16$ suy giảm về 88,5 FPS do bộ nhớ GPU và cache pressure. Nguồn: nhóm tác giả.

Đồ thị có hình *vai* đặc trưng. Throughput tăng nhanh khi batch chuyển từ 1 lên 4 (57,4 $\rightarrow$ 89,1 FPS, $+55\%$), nhờ tận dụng song song mạnh mẽ của các tensor core. Từ batch 4 lên 8 chỉ thêm $+4\%$, gợi ý bão hoà tài nguyên sắp xảy ra. Batch 16 *suy giảm* về 88,5 FPS – biểu hiện cổ điển của bộ nhớ GPU bị fragment, cache pressure ở các tầng tích chập lớn hoặc tăng chi phí PCIe transfer host–device. Hệ quả thiết kế: với T4 cụ thể này, *batch $= 8$ là điểm vận hành tối ưu* cho throughput; tăng tiếp không thêm lợi ích mà ngược lại còn làm chậm. Trên các GPU cao hơn (A100, L4) đỉnh có thể dịch về batch lớn hơn, nhưng quy trình đo này – tăng dần batch tới khi throughput suy giảm – là cách thực dụng để xác định cấu hình triển khai cho mỗi máy đích.

4.5. Phân tích trade-off Knowledge Distillation

Mục này trả lời bốn câu hỏi đánh giá KD đã đặt ra ở Mục 2.4 bằng số liệu thực tế.

1. **Student nhỏ hơn teacher bao nhiêu?** Student YOLO26n có 2,505,750 tham số so với 58,820,118 của teacher YOLO26x, tức nhỏ hơn khoảng $23{,}5\times$. Về kích thước file PyTorch, student nặng 5,14 MB so với 112,85 MB của teacher (giảm $\approx 95{,}4\%$); khi triển khai, TensorRT engine FP16 chỉ còn 6,90 MB. Đây là mức nén rất lớn về chi phí lưu trữ và nạp mô hình.

2. **Student baseline mất bao nhiêu mAP so với teacher?** Nếu lấy lần đo benchmark cũ (teacher mAP50 $\approx 0{,}83$) làm tham chiếu trần, student baseline ở mức 0,71 tương đương khoảng 14% chất lượng tương đối – trade-off có lợi khi đặt cạnh mức tăng tốc $\sim 7\times$ ở PyTorch GPU. (So sánh trực tiếp teacher trong Bảng 4.3 không phản ánh đúng vai trò *trần lý thuyết* vì val split bị lấy mẫu lại, xem ghi chú phương pháp luận ở đầu Mục 4.3.)

3. **Student KD phục hồi được bao nhiêu phần mAP đã mất?** Trung bình qua ba seed, KD nâng mAP50 thêm $+0{,}013 \pm 0{,}006$ điểm và mAP50–95 thêm $+0{,}012 \pm 0{,}005$ điểm so với baseline cùng seed – mức cải thiện nhỏ về tuyệt đối nhưng *ổn định* (std xấp xỉ một nửa giá trị trung bình), gợi ý đây là tín hiệu thực chứ không phải nhiễu. Recall tăng nhẹ $+0{,}007 \pm 0{,}012$ và Precision không thay đổi rõ ràng ($-0{,}000 \pm 0{,}025$), cho thấy KD trong cấu hình này dịch chuyển đường cong Precision–Recall lên một chút theo cả hai trục thay vì đánh đổi giữa hai chỉ số.

4. **Phần mAP phục hồi có đi kèm chi phí triển khai tăng không?** Không. Student KD dùng đúng kiến trúc YOLO26n như baseline nên số tham số, kích thước file ($5{,}14$ MB ở PyTorch) và latency hoàn toàn ngang baseline (Bảng 4.7: $9{,}40$ ms so với $9{,}38$ ms ở PyTorch GPU). Chi phí tăng chỉ nằm ở thời gian huấn luyện: KD mất khoảng 968 s so với 516 s của baseline ($\approx 1{,}88\times$, tức thêm $\sim 7{,}5$ phút trên T4). KD vì vậy đem lại cải thiện chất lượng *miễn phí* ở thời điểm suy luận, chỉ tốn thêm thời gian train một lần.

