Intelligent Multi-Model ML Orchestration Platform

Sujith Peddireddy
peddireddy.su@northeastern.edu

Jan Mollet
mollet.j@northeastern.edu

Hemanth Sai Madadapu
madadapu.h@northeastern.edu

Sayee Ashish Aher
aher.sa@northeastern.edu

Sowmyashree Jayaram
jayaram.so@northeastern.edu

January 27, 2026

1 Introduction

1.1 Project Overview

Autonomous robots and drones need real-time perception to navigate safely. The problem? Most current systems use a single heavy model (like YOLOv8-large) for every frame, which introduces 40-50ms of latency no matter how simple or complex the scene is. At speeds of 2-10 m/s, this means the robot is traveling 'blind' for 8-50cm during each inference—creating both safety risks and efficiency problems.

After analyzing existing approaches, our team identified this as a fundamental inefficiency in how autonomous systems handle perception. Our team is building an **intelligent multi-model orchestration platform to tackle this problem** that deploys three YOLOv8 variants simultaneously and uses Reinforcement Learning to route each frame to the optimal model based on real-time complexity analysis. Simple scenes use ultra-fast models (8ms), complex scenes use accurate models (48ms), achieving 62% average latency reduction while maintaining 95%+ accuracy.

Core Innovation: The RL-powered orchestration layer, not the CV models. This architecture generalizes to any ML domain—we demonstrate it with autonomous perception because real-time requirements make the value proposition clear.

1.2 The Problem

- **Current:** Heavy models waste compute on simple scenes (70% of frames)
- **Impact:** 48ms average latency, $200 per 1M inferences
- **Our Solution:** Adaptive routing achieves 18ms average, $115 per 1M
- **At Scale:** 1,000 robots, 30 FPS = $72K monthly savings

1.3 Performance Targets

Metric	Baseline	Our System	Improvement
Avg Latency	48ms	18ms	62% reduction
Throughput	21 FPS	55 FPS	2.6x increase
Accuracy	52.8% mAP	50.2% mAP	95% retention
Cost per 1M	$200	$115	42% reduction

2 Dataset Information

2.1 Dataset: COCO 2017

Description: We're using the COCO 2017 dataset, which is the industry standard for object detection. While it wasn't specifically designed for autonomous systems, it turns out COCO has exactly the classes we need.
Navigation-Relevant Classes:

- **High Priority:** person (262K instances), car, truck, bicycle, traffic light, stop sign
- **Medium Priority:** chair, bench, backpack, potted plant (common obstacles)

Why COCO:

- Contains vehicles, pedestrians, obstacles needed for autonomous navigation
- Diverse indoor/outdoor scenes with varying complexity
- Industry-standard benchmark (reproducible comparisons)
- Free, accessible, well-documented

2.2 Data Card

Attribute	Value
Training Images	118,287
Validation Images	5,000
Object Instances	886,284
Categories	80 (15 navigation-relevant)
Format	JPEG, RGB, 640×640
Person Instances	262,465 (29.6%)
Vehicle Instances	60,000 (6.8%)
Scene Complexity	35% simple, 45% moderate, 20% complex

2.3 Data Sources

- **URL:** `https://cocodataset.org/#download`
- **Training:** `train2017.zip` (18GB)
- **Validation:** `val2017.zip` (1GB)
- **Annotations:** `annotations_trainval2017.zip` (241MB)

2.4 License & Privacy

License: Creative Commons Attribution 4.0 (CC BY 4.0)

- Commercial use permitted
- Modification allowed
- Attribution required

Citation: Lin et al., "Microsoft COCO: Common Objects in Context", ECCV 2014

Privacy:

- Images from Flickr with CC licenses (no PII)
- Storage: Google Cloud Storage (encrypted, team-access only)
- Academic use only
- Retention: Course duration + 90 days

3 Data Planning and Splits

3.1 Preprocessing Pipeline

Pipeline Stages:

1. Download COCO dataset (automated script)
2. Validate file integrity (checksums)
3. Resize images to 640×640 (letterbox padding)
4. Normalize: mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]
5. Convert COCO JSON annotations to YOLO format
6. Calculate complexity scores (1-10)
7. Create stratified splits
8. Version control with DVC

