National College of Ireland

Configuration Manual

MSc Research Project
MSc in Cloud Computing

Min Ko Aung
Student ID: 23340355

School of Computing
National College of Ireland

Supervisor: Prof. Shaguna Gupta

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	Min Ko Aung
Student ID:	23340355
Programme:	MSc in Cloud Computing **Year:** 2024 - 2025
Module:	MSc Research Project
Lecturer:	Prof. Shaguna Gupta
Submission Due Date:	15.09.2025
Project Title:	AI-Driven Auto-Tuning for Serverless Performance Optimization
Word Count:	2645 **Page Count:** 29

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: Min Ko Aung

Date: 15.9.2025

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Min Ko Aung
Student ID: 23340355

Table of Contents

1 Introduction

This manual/paper provides an explanation of the procedure of coding the AI-driven auto-tuning of the Serverless Performance Optimization project. The requirements and system configuration are explained in a manner that allows anyone to reproduce this work and improve it, as part of cloud/serverless performance research. Afterwards, the process of acquiring the Colab notebook and artifacts, installing the dependencies, creating artificial datasets, training the forecasting model (XGBoost), setting the custom Gymnasium environment, and instigating the reinforcement learning agents (A2C and Q learning) is clarified in detail.

2 Environment & Setup

2.1 Hardware Configurations

The experiments are carried out mainly in Google Colab. Training and evaluation can be rerun on CPU and may optionally be accelerated on NVIDIA T4 (15 GB VRAM), where available. For local validation, a Windows 11 machine with an Intel Core i7 (>=3.0 GHz) and 16 GB RAM is required, although similar machines on macOS/Linux are also compatible.

2.2 Software Configurations

- Python 3.10.x (Colab default runtime)
- Google Colab / Jupyter Notebook
- (Optional) Visual Studio Code 1.95+ for local editing

2.3 Prerequisites of Cloud Environment for Training Models

- Google Colab (Free or Pro) and Google Drive mounted for dataset/model persistence.
- Colab runtime: *Runtime* → *Change runtime type* → GPU: T4 (optional), RAM ~15 GB.

2.4 Libraries Requirements

The libraries illustrated below are necessary in data generation, forecasting, environment simulation and RL training.

- numpy==2.0.2
- pandas==2.2.2
- scikit-learn==1.6.1
- xgboost==3.0.3
- gym==0.26.2
- gymnasium==1.2.0
- boto3==1.40.1
- stable-baselines3[extra]==2.7.0
- shimmy>=2.0
- matplotlib==3.10.0
- tensorflow==2.19.0
- keras==3.8.0,
- prophet==1.1.7

Installation (Colab cell used in this project):

```
# Install necessary tools
!pip install numpy pandas scikit-learn xgboost tensorflow keras prophet \
gym gymnasium boto3 stable-baselines3[extra] "shimmy>=2.0"
```

2.5 Colab Runtime Initialization

Starts a Colab session and imports fundamental tools utilized across the chain: pandas to handle tables of data, numpy to perform numerical operations in a vectorized manner, random to sample stochastically, and datetime /timedelta to build on a 5-minute timestamp index and time management. These imports are positioned at the beginning of the notebook so that other parts of the work have the same run time context. To achieve reproducibility, numpy and random seed values are usually assigned as soon as they are imported.

```
import pandas as pd
import numpy as np
import random
from datetime import datetime, timedelta

np.random.seed(42)
```

Note: Stable-Baselines3 on Gymnasium requires shimmy>=2.0 to bridge APIs.

3 Synthetic Data Generation

Generate a time-indexed synthetic workload that resembles realistic serverless behaviour (steady demand with occasional bursts) and generate features required in forecasting and RL.

Configuration:

- Horizon: 30 days, and using 5-min resolution,
- Output file: **synthetic_invocations.csv**

Method:

1. Timestamps: Build a continuous 5-minute index for the entire horizon.
2. Invocations (target series): Generate a predictable bursty pattern: baseline + sinusoid + Gaussian noise, with a 5-step burst every 50 steps (~4.2 hours). Values are lower-bounded at 1.
3. Exogenous settings:
    - Memory (MB): {128 to 3008}
    - Timeout (s): {1 to 30}.
    - Cold starts: Bernoulli with 10% probability.
4. Duration & latency:
    - Duration (ms): base term scaled by load (~100 + inv/3) with small noise.
    - Latency (ms): duration + cold-start penalty (e.g., +150 ms if cold, else +20 ms) + small noise.

Feature engineering:

- Calendar features: hour, minute, second, day-of-week from the timestamp.
- Lagged and rolling stats for invocations: lag1, lag2, rolling mean (5), rolling std (5).
- Drop initial rows with NaNs introduced by lags/rolls; reset index for a clean frame.