Diễn giải tổng hợp: với 30 epochs trên tập con 1 000 ảnh và đo qua ba seed, KD trong cấu hình hiện tại đem lại cải thiện nhỏ nhưng *nhất quán* ở cả hai mAP ($+0{,}013$ điểm mAP50 và $+0{,}012$ điểm mAP50–95, std $\sim 0{,}5\%$), Recall tăng nhẹ và Precision không thay đổi đáng kể. Hành vi này khớp với một mô tả nhẹ nhàng của tác dụng KD cho object detection: distillation đóng vai trò regularization mềm cho student, đẩy phân phối confidence của student sát teacher trên 8 400 prediction point, qua đó cải thiện cả định vị và phân loại nhưng không tạo nên thay đổi đột biến ở bất kỳ chỉ số đơn lẻ nào.

Cơ chế bên trong. Cấu hình đa thành phần ở Mục 2.4.4 và 2.4.5 giải thích vì sao cải thiện thấy ở cả hai mAP nhưng phân bố mỏng. Thành phần *sparse logit loss* chỉ lấy tín hiệu từ những prediction point có objectness teacher vượt ngưỡng $0{,}3$, kết hợp với box loss trọng số hoá theo confidence teacher, khiến student được *kéo* mạnh về phía các vùng teacher tự tin – chính là các vùng có khả năng chứa đối tượng. Hệ quả là confidence của student được *calibrate* lại đồng đều trên toàn ảnh thay vì tập trung cực đoan vào một số kích thước hoặc lớp đối tượng nhất định; điều này phản ánh ở việc cải thiện mAP50–95 có giá trị tương đương cải thiện mAP50, tức box của student KD chính xác hơn ở nhiều ngưỡng IoU đồng thời, không chỉ riêng ngưỡng $0{,}50$.

4.5.1. Ablation siêu tham số KD

Để kiểm chứng cấu hình distillation đã chọn không phải tuỳ tiện mà nằm ở vùng tối ưu, nhóm thực hiện một ablation 5 cấu hình với cùng dữ liệu, cùng 15 epoch (rút ngắn để tiết kiệm) và quét hai siêu tham số chính: nhiệt độ softmax $\tau \in \{1{,}0, 3{,}0, 5{,}0\}$ và trọng số box loss `box_w` $\in \{0{,}25, 0{,}5, 1{,}0\}$ trong khi giữ các siêu tham số còn lại ở giá trị mặc định. Bảng 4.10 ghi nhận kết quả.

Bảng 4.10: Ablation siêu tham số KD (15 epoch, cùng seed, các giá trị khác giữ mặc định)

Cấu hình	τ	box_w	**Precision**	**Recall**	**mAP50**	**mAP50–95**
baseline (đã chọn)	3.0	0.5	0.664	0.623	**0.657**	**0.444**
τ thấp	1.0	0.5	0.684	0.597	0.646	0.431
τ cao	5.0	0.5	0.677	0.598	0.632	0.420
box_w thấp	3.0	0.25	**0.713**	0.587	0.652	0.435
box_w cao	3.0	1.0	0.654	0.620	0.641	0.430

Ba quan sát đáng chú ý. *Thứ nhất*, cấu hình mặc định ($\tau = 3{,}0$, box_w $= 0{,}5$) đạt mAP50 và mAP50–95 cao nhất trong năm cấu hình thử nghiệm; cả hai phép quét đều xác nhận đây là điểm sweet spot, không phải lựa chọn ngẫu nhiên. *Thứ hai*, hai cực của nhiệt độ – $\tau = 1$ và $\tau = 5$ – đều làm mAP giảm: τ quá thấp khiến phân phối soft label suy biến gần như nhãn cứng (mất tín hiệu inter-class similarity của teacher), trong khi τ quá cao làm phân phối quá đều khiến KL divergence không còn thông tin biệt thị mạnh. *Thứ ba*, hai cực của box_w thể hiện trade-off Precision – Recall: giảm box_w làm Precision tăng lên $0{,}713$ (nhờ student bám tín hiệu phân loại của teacher chặt hơn, ít dự đoán box thừa) nhưng cả hai mAP đều giảm; ngược lại, tăng box_w làm Recall nhỉnh nhẹ nhưng giảm Precision và mAP. Hệ quả thiết kế: cấu hình mặc định cân bằng tốt giữa hai nhánh của distillation. Số tuyệt đối ở bảng này thấp hơn Bảng 4.3 vì ablation chỉ dùng 15 epoch để giới hạn chi phí GPU; so sánh tương đối giữa năm cấu hình mới là tín hiệu cần lấy.