3.2 Data Splits

Split	**Images**	**Purpose**	**Stratified**
Train	115K	Model training	Yes (35/45/20)
Validation	5K	Hyperparameter tuning	Yes (35/45/20)
Test	3K	Final evaluation	Yes (35/45/20)
RL Benchmark	2K	RL agent training	Balanced (700/700/600)

3.3 Complexity Stratification

Formula:

$$complexity = 0.4 \times obj\_count + 0.3 \times occlusion + 0.2 \times edge\_density + 0.1 \times lighting \quad (1)$$

Tiers:

- **Simple (1-3):** 35% of data, 1-3 objects, clear scenes → Nano model optimal
- **Moderate (4-7):** 45% of data, 4-10 objects → Small model optimal
- **Complex (8-10):** 20% of data, 10+ objects, occlusion → Large model required

3.4 RL Benchmarking Dataset

Purpose: Train RL agent with ground-truth routing decisions
Creation:

1. Select 2,000 diverse images
2. Run all 3 models (nano, small, large)
3. Calculate reward: $R = 0.5 \times acc - 0.3 \times lat - 0.2 \times cost$
4. Label with optimal model
5. Split: 1,600 train, 400 validation

4 GitHub Repository

Link: `https://github.com/hemanth1403/IE7374-MLOps-Adaptive-ML-Inference.git`

5 Project Scope

5.1 Scope of Implementation

The scope of this project encompasses the development of a production-ready orchestration layer and the supporting MLOps infrastructure. The following components are explicitly within scope:

- **Multi-Model Deployment:** Containerized deployment of three YOLOv8 variants (Nano, Small, Large) as independent inference services.
- **Intelligent Router:** A Reinforcement Learning (RL) agent based on the PPO (Proximal Policy Optimization) algorithm to perform real-time model selection.
- **Feature Extraction Pipeline:** A lightweight preprocessing module to extract scene complexity metrics (e.g., object density, luminosity, and motion vectors) to serve as the RL state space.
- **MLOps Pipeline:** Full CI/CD integration, model versioning via a Model Registry, and experiment tracking using tools like MLflow.
- **Monitoring Observability:** Real-time dashboards tracking routing accuracy, system latency, and cloud cost metrics.

5.2 Project Constraints and Out-of-Scope

To ensure project feasibility within the academic timeline, the following are considered out-of-scope:

- **Custom Model Architecture:** We will use pre-trained YOLOv8 weights; we are not inventing new computer vision architectures.

- **Hardware Acceleration:** Optimization for specific edge hardware (like NVIDIA Jetson or TPU) is not required; the system will be evaluated on standardized cloud GPU instances.
- **Live Data Collection:** The system will be validated using the COCO 2017 dataset rather than live, real-world drone telemetry.

5.3 Business Value Proposition

By implementing an intelligent routing layer, organizations can achieve:

1. **Scalability:** Support for 3x more concurrent streams on the same hardware budget.
2. **Adaptability:** The ability to shift the "Reward Function" weights (Latency vs. Accuracy) in real-time without redeploying code.
3. **Safety:** Faster reaction times for autonomous systems by reducing the "blind travel" distance between frames.

6 Current Approach Flow Chart and Bottleneck Detection

6.1 Traditional Single-Model Approach

In most production systems, a single heavy model is deployed and every request is passed through it, regardless of how simple or complex the input is. This leads to unnecessary compute usage and higher latency for a large portion of traffic.

Image Input

↓

Preprocessing (Resize + Normalize)

↓

Heavy Model Inference
YOLOv8-large
48ms $200/1M

↓

Post-Processing

↓

Response Output

Figure 1: Text-based baseline pipeline using a single heavy model (YOLOv8-large) for all inputs.

6.2 Our Proposed Multi-Model Approach

Our approach introduces an intelligent routing layer that evaluates each incoming request and chooses among multiple model variants (nano, small, large) in real time. The goal is to maintain near baseline accuracy while significantly reducing latency and cost.

$$\begin{tikzcd}
 & \textbf{Camera Feed} \arrow[d] & \\
 & \textbf{Preprocessing} \arrow[d] & \\
 & \textbf{Feature Extraction} \arrow[d] & \\
 & \textbf{RL Agent (Routing Decision)} & \\
\textbf{Simple} \arrow[d] & \textbf{Moderate} \arrow[d] & \textbf{Complex} \arrow[d] \\
\textbf{Nano Model (8)} & \textbf{Small Model (15)} \arrow[d] & \textbf{Large Model (48)} \\
 & \textbf{Result} \arrow[d] & \\
 & \textbf{Navigation / Output} &
\end{tikzcd}$$

Figure 2: Text-based flowchart for RL-based model selection and inference pipeline.