Export & reuse:

- Save to synthetic_invocations.csv for downstream steps (forecasting, RL env).
- Print a small preview (head()) to verify schema and ranges.

```python
# Configuration
num_days = 30
interval_minutes = 5
total_points = int((24 * 60 / interval_minutes) * num_days)

# Generate timestamps
start_time = datetime(2024, 1, 1, 0, 0, 0)
timestamps = [start_time + timedelta(minutes=i * interval_minutes) for i in range(total_points)]

# === Predictable Bursty Invocations ===
def generate_bursty_invocations(n, base=100):
    invocations = []
    for i in range(n):
        # Inject burst every 50 steps (approx every 4.2 hrs)
        if i % 50 in range(5):  # 5-step burst duration
            val = base + 120 + np.random.normal(0, 10)
        else:
            val = base + np.sin(i / 10.0) * 25 + np.random.normal(0, 5)
        invocations.append(max(1, int(val)))
    return invocations

invocations = generate_bursty_invocations(total_points)
```

```python
# Memory allocations
memory_choices = [128, 256, 512, 1024, 2048, 3008]
memory_MB = [random.choice(memory_choices) for _ in range(total_points)]

# Timeout (in seconds): AWS Lambda typical range
timeout_sec = [random.choice([1, 3, 5, 10, 15, 30]) for _ in range(total_points)]

# Concurrency estimate (inferred from invocation rate)
concurrency = [max(1, int(inv / 10)) for inv in invocations]

# Cold start (10% probability)
cold_start = [1 if random.random() < 0.1 else 0 for _ in range(total_points)]

# Duration (ms): base duration affected by load
duration_ms = [round(random.gauss(100 + inv / 3, 15), 2) for inv in invocations]

# Latency (ms): base + cold start penalty + noise
latency_ms = [
    round(d + (150 if cs else 20) + np.random.normal(0, 10), 2)
    for d, cs in zip(duration_ms, cold_start)
]
```

```python
# Create DataFrame
df = pd.DataFrame({
    "timestamp": timestamps,
    "invocations": invocations,
    "memory_MB": memory_MB,
    "timeout_sec": timeout_sec,
    "concurrency": concurrency,
    "cold_start": cold_start,
    "latency_ms": latency_ms,
    "duration_ms": duration_ms,
})

# Timestamp feature engineering
df["hour"] = df["timestamp"].dt.hour
df["minute"] = df["timestamp"].dt.minute
df["second"] = df["timestamp"].dt.second
df["dayofweek"] = df["timestamp"].dt.dayofweek

# Lag features
df["invocations_lag1"] = df["invocations"].shift(1)
df["invocations_lag2"] = df["invocations"].shift(2)
df["invocations_avg_5"] = df["invocations"].rolling(window=5).mean()
df["invocations_std_5"] = df["invocations"].rolling(window=5).std()

# Drop initial NaN rows
df = df.dropna().reset_index(drop=True)

# Export
df.to_csv("synthetic_invocations.csv", index=False)
print(df.head())
```

4 Forecasting – XGBoost + GridSearchCV

4.1 Feature Design

Forecasting was performed using XGBoost (Chen & Guestrin; 2016), with features capturing short-term behavior (lags/rolling statistics) and seasonality per day and week (hour, dayofweek, weekend). Drops rows with NaNs created by lag/rolling windows so that training remains clean.

```
# --- Time-based features
df['timestamp'] = pd.to_datetime(df['timestamp'])
df['hour'] = df['timestamp'].dt.hour
df['dayofweek'] = df['timestamp'].dt.dayofweek
df['is_weekend'] = df['dayofweek'].isin([5, 6]).astype(int)

# --- Lag-based features
df["invocations_lag1"] = df["invocations"].shift(1)
df["invocations_lag2"] = df["invocations"].shift(2)
df["invocations_avg_5"] = df["invocations"].rolling(window=5).mean()
df["invocations_std_5"] = df["invocations"].rolling(window=5).std()

# --- Drop NA values from lag features
df = df.dropna().reset_index(drop=True)
```

4.2 Train/Test Strategy

The chronological 80/20 split is used to prevent look-ahead bias. The test window creates a simulation of unobserved and future data.

```
# --- Feature selection
features = [
    'hour', 'dayofweek', 'is_weekend',
    'invocations_lag1', 'invocations_lag2',
    'invocations_avg_5', 'invocations_std_5'
]
target = 'invocations'

X = df[features]
y = df[target]

# --- Time-aware train/test split
train_size = int(len(df) * 0.8)
X_train, X_test = X.iloc[:train_size], X.iloc[train_size:]
y_train, y_test = y.iloc[:train_size], y.iloc[train_size:]
```

4.3 Grid Search parameter space

Hyperparameters were tuned using GridSearchCV from scikit-learn (Pedregosa et al.; 2011), exploring depth, learning rate, estimators, subsampling, and child weight to balance the bias-variance tradeoff and to stabilize short-horizon error (MAE).

```
# --- Grid search parameter space
param_grid = {
    'max_depth': [4, 5, 6],
    'learning_rate': [0.01, 0.05],
    'n_estimators': [100, 200],
    'subsample': [0.8, 0.9],
    'colsample_bytree': [0.8, 1.0],
    'min_child_weight': [1, 5]
}

grid = GridSearchCV(
    estimator=xgb.XGBRegressor(objective='reg:squarederror', random_state=42),
    param_grid=param_grid,
    cv=3,
    scoring='neg_mean_absolute_error',
    verbose=1,
    n_jobs=-1
)
```

4.4 Model fitting & selection

Trains the candidates on the training window and returns the optimal evaluator on out-of-sample data.

```
# --- Fit the grid search
grid.fit(X_train, y_train)

# --- Use the best model
best_model = grid.best_estimator_
print(f"Best Params: {grid.best_params_}")
```

4.5 Evaluation (Test Window)

Saves the selected model to be used again (e.g., to create a forecasting series on the RL environment). And prints out MAE, RMSE, and R^2 on the hold-out period to validate generalization.

```
#Save the best model for re-producitivity.
import joblib

# Save to disk
joblib.dump(best_model, "xgb_best_model.pkl")

# Load it back:
loaded_model = joblib.load("xgb_best_model.pkl")

# Verify it gives the same predictions
y_pred_loaded = loaded_model.predict(X_test)
assert np.allclose(y_pred_loaded, best_model.predict(X_test))

# --- Evaluate on test set
y_pred = loaded_model.predict(X_test)
mae = mean_absolute_error(y_test, y_pred)
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
r2 = r2_score(y_test, y_pred)

print(f"MAE: {mae:.2f}")
print(f"RMSE: {rmse:.2f}")
print(f"R² Score: {r2:.2f}")
```