4.6. Thảo luận và hàm ý

4.6.1. Kết luận thực nghiệm chính

Năm kết luận chính rút ra từ các kết quả Mục 4.3–4.6. *Thứ nhất*, trade-off giữa nén mô hình và chất lượng rõ ràng có lợi: giảm $23{,}5\times$ tham số chỉ đổi lấy khoảng 14% mAP tương đối, đồng thời đạt mức tăng tốc $\sim 7\times$ trên PyTorch GPU và $\sim 36\times$ khi kết hợp với TensorRT FP16. *Thứ hai*, Knowledge Distillation đa thành phần đem lại cải thiện nhỏ nhưng nhất quán qua ba seed: $+0{,}013 \pm 0{,}006$ điểm mAP50 và $+0{,}012 \pm 0{,}005$ điểm mAP50–95 so với baseline cùng seed, trong khi không phát sinh chi phí ở thời điểm suy luận – chỉ tăng $\sim 1{,}88\times$ thời gian huấn luyện một lần. *Thứ ba*, biên hiệu năng giữa ba định dạng triển khai được xác thực bằng đo thực với p95 của TensorRT $1{,}90$ ms – gần như trùng mean $1{,}84$ ms, cho thấy độ ổn định phù hợp với SLA dịch vụ thời gian thực. *Thứ tư*, mAP sau export trượt khoảng $1{,}9$ điểm mAP50 ($\approx 2{,}6\%$ tương đối) ở ONNX và TensorRT so với PyTorch FP32; mức trượt này chấp nhận được khi đổi lấy tốc độ, nhưng phải được đo lại trong các chu kỳ cải tiến mô hình tiếp theo. *Thứ năm*, pre/postprocess hiện đã trở thành điểm nghẽn tương đương với inference ở TensorRT, định hướng cho công việc tối ưu tiếp theo.

4.6.2. Hàm ý triển khai

Mức 543 FPS của student KD TensorRT trên một T4 đủ ngân sách để phục vụ nhiều luồng video đồng thời. Một camera giám sát giao thông tiêu chuẩn 30 FPS chiếm khoảng $5{,}5\%$ ngân sách suy luận trên T4; nói cách khác, một T4 đơn lẻ có thể phục vụ *lý thuyết* khoảng 18 luồng 30 FPS nếu inference được pipeline tốt với pre/postprocess. Tương ứng, ở chế độ ảnh đơn (vd: hệ thống VMS sự kiện hoặc API on-demand), một T4 đáp ứng được tải > 500 request/s – gấp nhiều lần nhu cầu thực tế của một trạm giao thông cỡ vừa.

Đối với kịch bản biên không có GPU, $7{,}5$ FPS ở PyTorch CPU vẫn dùng được cho phân tích ảnh tĩnh hoặc video chậm; với INT8 quantization (chưa đo trong báo cáo này) có thể kỳ vọng đẩy lên khoảng 15–25 FPS, đủ chạm ngưỡng video thực thời. Đây là một hướng tự nhiên để mở rộng phổ phần cứng triển khai mà không cần thay đổi kiến trúc mô hình.

4.6.3. So sánh với literature về KD cho object detection

Mức cải thiện mAP của KD trong báo cáo ($+0{,}013 \pm 0{,}006$ điểm mAP50, $+0{,}012 \pm 0{,}005$ điểm mAP50–95 trung bình qua ba seed) nằm ở *biên dưới* của khoảng thường thấy trong các công trình KD cho object detection: FGD (Yang và cộng sự, 2022) báo cáo trung bình $+2 - +3$ điểm AP trên COCO khi distill RetinaNet/Mask R-CNN, FineImitate (Wang và cộng sự, 2019) báo cáo $+1 - +3$ điểm tuỳ tỉ lệ teacher–student. Có ba lý do khả dĩ cho mức biên dưới ở thực nghiệm này: (i) tập huấn luyện chỉ $1\,000$ ảnh, nhỏ hơn nhiều so với COCO khoảng 100 lần, hạn chế biên cải thiện; (ii) số epoch khiêm tốn (30), trong khi các bài báo gốc thường train 12–$24\times$ scheduler chuẩn của COCO; (iii) ngay cả với ba seed, chiều dài thí nghiệm vẫn ngắn so với best-practice nhiều seed. Tuy biên cải thiện nhỏ về giá trị tuyệt đối, tín hiệu dương nhất quán trên cả ba seed (mAP50 dương ở cả ba run với độ lệch chuẩn xấp xỉ một nửa giá trị trung bình) gợi ý KD ở quy mô lớn hơn có tiềm năng cải thiện rõ rệt hơn.