7 Metrics, Objectives, and Business Goals

7.1 Key Performance Metrics

We measure success across three dimensions: inference performance, RL routing quality, and cost efficiency. All metrics are tracked continuously via dashboards.

Metric	**Target**	**Baseline**	**Our System**	**Measurement**
Latency (p95)	< 80ms	48ms	20ms (avg)	End-to-end request time
Throughput (FPS)	≥ 55	21	55	Requests/sec (converted to FPS)
Accuracy (mAP@0.5)	$\geq 95\%$ of baseline	52.8%	50.2%	Standard benchmark evaluation
Cost / 1M inferences	$\leq \$120$	\$200	\$115	Cloud pricing $\times$ measured utilization

Table 1: Inference performance targets vs. baseline and our system.

Metric	**Target**	**Alert Threshold**
GPU Utilization	75–85% (efficient)	<30% (underused) or >95% (bottleneck)
CPU Utilization	70–80%	>90% sustained
Error Rate	<1%	>2% triggers alert
Queue Depth	<10 sustained	>20 triggers scaling

Table 2: Infrastructure monitoring targets and alert thresholds.

Metric	Target	Purpose
Routing Accuracy	$\geq$90%	% of model selections that match ground-truth optimal choice
RL Convergence Time	2–3 hours	Time to train RL agent on 2K benchmarking dataset (single GPU)
Decision Confidence	>0.80	RL agent confidence in routing decisions (to detect uncertain cases)

Table 3: RL routing quality and training behavior metrics.

7.2 Business Goals

Goal 1: Enterprise Cost Reduction. **Problem:** Companies waste an estimated 40–60% of inference budget by running heavy models on easy inputs. **Our target** is a 42% cost reduction, decreasing cost from \$200 to \$115 per 1M inferences. At scale, this translates to meaningful savings: at 100M inferences per month, the reduction corresponds to approximately \$85K saved monthly, or \$1.02M annually. **Market relevance:** Large platforms run inference at massive volume, so even small percentage improvements translate into substantial dollar savings. Cloud-native organizations are increasingly cost-conscious and actively optimize compute spend, and the same cost pressure appears across high-scale product and recommendation pipelines.

Goal 2: Improved User Experience via Latency Reduction. **Problem:** Users expect sub-100ms responses, and increased latency can raise bounce rates and reduce engagement. **Our target** is a 58% latency reduction, improving average latency from 48ms to 20ms, while sustaining p95 latency below 80ms. For simple scenes, the system aims to be up to 6× faster by routing to a lighter model tier (8ms vs. 48ms). **Market relevance:** Real-time applications such as autonomous driving, search, and video analytics are highly latency-sensitive, and even small latency increases (e.g., 100ms) can negatively impact engagement in interactive systems.

Goal 3: Operational Efficiency and Scalability. **Problem:** Single-model inference pipelines tend to scale nearly linearly in cost as traffic grows, forcing teams to add hardware to maintain latency SLAs. **Our target** is a 2.6× throughput improvement (21 to 55 FPS) on the same hardware, enabling the platform to handle approximately 3× higher traffic without proportional infrastructure expansion. We also aim to reduce GPU utilization from 95% to 30%, creating headroom for additional workloads and improving cluster efficiency. **Market relevance:** Kubernetes is widely adopted for production serving, but many teams lack robust patterns for operating multi-model inference systems efficiently, especially when coordinating scaling, routing, and resource utilization across tiers.

Goal 4: Production Maturity and Risk Mitigation. **Problem:** Production ML systems can fail silently due to drift, regressions, or hidden data quality issues, and delayed detection can lead to substantial business impact. **Our target** is full model lifecycle governance and observability, including 100% model registry coverage via MLflow, 99.9% monitoring uptime, and incident response within 5 minutes. The operational objective is zero undetected accuracy drops by combining metric tracking, drift indicators, and alerting tied to defined thresholds. **Market relevance:** As ML deployments mature, organizations increasingly require auditability, monitoring, and rapid rollback capability to reduce operational risk and maintain reliable model performance over time.