4.6 Actual vs Predicted plot

The visual verification of the predictions occurs in day/weekly patterns without systematic drifts being too substantial.

```python
# --- Plot actual vs predicted
plt.figure(figsize=(12, 5))
plt.plot(y_test.values, label='Actual')
plt.plot(y_pred, label='Predicted')
plt.title("Actual vs Predicted Invocations")
plt.xlabel("Time Step")
plt.ylabel("Invocations")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

# --- Generate forecast for RL use
forecast_df = df.copy()
forecast_df['forecast'] = best_model.predict(X)
forecast_series = forecast_df['forecast'].tolist()
```

```
MAE: 9.48
RMSE: 19.61
R² Score: 0.76
```

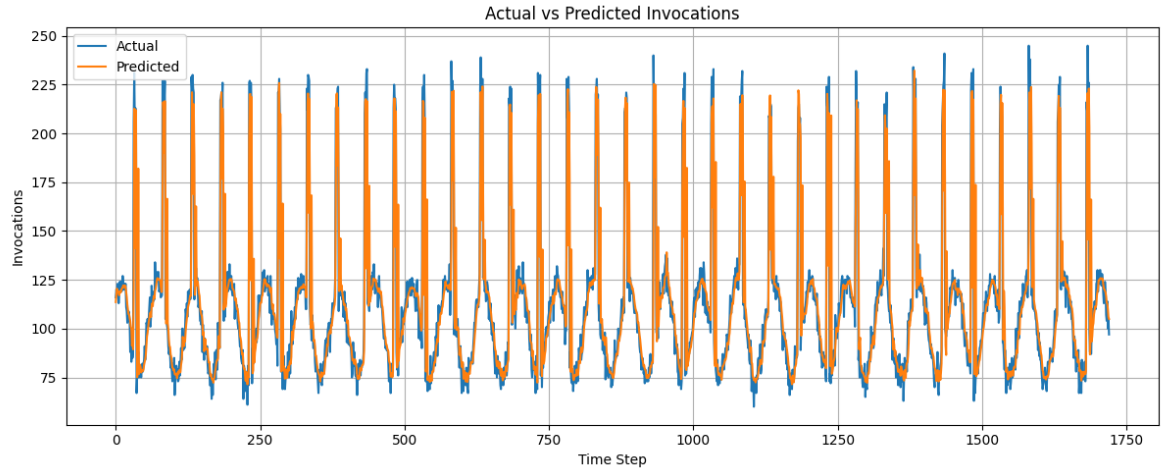


Fig 01 - Actual vs Predicted Invocations (test window)

XGBoost tracks daily/weekly cycles; residual spikes remain at burst peaks. MAE=9.48, RMSE=19.61, R^2=0.76.

5 Custom RL Environment - ServerlessTuningEnv

5.1 Purpose & Module Imports

Deploys a Gymnasium setting (Brockman et al., 2016) that associates a predictive-based stack to serverless config interventions (memory, timeout, concurrency). Every step simulates latency and cost, calculates a reward in an SLA-aware manner, and returns a 3-feature state to the agent.

Loads core dependencies and Gymnasium spaces used by the environment.

```python
import gymnasium as gym
from gymnasium import spaces
```

5.2 Environment Class Declaration and Initialization (__init__)

Defines the Gymnasium setting and initializes the forecast series, step counters, discrete action space (memory, timeout, concurrency), and 3-diminch observation space [normalized_forecast, normalized_latency, normalized_cost].

```python
class ServerlessTuningEnv(gym.Env):
    """
    A custom Gymnasium environment for tuning serverless function configurations.
    """

    def __init__(self, forecast_sequence):
        """
        Initialize the tuning environment.
        """
        super(ServerlessTuningEnv, self).__init__()
        # Forecast series and step counters
        self.forecast_sequence = forecast_sequence
        self.current_step = 0
        self.max_steps = len(self.forecast_sequence)
        self.max_forecast = max(self.forecast_sequence)

        # Discrete action options
        self.memory_options = [128, 256, 512, 1024]
        self.timeout_options = [3, 5, 10]
        self.concurrency_options = [1, 2, 3]

        # Combined action space size
        self.action_space = spaces.Discrete(
            len(self.memory_options) *
            len(self.timeout_options) *
            len(self.concurrency_options)
        )

        # Observation: [normalized_forecast, normalized_latency, normalized_cost]
        self.observation_space = spaces.Box(low=0, high=1, shape=(3,), dtype=np.float32)
        self.state = None
```

5.3 Action Decoding Routine (_decode_action)

Maps the integer action to the (memory, timeout, concurrency) triple used by the simulator.

```python
def _decode_action(self, action):
    """
    Decode a single integer action into memory, timeout, and concurrency settings.
    """
    m_idx = action // (len(self.timeout_options) * len(self.concurrency_options))
    t_idx = (action // len(self.concurrency_options)) % len(self.timeout_options)
    c_idx = action % len(self.concurrency_options)
    return (self.memory_options[m_idx],
            self.timeout_options[t_idx],
            self.concurrency_options[c_idx])
```