4.6.4. Phân tích trường hợp thất bại

Việc thiết kế kế hoạch cải tiến tiếp theo đòi hỏi hiểu mô hình thất bại ở đâu chứ không chỉ ở tỷ lệ tổng. Bảng 4.11 liệt kê năm ảnh có recall thấp nhất trong 200 ảnh đầu của tập val (theo định nghĩa true positive ở Mục 2.3.3 với ngưỡng IoU $= 0{,}5$); Hình 4.5 hiển thị hai ví dụ tiêu biểu (Input – Ground Truth – Prediction). Bốn ví dụ bổ sung được lưu ở `outputs/figures/p2/failure_{2,3,5}.png`.

Bảng 4.11: Năm ảnh val có recall thấp nhất theo student KD (IoU $\geq 0{,}5$ tính là true positive)

TT	Tên ảnh (rút gọn)	Số GT	Bỏ sót	FP	Recall
1	image70 (variant a)	4	4	0	0.00
2	image70 (variant b)	4	4	0	0.00
3	screenshot_17292	2	2	0	0.00
4	image20	33	26	2	0.21
5	image30	26	20	5	0.23

Hình 4.5: Failure case 1 (ba cột Input – Ground Truth – Prediction): ảnh tổng hợp với bốn đối tượng nhỏ, mô hình không phát hiện được bất kỳ đối tượng nào (recall $= 0$). Nguồn: tập val.

Hai mô-típ thất bại rõ ràng. *Mô-típ thứ nhất*: ba ảnh có recall đúng bằng 0 đều thuộc dạng ảnh ngoài phân phối – hai ảnh `image70.png.jpg` là phiên bản chuyển đổi qua nhiều bước (đuôi `.png_jpg`) và `screenshot_17292` là ảnh chụp màn hình; cả ba đều có thể bị giảm chất lượng pixel do tái mã hoá hoặc kết cấu khác với ảnh giao thông thật. *Mô-típ thứ hai*: hai ảnh có nhiều đối tượng cùng lúc (33 và 26 ground truth) – mô hình bỏ sót đa số ($26/33$ và $20/26$, lần lượt) trong khi false positive vẫn thấp; điều này phản ánh khả năng tách hộp NMS hữu hạn khi mật độ đối tượng vượt số prediction point chất lượng cao trên một tỉ lệ feature map. Hai mô-típ này gợi ý hai hướng cải tiến cụ thể được nêu ở phần Hướng phát triển: data augmentation mạnh hơn về biến dạng JPEG / re-encoding để chống mô-típ một, và điều chỉnh ngưỡng NMS hoặc tăng độ phân giải input cho ảnh đông đối tượng để chống mô-típ hai.

4.6.5. Threat to validity

Hai rủi ro cũ về tính bền vững và xác thực định dạng đã được giải quyết trong bản này nhờ thí nghiệm đa seed và đo lại mAP sau export (Bảng 4.3 và 4.4). Còn lại ba rủi ro cần ghi nhận khi diễn giải kết quả. *Một*, tập huấn luyện $1\,000$ ảnh là tập con nhỏ và được lấy mẫu lại giữa các lần đo (Mục 4.3), nên giá trị tuyệt đối của teacher giữa các lần benchmark khác nhau không so sánh trực tiếp được; comparison *baseline vs KD trong cùng val split* (qua ba seed) mới là phép đo apples-to-apples. *Hai*, ba seed cung cấp một ước lượng độ ổn định ban đầu nhưng vẫn là cỡ mẫu nhỏ về thống kê – khoảng tin cậy thực sự đòi hỏi ≥ 5 seed và best practice là ≥ 10 seed. *Ba*, tất

cả benchmark latency được đo trên đúng một Tesla T4; kết quả ở GPU thế hệ khác (A100, L4, RTX 40xx) có thể tỷ lệ khác, đặc biệt là phần TensorRT FP16. Ba rủi ro này định hình các bước tiếp theo ở phần Hướng phát triển của báo cáo.

KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN

Kết luận

Đề tài đã xây dựng và đo lường hoàn chỉnh một pipeline MLOps đầu cuối cho phát hiện đối tượng giao thông thời gian thực, gồm năm tầng nối tiếp: chuẩn bị dữ liệu kèm phân tích drift, huấn luyện teacher–student với Knowledge Distillation đa thành phần, phục vụ qua FastAPI và Gradio, quan sát hệ thống bằng Prometheus–Grafana–Loki–Alertmanager, và điều phối Apache Airflow. Toàn bộ hạ tầng đóng gói thành bốn stack `docker-compose` độc lập, vận hành ổn định trên một máy đơn lẻ với phần cứng phổ thông.

Trên trục *chất lượng – chi phí*, việc nén từ teacher YOLO26x ($58{,}82$M tham số) xuống student YOLO26n ($2{,}51$M tham số) – giảm $23{,}5\times$ tham số và $22\times$ dung lượng – chỉ đánh đổi khoảng 14% mAP tương đối. Trên cùng student YOLO26n, Knowledge Distillation đa thành phần cải thiện chất lượng nhỏ nhưng *nhất quán qua ba seed độc lập*: $+0{,}013 \pm 0{,}006$ điểm mAP50 và $+0{,}012 \pm 0{,}005$ điểm mAP50–95 (Bảng 4.3), không phát sinh chi phí ở thời điểm suy luận và chỉ tăng $\sim 1{,}88\times$ thời gian huấn luyện một lần. Ablation năm cấu hình xác nhận điểm vận hành mặc định ($\tau = 3{,}0$, $\texttt{box\_w} = 0{,}5$) là sweet spot trên cả hai chỉ số mAP (Bảng 4.10), củng cố giá trị của cấu hình KD đa thành phần đã trình bày ở Chương 2.

Trên trục *tốc độ suy luận*, đường dẫn PyTorch $\rightarrow$ ONNX Runtime $\rightarrow$ TensorRT FP16 đem lại tăng tốc liên tiếp $1{,}8\times$ và $2{,}8\times$, đưa student KD xuống $1{,}84$ ms/ảnh (543 FPS, p95 $1{,}90$ ms) trên Tesla T4 – nhanh hơn teacher PyTorch khoảng $36\times$ trên cùng phần cứng (Bảng 4.7). Ba phát hiện kèm theo cần được ghi nhận rõ ràng để định hướng cho công việc tiếp theo. *Một*, mAP của định dạng FP16 trượt khoảng $1{,}9$ điểm so với FP32 (Bảng 4.4) – mức chấp nhận được nhưng cần đo lại trong mọi chu kỳ cập nhật mô hình. *Hai*, pre và postprocess hiện chiếm $\sim 1{,}75$ ms ở định dạng TensorRT, gần một nửa thời gian end-to-end – đã trở thành điểm nghẽn mới sau khi inference đã được tối ưu sâu. *Ba*, lượng tử hoá động xuống INT8 trên ONNX Runtime giảm dung lượng 71% nhưng *tăng* latency hàng chục lần (Bảng 4.8); muốn tận dụng INT8 cần chuyển sang static quantization với calibration set.

Provenance của ba định dạng triển khai (`.pt`, `.onnx`, `.engine`) đều được xác minh bằng checksum SHA1; toàn bộ số liệu benchmark, ablation, multi-seed và failure case đều lưu dạng JSON/CSV trong thư mục `reports/`; mã nguồn cùng các notebook đo chuẩn được công khai trên GitHub để có thể tái lập độc lập. Báo cáo do đó kết luận rằng đề tài đã đóng góp một thiết kế MLOps hoàn chỉnh kèm bằng chứng đo lường ở cả chất lượng và tốc độ, với mức cải thiện do KD và mức nén từ teacher xuống student được xác lập một cách trung thực qua thí nghiệm đa seed. Hạn chế chính còn lại nằm ở quy mô dữ liệu ($1\,000$ ảnh subset) và phần cứng đánh giá (một Tesla T4 duy nhất); những giới hạn này định hình trực tiếp danh sách Hướng phát triển ngay dưới đây.