8 Failure Analysis

This project introduces an RL-based routing policy that selects among multiple object-detection models (YOLOv8 nano/small/large) to balance latency, cost, and accuracy. Because routing decisions and model performance interact, we anticipate failures across (i) RL policy quality, (ii) model performance and drift, (iii) data/pipeline integrity, and (iv) deployment infrastructure. Below we enumerate key risks, detection signals, and mitigations, including explicit fallback behaviors to preserve service availability.

8.1 Risk Register (Pre-Deployment)

Risk	Impact	Likelihood	Detection Signal	Mitigation / Fallback
RL policy selects suboptimal tier (e.g., too large too often)	Higher latency/cost; SLA risk	Medium	Spike in p95 latency; routing distribution shifts toward large; cost/request increases	Reward shaping; cap large-tier usage temporarily; revert to rule-based routing; retrain with more diverse states
RL policy under-explores or over-explores	Stagnant improvement or instability	Medium	Exploration rate abnormal; reward variance high; oscillating routing decisions	Tune exploration schedule; constrain action entropy; evaluate offline before rollout
Model drift (domain shift)	Accuracy degradation over time	Medium	mAP/precision/recall drop on sampled labels; confidence distribution shift	Scheduled evaluation; drift alarms; retrain affected tier; shadow deploy and A/B validate
Training data corruption / label noise	Poor model quality; inconsistent metrics	Low–Medium	Data validation failures; sudden metric drops between runs	Dataset checksums; schema validation; data versioning; quarantine suspicious batches
Feature extraction failure (complexity features)	RL input invalid; wrong routing	Low	Feature pipeline exceptions; missing/NaN feature rates	Input validation; default feature vector; route to safe default tier (small)
Model service fails to load / crashes	Partial outage; increased errors	Low–Medium	Liveness/readiness failures; increased 5xx; restart loops	Health checks; auto-restart; warm replicas; fallback to available tiers
Resource exhaustion (CPU/GPU memory)	Latency spikes; OOM crashes	Medium	GPU/CPU utilization near 100%; OOM logs; queue depth increases	HPA autoscaling; request queue limits; enforce max concurrency; degrade to smaller tier
Monitoring / telemetry gaps	Blind to issues; delayed response	Low	Missing metrics; dashboard gaps; exporter down	Redundant exporters; alerts on missing telemetry; log sampling + storage backpressure handling
Timeline / integration risk	Missed milestones; incomplete MLOps loop	Medium	Repeated integration blockers; delayed interfaces	Define service contracts early; incremental demos; mock services for integration tests

Table 4: Primary failure modes, detection signals, and mitigations for the orchestration platform.

8.2 Post-Deployment Failure Scenarios (Playbooks)

Scenario 1: RL routing degrades (policy regression).

- **Symptom:** Increased p95 latency and/or cost per request; routing shifts toward larger model tier without accuracy gain.

- **Detect:** Alert on latency/cost thresholds; monitor routing distribution and reward trend.
- **Response:** Immediately switch to a safe fallback policy (rule-based or fixed "small" tier), freeze RL updates, and preserve logs/metrics for root-cause.
- **Recover:** Retrain policy offline using recent trajectories; validate on holdout workload; redeploy behind an A/B flag; gradually ramp traffic.

Scenario 2: Cost spike due to overuse of large model.

- **Symptom:** Cost/request rises above budget; GPU utilization persistently high.
- **Detect:** Budget alert; increased fraction of large-tier selections; sustained GPU pressure.
- **Response:** Impose temporary action cap (max % large); adjust reward to penalize cost; increase autoscaling limits if needed.
- **Recover:** Re-tune reward weights and constraints; re-evaluate decision boundaries; re-enable unconstrained policy after stability.

Scenario 3: Accuracy drop from drift (domain shift).

- **Symptom:** Sampled evaluation metrics drop; confidence scores shift; more false positives/negatives reported.
- **Detect:** Drift monitors on confidence distribution and feature statistics; periodic labeled sampling.
- **Response:** Route more traffic temporarily to the more accurate tier; trigger retraining pipeline for impacted tier(s).
- **Recover:** Validate retrained model in shadow mode; promote via registry; update routing policy if state-action values changed.