5.4 Environment Reset Procedure (reset)

Re-sets the episode to the beginning of the forecast and returns a neutral normalized state.

```python
def reset(self, *, seed=None, options=None):
    """
    Reset the environment state to start a new episode.
    """
    super().reset(seed=seed)
    self.current_step = 0
    # Initialize with neutral midpoints
    self.state = np.array([0.5, 0.5, 0.5], dtype=np.float32)
    return self.state, {}
```

5.5 Step Transition Function (step)

Applies the selected configuration, calculates latency, cost, SLA (>200 ms), and reward (latency-focused with cost balance and efficiency bonus), updates the normalized state, and advances time.

```python
def step(self, action):
    """
    Execute one time-step of the environment based on the agent's action.
    """
    # End episode if forecast exhausted
    if self.current_step >= self.max_steps:
        return self.state, 0.0, True, False, {}

    # Retrieve forecast and normalize
    forecast = self.forecast_sequence[self.current_step]
    normalized_forecast = forecast / self.max_forecast

    # Decode chosen configuration
    memory, timeout, concurrency = self._decode_action(action)

    # === Latency Calculation ===
    # Base latency minus savings from concurrency + added load impact
    latency = max(50, 250 - concurrency * 50 - memory * 0.2 + normalized_forecast * 120)

    # === Cost Calculation ===
    # Simple cost model: cost ∝ memory × timeout × concurrency × invocations
    invocations_per_step = 200
    cost = memory * 0.00001667 * (timeout / 60) * concurrency * invocations_per_step

    # === SLA Violation Check ===
    sla_threshold = 200
    sla_violated = latency > sla_threshold

    # === Reward Function (Latency-Focused, Balanced Cost, Soft SLA Penalty) ===
    latency_penalty = (latency / 200) ** 2
    cost_penalty = cost * (50 if normalized_forecast >= 0.5 else 30)
    sla_penalty = 2 if sla_violated else 0

    reward = - (latency_penalty + cost_penalty + sla_penalty)

    # Efficiency bonus for low-latency, low-cost settings
    if latency < 170 and cost < 0.002:
        reward += 0.4

    # Penalize overprovisioning when load is light
    if normalized_forecast < 0.3:
        if memory > 512:
            reward -= 0.2
        if concurrency > 1:
            reward -= 0.2

    # --- State Update ---
    normalized_latency = min(latency / 300, 1.0)
    normalized_cost = min(cost / 0.1, 1.0)

    self.state = np.array([
        normalized_forecast,
        normalized_latency,
        normalized_cost
    ], dtype=np.float32)

    # Advance to next time step
    self.current_step += 1
    done = self.current_step >= self.max_steps

    # Return observation, reward, and diagnostics
    info = {
        "latency": latency,
        "cost": cost,
        "sla_violated": sla_violated
    }

    return self.state, reward, done, False, info
```

6 A2C Training (Stable-Baseline3)

6.1 Dependencies, Vectorized Environment, and Compliance Check

Initializes Stable-Baselines3 components, wraps the custom environment to allow vectorized training, and checks compliance with Gymnasium API.
(Assumes forecast_series is prepared in the prior section.).

```
from stable_baselines3 import A2C
from stable_baselines3.common.env_checker import check_env
from stable_baselines3.common.vec_env import DummyVecEnv

# Wrap the custom env in a vectorized interface
env = DummyVecEnv([lambda: ServerlessTuningEnv(forecast_series)])

# Ensure the env follows Gym's API
check_env(ServerlessTuningEnv(forecast_series), warn=True)
```

6.2 Agent Configuration

Initialize your A2C agent based on the fixed hyperparameters that are used in short-horizon control on this environment.

```
# Create the A2C agent with fixed hyperparameters
a2c_model = A2C(
    "MlpPolicy",
    env,
    verbose=1,
    learning_rate=0.0007,
    gamma=0.99,
    n_steps=64,
    ent_coef=0.01,
    vf_coef=0.5,
    max_grad_norm=0.5,
    seed=42
)
```

6.3 Training and Model Persistence

Trains the agent over a fixed number of timesteps and saves the resulting policy to be reused during evaluation.

```
# Train for a set number of timesteps
a2c_model.learn(total_timesteps=11000)

# Save the trained model for later use
a2c_model.save("a2c_serverless_model")
print("A2C training completed with dual-penalty reward and saved as 'a2c_serverless_model.zip'")
```

7 Q-Learning Baseline

7.1 Agent Class — Declaration & Initialization (__init__)

Describes a tabular Q-learning agent with ε-greedy exploration and uniform discretization of the 3-dimensional state. Initializes the Q-table to zeros.

```python
# Define Q-Learning Agent
class QLearningAgent:
    """
    Simple Q-learning agent with epsilon-greedy policy and state discretization.
    """

    def __init__(self, env, alpha=0.1, gamma=0.99, epsilon=1.0, epsilon_decay=0.995, epsilon_min=0.05):
        """ Initialize Q-learning parameters and empty Q-table. """
        np.random.seed(42)
        self.env = env
        self.alpha = alpha
        self.gamma = gamma
        self.epsilon = epsilon
        self.epsilon_decay = epsilon_decay
        self.epsilon_min = epsilon_min

        # discretize each of the 3 state features into equal bins
        self.state_bins = 10
        # Q-table: (num_discrete_states x num_actions)
        self.q_table = np.zeros((self.state_bins**3, env.action_space.n))
```