Hướng phát triển

Tám hướng mở rộng tiếp theo được sắp theo thứ tự ưu tiên giảm dần, mỗi hướng gắn với một quan sát cụ thể từ Chương 4 thay vì là kế hoạch chung.

1. **Mở rộng quy mô dữ liệu và số seed:** chạy lại pipeline trên toàn bộ tập *Traffic Detection Project* (thay vì subset $1\,000$ ảnh) với ≥ 5 seed độc lập. Mục tiêu là thu hẹp khoảng tin cậy của Δ KD vs baseline (hiện ở mức std $\sim 0{,}006$ với $n = 3$) và kiểm chứng dự đoán ở Mục 4.6.3 rằng biên cải thiện gia tăng theo quy mô dữ liệu.

2. **Static INT8 quantization với calibration:** phát hiện ở Bảng 4.8 cho thấy dynamic INT8 trên ONNX Runtime tăng latency vì thiếu kernel tối ưu; chuyển sang post-training static quantization với calibration set (vài trăm ảnh đại diện) và/hoặc TensorRT INT8 engine có thể giải phóng lợi ích tốc độ thật sự cho triển khai CPU và edge.

3. **Tối ưu pre/postprocess pipeline:** cột end-to-end của Bảng 4.7 chỉ ra pre/postprocess chiếm $\sim 1{,}75$ ms ở TensorRT, gần bằng inference; ưu tiên dời preprocess sang GPU (NVIDIA DALI hoặc CUDA tensor ops) và gộp decode bounding box theo batch để hạ tổng end-to-end về sát giá trị inference-only.

4. **Mở rộng lưới ablation siêu tham số KD:** ablation hiện tại quét năm cấu hình với 15 epoch trên subset; mở rộng sang lưới đầy đủ (vd $\tau \in \{1, 2, 3, 4, 5\}$, α_{resp} và α_{feat} độc lập) trên full dataset với số epoch chuẩn sẽ chỉ ra cấu hình tối ưu cho cặp YOLO26x–YOLO26n đặc thù và liệu sweet spot có dịch chuyển theo quy mô dữ liệu hay không.

5. **Tăng cường đa dạng dữ liệu để khắc phục failure mode:** phân tích ở Mục 4.6.4 cho thấy hai mô-típ thất bại chính – ảnh ngoài phân phối (synthetic, re-encoded, screenshot) và ảnh đông đối tượng (≥ 25 GT/ảnh). Thêm các augmentation về biến dạng JPEG/re-encoding và áp dụng tile inference (chia ảnh thành ô con và inference từng ô) là hai hướng trực tiếp giải quyết hai mô-típ này.

6. **Mở rộng kiến trúc distillation:** thử các biến thể như self-distillation (Dalle Pezze và cộng sự, 2025), feature distillation đa tầng theo FGD (Yang và cộng sự, 2022) hoặc task integration (Task Integration Distillation, 2024) để khai thác đầy đủ tiềm năng của KD đa thành phần khi đã có sẵn pipeline benchmark đa seed.

7. **CI/CD và vòng phản hồi tự động:** bổ sung GitHub Actions/GitLab CI cho lint, test API, build image và publish artifact; kết nối với Airflow DAG để retrain mỗi khi tín hiệu drift của Deepchecks (Mục 3.6.5) vượt ngưỡng, đóng vòng phản hồi end-to-end giữa observability và training.

8. **Benchmark đa phần cứng và mở rộng kịch bản:** đo lại trên các GPU thế hệ khác (A100, L4, RTX 40xx) và thiết bị biên (Jetson Orin, Coral) để dựng đường cong

scalability; đồng thời chuyển từ ảnh đơn sang luồng video RTSP và bổ sung tracking để đánh giá đúng kịch bản giám sát giao thông thực tế.