9 Deployment Infrastructure

We deploy the platform as containerized microservices. The orchestration service provides an API endpoint, computes lightweight complexity features, queries the RL policy for a routing decision, and forwards requests to the selected model-tier service. Model artifacts are versioned via an experiment and model registry, and the system is deployed locally for development and on Kubernetes for scalable production-like testing.

9.1 Target Platforms

- **Local development:** Docker Compose (single-node) for fast iteration.
- **Kubernetes:** GKE (primary), with portability to EKS / local Minikube.

9.2 Services and Responsibilities

- **API Gateway / Load Balancer:** accepts external requests; routes to orchestrator.
- **Orchestrator (FastAPI):** feature extraction, RL decision, request forwarding, response aggregation, metrics emission.
- **RL Policy Service:** serves policy inference and (optionally) supports periodic policy updates.

- **Model Tier Services:** independent inference services for YOLOv8 nano/small/large.
- **Model Registry / Artifact Store:** stores versioned models and metadata (e.g., MLflow + object storage).
- **Monitoring Stack:** metrics collection and dashboards (e.g., Prometheus + Grafana) with centralized logging.

9.3 Architecture Diagram

User/Client
↓ (HTTP: `POST /detect` with image)
API Gateway / Load Balancer
↓ (routes request)
Orchestrator Service (FastAPI)
↓ (extract complexity features)
RL Policy Service → **Action: {`nano`, `small`, `large`}**
↓ (forward image to chosen tier)
Inference Tier Services (YOLOv8)
`YOLO-nano` | `YOLO-small` | `YOLO-large`
↓ (detections: boxes/classes/confidence)
Orchestrator Service
↓ (response)
User/Client

Internal supporting systems:
Model Registry/Artifacts (e.g., MLflow) → (models pulled by inference services)
Monitoring/Logging (Prometheus/Grafana/Logs) ← (metrics/logs from all services)

Figure 3: Deployment architecture for RL-based multi-model orchestration. Only the API Gateway endpoint is public; all other components are internal microservices.

10 Monitoring Plan

Monitoring is designed to ensure reliability (SLA), cost awareness, and sustained ML quality. We monitor at three layers: (1) infrastructure, (2) application/service, and (3) ML/RL behavior. Dashboards summarize system health, while alerts trigger automated notifications for rapid incident response.

10.1 Monitoring Layers

1. **Infrastructure monitoring:** node/pod CPU, memory, GPU utilization, disk I/O, and network throughput.
2. **Application monitoring:** request rate, error rate, latency (p50/p95/p99), queue depth, and routing distribution.
3. **ML/RL monitoring:** model accuracy proxies, confidence distributions, per-tier inference time, cost per request, and RL policy health signals (reward trend, action entropy, state distribution drift).

10.2 Key Metrics and Alerts

Category	Metrics	Example Threshold	Alert / Action
Infrastructure	CPU%, memory%, GPU%, disk I/O, network throughput	CPU > 85% (5 min), GPU > 90% (5 min)	Scale out replicas; throttle concurrency; reroute to smaller tier
Application	Requests/sec, 5xx rate, p95 latency, queue depth, routing distribution	5xx > 1% (5 min), p95 > SLA	Rollback deploy; failover tier; investigate upstream/downstream
ML quality	Sampled mAP/precision/recall, confidence shift, per-tier inference time	Metric drop > X% week-over-week	Trigger drift investigation; retrain; shadow deploy and validate
RL health	Reward trend, action entropy, exploration rate, state distribution drift	Reward downtrend; entropy collapse/spike	Freeze updates; fallback routing; policy retraining and re-tuning
Cost	Cost/request, GPU hours, large-tier selection fraction	Cost/request > budget	Cap large-tier fraction; adjust reward weights; optimize replicas

Table 5: Representative monitoring metrics, thresholds, and response actions.

10.3 Dashboards (Grafana Panels)

- **System Overview:** traffic, p95 latency, error rate, pod health, restart counts.
- **Model Performance:** per-tier inference time, throughput, and sampled quality metrics.
- **Resource Utilization:** CPU/memory/GPU per service and node; saturation indicators.
- **RL Decision Health:** routing distribution over time; reward trend; entropy/exploration signals.
- **Business/Cost:** cost per request; budget burn; savings vs always-large baseline.