7.2 State Discretization (discretize_state)

Maps the continuous state to a single discrete index using equal-width bins in [0,1].

```python
    def discretize_state(self, state):
        """
        Map continuous state to a single discrete index.
        Splits each feature into bins, then flattens to 1D index.
        """
        bins = np.linspace(0, 1, self.state_bins + 1)
        indices = [min(self.state_bins - 1, max(0, np.digitize(s, bins) - 1)) for s in state]
        return indices[0] * (self.state_bins**2) + indices[1] * self.state_bins + indices[2]
```

7.3 Action Selection (choose_action)

Selects actions via ε-greedy policy: random exploration with probability ε, otherwise greedy exploitation.

```python
    def choose_action(self, state):
        """
        Select an action via ε-greedy:
        - with prob. ε, choose random action (explore)
        - otherwise, pick the best-known action (exploit)
        """
        state_idx = self.discretize_state(state)
        if np.random.rand() < self.epsilon:
            return self.env.action_space.sample()
        return np.argmax(self.q_table[state_idx])
```

7.4 Q-Value Update (update_q_value)

Applies the Q-learning rule using the target r + γ max_a' Q(s', a').

```python
    def update_q_value(self, state, action, reward, next_state):
        """
        Apply the Q-learning update rule:
        Q(s,a) ← Q(s,a) + α [r + γ max_a' Q(s',a') − Q(s,a)]
        """
        state_idx = self.discretize_state(state)
        next_state_idx = self.discretize_state(next_state)
        best_next_action = np.max(self.q_table[next_state_idx])
        td_target = reward + self.gamma * best_next_action
        td_error = td_target - self.q_table[state_idx][action]
        self.q_table[state_idx][action] += self.alpha * td_error
```

7.5 Training Loop (train)

Runs multiple episodes, updates the Q-table, decays ε, and records per-episode total reward.

```
def train(self, episodes=300):
    """
    Run training loop for given episodes.
    Returns a list of total rewards per episode.
    """
    episode_rewards = []
    for ep in range(episodes):
        state, _ = self.env.reset()
        done = False
        total_reward = 0

        while not done:
            action = self.choose_action(state)
            next_state, reward, done, _, info = self.env.step(action)
            self.update_q_value(state, action, reward, next_state)
            state = next_state
            total_reward += reward

        # decay exploration rate
        self.epsilon = max(self.epsilon_min, self.epsilon * self.epsilon_decay)
        episode_rewards.append(total_reward)

    return episode_rewards
```

7.6 Training Loop (train)

Initializing the environment instance, instantiates the agent, and runs training for 300 episodes.

```
#Initialize and train Q-learning agent
env = ServerlessTuningEnv(forecast_sequence=forecast_series)
agent = QLearningAgent(env)
rewards = agent.train(episodes=300)
```

7.7 Learning Curve Visualization

Plots per-episode total reward to verify learning progress.

```
#Plot learning curve
plt.plot(rewards)
plt.xlabel("Episode")
plt.ylabel("Total Reward")
plt.title("Q-Learning Training Progress")
plt.grid(True)
plt.tight_layout()
plt.show()
```

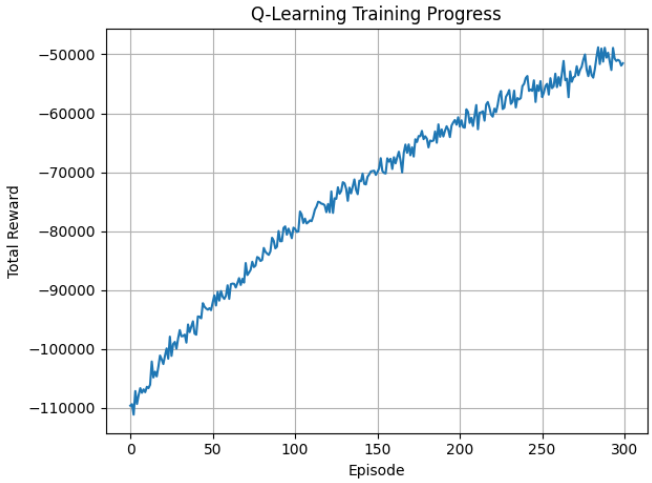


Fig 02 — Q-Learning Training Progress

Per-episode total reward across 300 episodes for the tabular agent (α=0.1, γ=0.99, ε: 1.0→0.05, 10 bins). The upward trend (less negative reward) indicates policy improvement as ε decays; residual variance reflects ongoing exploration.