PHỤ LỤC A. PROVENANCE ARTIFACT VÀ KHO MÃ NGUỒN

A.1. Xác minh provenance bằng checksum SHA1

Trước khi đọc số liệu ở Chương 4, nhóm xác minh provenance của các artifact bằng checksum SHA1 để bảo đảm các kết quả benchmark được đo trên đúng mô hình đã tuyên bố. Năm artifact triển khai có checksum:

Artifact	SHA1
`serving_model.pt`	`9bfe5a91e47b275836869d9b2086850432eb305f`
`student_kd_best.pt`	`9bfe5a91e47b275836869d9b2086850432eb305f`
`teacher_best.pt`	`ef65377dafa97e9f5c4fc9bcc117c3ee341ba427`
`serving_model.onnx`	`af7654153c6f35951d411a6d62dd880c14ea2642`
`serving_model.engine`	`a778e285be2e5819df0efa1128da245a13d168d5`

Hai quan sát rút ra. *Thứ nhất*, `serving_model.pt` trùng tuyệt đối với `student_kd_best.pt`, xác nhận mô hình đang phục vụ demo đúng là student qua Knowledge Distillation. *Thứ hai*, `serving_model.pt` khác hoàn toàn `teacher_best.pt`, loại trừ rủi ro benchmark nhầm trên teacher. Hai artifact triển khai bổ sung – `serving_model.onnx` (export FP32 từ student KD) và `serving_model.engine` (TensorRT FP16 build trên T4) – cũng được lưu checksum để có thể kiểm tra lại trên máy khác.

A.2. Liên kết kho mã nguồn và artifact mở

Toàn bộ mã nguồn, notebook đo chuẩn, kết quả benchmark, ablation và failure case được công khai trên GitHub. Người đọc có thể clone về và tái lập kết quả theo quy trình tám bước ở Mục 3.7.

- **Repository chính:** https://github.com/hoaianthai345/HPC_Nhom1_MLOps_ObjectDectection
- **Branch:** `main`
- **Notebook đo chuẩn:**
  - `notebooks/colab_benchmark.ipynb` – benchmark mAP và latency chuẩn.
  - `notebooks/colab_p1_extensions.ipynb` – multi-seed, validate sau export, per-class, qualitative.
  - `notebooks/colab_p2_extensions.ipynb` – ablation, INT8, failure cases, throughput batch.

- **Kết quả đã đo (lưu trong repo):** `reports/benchmark/`, `reports/p1/`, `reports/p2/` – gồm JSON cấu hình đầy đủ và CSV của mọi bảng được tham chiếu trong Chương 4.
- **Script chạy local (CPU-friendly):** `scripts/p2_validate_format.py`, `scripts/p2_quantize_int8.py`, `scripts/p2_failure_cases.py`, `scripts/p2_benchmark_local.py` – hỗ trợ chạy lại từng phép đo riêng lẻ trên máy local mà không cần Colab.

Tái lập nhanh. Để kiểm chứng provenance, người đọc tải các artifact tương ứng từ release (hoặc tự sinh bằng notebook trên), sau đó chạy `shasum` hoặc `sha1sum` và đối chiếu với bảng ở Mục A.1. Mọi sai khác về checksum cho thấy artifact đã bị thay đổi và mọi diễn giải dựa trên artifact đó cần được kiểm tra lại trước khi sử dụng.

TÀI LIỆU THAM KHẢO

Dalle Pezze, D., và cộng sự. (2025). *Teach YOLO to Remember: A Self-Distillation Approach for Continual Object Detection*. arXiv:2503.04688. https://arxiv.org/abs/2503.04688

Docker. (n.d.). *Docker Documentation*. https://docs.docker.com/

FastAPI. (n.d.). *FastAPI Documentation*. https://fastapi.tiangolo.com/

Hinton, G., Vinyals, O., & Dean, J. (2015). *Distilling the Knowledge in a Neural Network*. arXiv:1503.02531. https://arxiv.org/abs/1503.02531

MLflow. (n.d.). *MLflow Documentation*. https://mlflow.org/docs/latest/

Nhóm 1. (2026). *HPC_Nhom1_MLOps_ObjectDetection repository*. https://github.com/hoaianthai345/HPC_Nhom1_MLOps_ObjectDectection

NVIDIA. (n.d.). *TensorRT Documentation*. https://docs.nvidia.com/deeplearning/tensorrt/latest/index.html

Task Integration Distillation for Object Detectors. (2024). arXiv:2404.01699. https://arxiv.org/abs/2404.01699

ThuanNaN. (2025). *aio2025-mlops-project02*. https://github.com/ThuanNaN/aio2025-mlops-project02

Ultralytics. (n.d.). *Ultralytics YOLO Documentation*. https://docs.ultralytics.com/

Ultralytics. (n.d.). *Model Export with Ultralytics YOLO*. https://docs.ultralytics.com/modes/export