10.4 Logging and Tracing

- **Structured logs (JSON):** include request id, selected tier, latency, confidence summary, and any exceptions.
- **Decision logs:** record RL state features, chosen action, and reward components for offline analysis.
- **Tracing:** propagate request id across orchestrator and model services to pinpoint bottlenecks.

11 Success and Acceptance Criteria

This section defines the objectives to be achieved in each phase of the project and establishes the criteria used to evaluate performance, adherence to best practices, and the overall quality of the deliverable.

11.1 Checkpoint Success Criteria

- **Checkpoint 1: Project Scoping (Jan 27)**

    - Complete scoping document submitted
    - Team roles assigned
    - GitHub repository created with defined folder structure
    - Project approved by instructor

- **Checkpoint 2: Data Pipeline (Feb 24)**
    - COCO dataset downloaded and preprocessed
    - Data splits created and validated
    - All three YOLO models trained and benchmarked
    - Performance metrics documented
    - Data versioning (DVC) operational

- **Checkpoint 3: Model Development (Mar 24)**
    - All three models deployed as services
    - Reinforcement learning agent trained and integrated
    - Orchestration service operational
    - End-to-end inference pipeline functional
    - Basic monitoring in place

- **Checkpoint 4: Deployment (Apr 21)**
    - Complete system deployed using Docker and Kubernetes
    - CI/CD pipeline operational
    - Full monitoring dashboards implemented
    - Performance meets defined targets
    - Project documentation completed

11.2 Performance Acceptance Criteria

Criterion	**Target**	**Measurement**
	CORE IMPLEMENTATIONS	
System Functional	All components working end-to-end	Smoke tests pass
Accuracy	$\geq$ 90% of heavy model MAP	Benchmark on test set
Throughput	$\geq$ 2x vs. baseline	Load testing
Cost Reduction	$\geq$ 30% vs. always heavy	Cost analysis
Latency P95	$<$ 100ms	Performance tests
	ADVANCED IMPLEMENTATIONS	
Accuracy	$\geq$ 95% of heavy model MAP	Benchmark on test set
Throughput	$\geq$ 3x vs. baseline	Load testing
Cost Reduction	$\geq$ 40% vs. always heavy	Cost analysis
Latency P95	$<$ 80ms	Performance tests
RL Routing Accuracy	$\geq$ 90% correct decisions	Ground truth comparison
	FURTHER IMPLEMENTATIONS	
Active Learning	RL identifies uncertain cases	Validation set
Automated Retraining	Continuous improvement loop	Integration test
Multi-Domain Demo	Show generalizability	Proof of concept

Table 6: Project Timeline and Key Deliverables

11.3 Quality Criteria

- **Code Quality**
  - Test coverage: $\geq$ 80% unit test coverage
  - Linting: All code passes `flake8` and `pylint`
  - Documentation: All functions and classes documented
  - Type hints: Python type hints provided for public APIs
- **MLOps Quality**
  - CI/CD: Automated tests executed on every commit
  - Model registry: All models tracked with associated metadata
  - Monitoring: All key system and model metrics collected and visualized
  - Alerts: Critical alerts configured and validated
- **Demo Quality**

- Live demo: Real-time inference operational
- Visualizations: Split-screen views, dashboards, and performance metrics
- Interactivity: System accessible for hands-on testing by Google engineers
- Presentation: Clear explanation of system architecture and value proposition

11.4 Google Expo Success Criteria

- Working live demo (no crashes)
- Clear value proposition explained
- Strong technical depth when questioned
- Production-quality code and infrastructure
- Impressive visualizations and metrics
- Unique insight or innovation that surprises evaluators
- Clear understanding of production tradeoffs and business value

12 Timeline Planning

Phase	**Dates**	**Key Deliverables**
Project Scoping	January 20 – January 27	Scoping document, GitHub repository setup, team formation
Data Pipeline	January 28 – February 24	Exploratory data analysis, model selection and training, benchmarking
Model Development	February 25 – March 24	Reinforcement agent, orchestration service, system integration
Deployment	March 25 – April 20	Full deployment, monitoring, CI/CD pipeline, performance optimization
Google Expo Preparation	April 8 – April 20	Demonstration development, presentation rehearsal
Google Expo Presentation	April 21	Final presentation

Table 7: Project Timeline and Key Deliverables