8 Strategy Evaluation (Static / ML-only / RL-only / Hybrid)

8.1 Strategy Evaluation Routine — Definition and Behavior

Defines a reusable function that runs one full episode under the selected strategy and returns a step-by-step log (forecast, chosen config, latency, cost, reward, SLA flag).

```python
def evaluate_strategy_a2c(env, strategy='ml+rl', fixed_action=None):
    """Run one episode with the given strategy and return step logs."""
    logs = []
    state, _ = env.reset()
    done = False
    step = 0

    while not done:
        forecast = env.forecast_sequence[env.current_step] / env.max_forecast

        if strategy == 'static':
            action = fixed_action

        elif strategy == 'ml-only':
            if forecast > 0.75:
                action = 15  # Max provision
            elif forecast > 0.5:
                action = 10  # Medium provision
            else:
                action = 0   # Low provision

        elif strategy == 'rl-only':
            dynamic_state = state + np.random.uniform(-0.05, 0.05, size=3)
            dynamic_state = np.clip(dynamic_state, 0, 1)
            action, _ = a2c_model.predict(dynamic_state, deterministic=True)

        elif strategy == 'ml+rl':
            forecast = env.forecast_sequence[env.current_step] / env.max_forecast
            state[0] = forecast
            action, _ = a2c_model.predict(state, deterministic=True)
            memory, timeout, concurrency = env._decode_action(action)

            # Track latency trend
            if len(logs) >= 3:
                recent_latencies = [log["latency"] for log in logs[-3:]]
                avg_recent_latency = sum(recent_latencies) / 3
            else:
                avg_recent_latency = 200  # default conservative
```

```
            # Rule 1: If latency is low and concurrency is high → downscale to save cost
            if avg_recent_latency < 150 and concurrency > 1:
                action = max(action - 1, 0)

            # Rule 2: If latency > 220ms or forecast > 0.85 → upscale
            elif avg_recent_latency > 220 or forecast > 0.85:
                if concurrency < 2:
                    action = min(action + 1, env.action_space.n - 1)

            # Rule 3: If recent cost is too high while latency is OK → reduce action
            if len(logs) >= 5:
                recent_costs = [log["cost"] for log in logs[-5:]]
                if np.mean(recent_costs) > 0.09 and avg_recent_latency < 200:
                    action = max(action - 1, 0)

        else:
            raise ValueError("Unknown strategy")

        next_state, reward, done, _, _ = env.step(action)
        memory, timeout, concurrency = env._decode_action(action)
        latency = next_state[1] * 300
        cost = next_state[2] * 0.1
        forecast_val = next_state[0] * env.max_forecast

        logs.append({
            "step": step,
            "forecast": forecast_val,
            "memory": memory,
            "timeout": timeout,
            "concurrency": concurrency,
            "latency": latency,
            "cost": cost,
            "reward": reward,
            "sla_violated": latency > 200
        })

        state = next_state
        step += 1

    return pd.DataFrame(logs)
```

8.2 Environment Preparation and Fixed-Action Mapping

Create new environments per strategy to prevent shared state and set the fixed action index of the Static baseline.

```
# Load trained A2C model
from stable_baselines3 import A2C
a2c_model = A2C.load("a2c_serverless_model")

# Fresh environments for fair comparison
env_static_a2c = ServerlessTuningEnv(forecast_sequence=forecast_series)
env_mlonly_a2c = ServerlessTuningEnv(forecast_sequence=forecast_series)
env_rlonly_a2c = ServerlessTuningEnv(forecast_sequence=forecast_series)
env_hybrid_a2c = ServerlessTuningEnv(forecast_sequence=forecast_series)

# Evaluate all strategies
fixed_action_idx = 4
```

8.3 Execute Evaluations for All Strategies

Runs the evaluation for Static, ML-only heuristic, RL-only (Q-learning agent), and ML+RL (hybrid using the agent's policy).

```
# Evaluate all strategies
df_static_a2c = evaluate_strategy_a2c(env_static_a2c, strategy='static', fixed_action=fixed_action_idx)
df_mlonly_a2c = evaluate_strategy_a2c(env_mlonly_a2c, strategy='ml-only')
df_rlonly_a2c = evaluate_strategy_a2c(env_rlonly_a2c, strategy='rl-only')
df_hybrid_a2c = evaluate_strategy_a2c(env_hybrid_a2c, strategy='ml+rl')
```

8.4 Summary Reporting – Latency, Cost, SLA Violation

Calculates and prints mean latency, mean cost, and SLA violation rate on each strategy.

```
# Summary print
def summarize_a2c(df, label):
    """Print average latency, cost, and SLA violation rate."""
    avg_latency = df["latency"].mean()
    avg_cost = df["cost"].mean()
    sla_rate = df["sla_violated"].mean() * 100
    print(f"--- {label} (A2C) ---")
    print(f"Average Latency: {avg_latency:.2f} ms")
    print(f"Average Cost: ${avg_cost:.6f}")
    print(f"SLA Violation Rate: {sla_rate:.2f}%\n")

summarize_a2c(df_static_a2c, "Static Baseline")
summarize_a2c(df_mlonly_a2c, "ML-only Baseline")
summarize_a2c(df_rlonly_a2c, "RL-only Baseline")
summarize_a2c(df_hybrid_a2c, "ML + RL (Hybrid)")
```