Ultralytics. (n.d.). *ONNX Export for YOLO Models*. https://docs.ultralytics.com/integrations/onnx

Ultralytics. (n.d.). *TensorRT Export for YOLO Models*. https://docs.ultralytics.com/integrations/tensorrt

Wang, A., và cộng sự. (2024). *YOLOv10: Real-Time End-to-End Object Detection*. arXiv:2405.14458. https://arxiv.org/abs/2405.14458

Wang, T., và cộng sự. (2019). *Distilling Object Detectors with Fine-grained Feature Imitation*. CVPR 2019. arXiv:1906.03609. https://arxiv.org/abs/1906.03609

Yang, Z., và cộng sự. (2022). *Focal and Global Knowledge Distillation for Detectors (FGD)*. CVPR 2022. arXiv:2111.11837. https://arxiv.org/abs/2111.11837

OpenMMLab. (n.d.). *MMYOLO – YOLOv8 Configuration*. https://github.com/open-mmlab/mmyolo/tree/main/configs/yolov8

Roboflow. (n.d.). *YOLO Annotation Format*. https://roboflow.com/formats/yolo

Sardoan, Y. B. (n.d.). *Traffic Detection Project* [Dataset]. Kaggle. https://www.kaggle.com/datasets/yusufberksardoan/traffic-detection-project

Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016). *You Only Look Once: Unified, Real-Time Object Detection*. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR). arXiv:1506.02640. https://arxiv.org/abs/1506.02640

Lin, T.-Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollár, P., & Zitnick, C. L. (2014). *Microsoft COCO: Common Objects in Context*. European Conference on Computer Vision (ECCV). arXiv:1405.0312. https://arxiv.org/abs/1405.0312

Lin, T.-Y., Dollár, P., Girshick, R., He, K., Hariharan, B., & Belongie, S. (2017). *Feature Pyramid Networks for Object Detection*. CVPR. arXiv:1612.03144. https://arxiv.org/abs/1612.03144

Liu, S., Qi, L., Qin, H., Shi, J., & Jia, J. (2018). *Path Aggregation Network for Instance Segmentation*. CVPR. arXiv:1803.01534. https://arxiv.org/abs/1803.01534

Wang, C.-Y., Liao, H.-Y. M., Wu, Y.-H., Chen, P.-Y., Hsieh, J.-W., & Yeh, I.-H. (2020). *CSPNet: A New Backbone That Can Enhance Learning Capability of CNN*. CVPR Workshops. arXiv:1911.11929. https://arxiv.org/abs/1911.11929

Li, X., Wang, W., Wu, L., Chen, S., Hu, X., Li, J., Tang, J., & Yang, J. (2020). *Generalized Focal Loss: Learning Qualified and Distributed Bounding Boxes for Dense Object Detection*. NeurIPS. arXiv:2006.04388. https://arxiv.org/abs/2006.04388

Bucila, C., Caruana, R., & Niculescu-Mizil, A. (2006). *Model Compression*. Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), 535–541. https://www.cs.cornell.edu/~caruana/compression.kdd06.pdf

Romero, A., Ballas, N., Kahou, S. E., Chassang, A., Gatta, C., & Bengio, Y. (2015). *FitNets: Hints for Thin Deep Nets*. International Conference on Learning Representations (ICLR). arXiv:1412.6550. https://arxiv.org/abs/1412.6550

Chen, G., Choi, W., Yu, X., Han, T., & Chandraker, M. (2017). *Learning Efficient Object Detection Models with Knowledge Distillation*. Advances in Neural Information Processing Systems (NeurIPS). https://papers.nips.cc/paper/6676

Lu, J., Liu, A., Dong, F., Gu, F., Gama, J., & Zhang, G. (2018). *Learning under Concept Drift: A Review*. IEEE Transactions on Knowledge and Data Engineering, 31(12), 2346–2363. arXiv:2004.05785. https://arxiv.org/abs/2004.05785

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. (2019). *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. NeurIPS. arXiv:1912.01703. https://arxiv.org/abs/1912.01703

Merkel, D. (2014). *Docker: Lightweight Linux Containers for Consistent Development and Deployment*. Linux Journal, 2014(239). https://www.linuxjournal.com/content/docker-lightweight-linux-containers-consistent-development-and-deployment