8.5 Artifact Persistence and Comparative Visualization

Stores fine-grained logs to CSV to allow reproducibility and produces a three-panel chart comparing average latency, cost, and SLA violation rate.

```
# Save detailed logs for later analysis
df_static_a2c.to_csv("baseline_static_a2c.csv", index=False)
df_mlonly_a2c.to_csv("baseline_ml_only_a2c.csv", index=False)
df_rlonly_a2c.to_csv("baseline_rl_only_a2c.csv", index=False)
df_hybrid_a2c.to_csv("baseline_hybrid_a2c.csv", index=False)

# Plot comparison charts
import matplotlib.pyplot as plt

strategies_a2c = ["Static", "ML-only", "RL-only", "ML+RL (A2C)"]
avg_latencies_a2c = [
    df_static_a2c["latency"].mean(),
    df_mlonly_a2c["latency"].mean(),
    df_rlonly_a2c["latency"].mean(),
    df_hybrid_a2c["latency"].mean()
]
avg_costs_a2c = [
    df_static_a2c["cost"].mean(),
    df_mlonly_a2c["cost"].mean(),
    df_rlonly_a2c["cost"].mean(),
    df_hybrid_a2c["cost"].mean()
]
sla_violations_a2c = [
    df_static_a2c["sla_violated"].mean() * 100,
    df_mlonly_a2c["sla_violated"].mean() * 100,
    df_rlonly_a2c["sla_violated"].mean() * 100,
    df_hybrid_a2c["sla_violated"].mean() * 100
]

fig, axs = plt.subplots(1, 3, figsize=(18, 5))
```

```python
# Chart 1 - Latency chart
axs[0].bar(strategies_a2c, avg_latencies_a2c, color='skyblue')
axs[0].set_title("Average Latency (ms)")
axs[0].set_ylabel("Latency (ms)")
axs[0].grid(axis='y')
for i, val in enumerate(avg_latencies_a2c):
    axs[0].text(i, val + 3, f"{val:.1f}", ha='center')

# Chart 2 - Cost chart
axs[1].bar(strategies_a2c, avg_costs_a2c, color='lightgreen')
axs[1].set_title("Average Cost (USD)")
axs[1].set_ylabel("Cost (USD)")
axs[1].grid(axis='y')
for i, val in enumerate(avg_costs_a2c):
    axs[1].text(i, val + 0.0001, f"{val:.5f}", ha='center')

# Chart 3 - SLA chart
axs[2].bar(strategies_a2c, sla_violations_a2c, color='salmon')
axs[2].set_title("SLA Violation Rate (%)")
axs[2].set_ylabel("Violation Rate (%)")
axs[2].grid(axis='y')
for i, val in enumerate(sla_violations_a2c):
    axs[2].text(i, val + 2, f"{val:.1f}%", ha='center')

plt.suptitle("Performance Comparison Across Strategies (A2C)", fontsize=14, y=1.05)
plt.tight_layout()
plt.show()
```

```
--- Static Baseline (A2C) ---
Average Latency: 181.89 ms
Average Cost: $0.071125
SLA Violation Rate: 11.53%

--- ML-only Baseline (A2C) ---
Average Latency: 208.41 ms
Average Cost: $0.045863
SLA Violation Rate: 71.70%

--- RL-only Baseline (A2C) ---
Average Latency: 155.09 ms
Average Cost: $0.100000
SLA Violation Rate: 6.21%

--- ML + RL (Hybrid) (A2C) ---
Average Latency: 118.60 ms
Average Cost: $0.090639
SLA Violation Rate: 5.47%
```

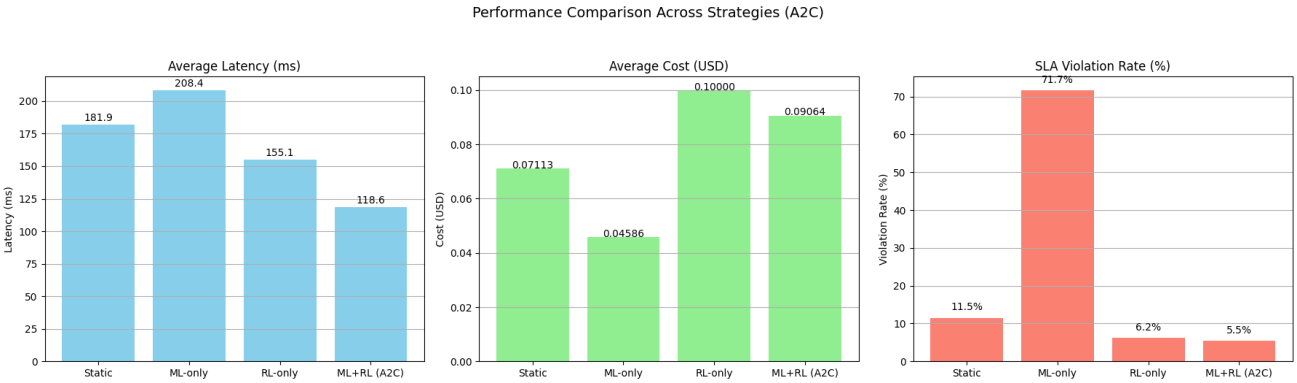


Fig 03 – Performance Comparison Across Strategies (A2C)

9 Hybrid Algorithm Comparison – Q-Learning vs A2C

9.1 Metric Aggregation

Prepares side-by-side metrics for the two hybrid strategies: ML+RL (Q-learning) and ML+RL (A2C). Assumes df_hybrid (Q-learning) and df_hybrid_a2c (A2C) have been produced by the earlier evaluation routine.

```python
# Labels for comparison
strategy_labels = ["ML+RL (Q-learning)", "ML+RL (A2C)"]

# Extract metrics
latency_values = [
    df_hybrid["latency"].mean(),
    df_hybrid_a2c["latency"].mean()
]

cost_values = [
    df_hybrid["cost"].mean(),
    df_hybrid_a2c["cost"].mean()
]

sla_values = [
    df_hybrid["sla_violated"].mean() * 100,
    df_hybrid_a2c["sla_violated"].mean() * 100
]
```

9.2 Comparative visualization

Generates three bar charts—Average Latency, Average Cost, and SLA Violation Rate—to compare algorithmic performance at a glance.

```python
# Plotting
fig, axs = plt.subplots(1, 3, figsize=(15, 5))

# Chart 1 - Latency Plot
axs[0].bar(strategy_labels, latency_values, color='cornflowerblue')
axs[0].set_title("Average Latency (ms)")
axs[0].set_ylabel("Latency (ms)")
axs[0].grid(axis='y')
for i, val in enumerate(latency_values):
    axs[0].text(i, val + 3, f"{val:.1f}", ha='center', fontsize=10)

# Chart 2 - Cost Plot
axs[1].bar(strategy_labels, cost_values, color='mediumseagreen')
axs[1].set_title("Average Cost (USD)")
axs[1].set_ylabel("Cost (USD)")
axs[1].grid(axis='y')
for i, val in enumerate(cost_values):
    axs[1].text(i, val + 0.0001, f"{val:.5f}", ha='center', fontsize=10)

# Chart 3 - SLA Violation Plot
axs[2].bar(strategy_labels, sla_values, color='lightcoral')
axs[2].set_title("SLA Violation Rate (%)")
axs[2].set_ylabel("Violation (%)")
axs[2].grid(axis='y')
for i, val in enumerate(sla_values):
    axs[2].text(i, val + 2, f"{val:.1f}%", ha='center', fontsize=10)

plt.suptitle("Algorithm Comparison: Q-Learning vs A2C", fontsize=14, y=1.05)
plt.tight_layout()
plt.show()
```

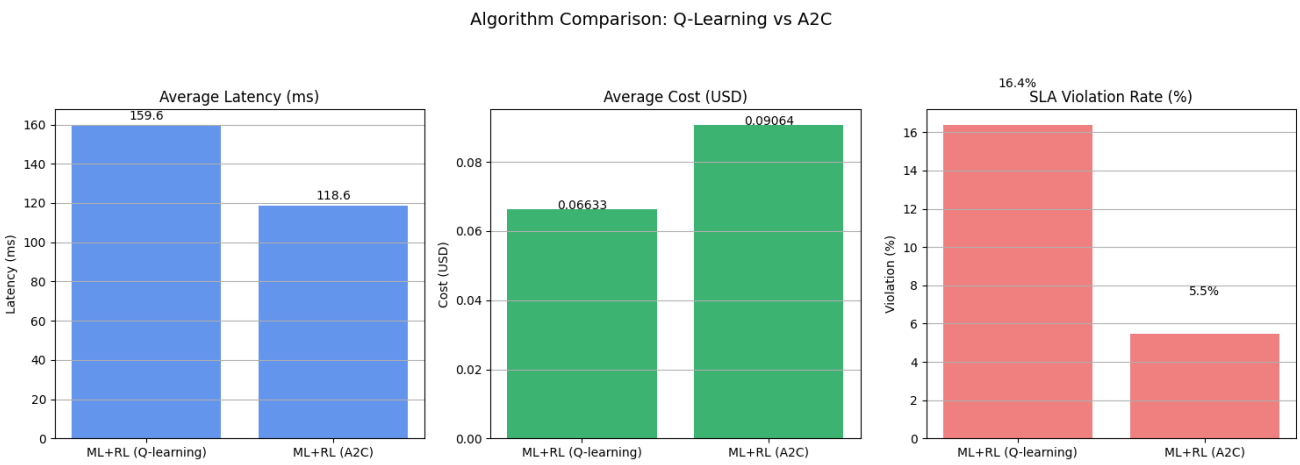


Fig 04 – Hybrid Algorithm Comparison – Q-Learning vs A2C

10 A2C Evaluation Across Traffic Patterns

10.1 Dependencies and Model Loading

Loads Stable-Baselines3 utilities and common libraries, then loads the previously trained A2C policy (a2c_serverless_model.zip).
Assumes ServerlessTuningEnv and forecast_series are already defined in the above steps.

```
from stable_baselines3 import A2C
from stable_baselines3.common.vec_env import DummyVecEnv
import numpy as np
import pandas as pd
import random
import matplotlib.pyplot as plt

# Load trained A2C model
a2c_model = A2C.load("a2c_serverless_model")
```

10.2 Traffic Pattern Generator

Defines generate_traffic(pattern, total_points) to create bursty, periodic, and irregular sequences with realistic variation.

```
# Enhanced traffic pattern generator with realistic variation
def generate_traffic(pattern, total_points):
    base = 100
    if pattern == "bursty":
        return [int(base + (np.sin(i / 15.0) * 60 + (i % 20) * 10 + np.random.normal(0, 10))) for i in range(total_points)]
    elif pattern == "periodic":
        return [int(base + np.cos(i / 10.0) * 50 + 30 * np.sin(i / 30.0) + np.random.normal(0, 10)) for i in range(total_points)]
    elif pattern == "irregular":
        return [int(base + random.randint(-120, 180)) for i in range(total_points)]
    else:
        raise ValueError("Invalid traffic pattern")
```

10.3 A2C Evaluation Routine

Runs one episode in a vectorized env using the loaded policy; returns a DataFrame with mean latency, mean cost, and SLA violation rate.

```
# Evaluation function
def evaluate_with_a2c(forecast_seq, model):
    env = DummyVecEnv([lambda: ServerlessTuningEnv(forecast_seq)])
    obs = env.reset()
    done = False

    latency_list = []
    cost_list = []
    sla_violations = 0
    total_steps = 0

    while not done:
        action, _ = model.predict(obs, deterministic=True)
        obs, reward, done, info = env.step(action)

        latency = info[0]["latency"]
        cost = info[0]["cost"]
        violated = info[0]["sla_violated"]

        latency_list.append(latency)
        cost_list.append(cost)
        sla_violations += int(violated)
        total_steps += 1

    return pd.DataFrame({
        "Avg_Latency": [np.mean(latency_list)],
        "Avg_Cost": [np.mean(cost_list)],
        "SLA_Violation": [sla_violations / total_steps * 100]
    })
```

10.4 Generate Traffic and Execute Evaluations

Creates three traffic sequences of identical length to forecast_series, then evaluates the A2C policy on each.

```
# Generate traffic
total_points = len(forecast_series)
traffic_bursty = generate_traffic("bursty", total_points)
traffic_periodic = generate_traffic("periodic", total_points)
traffic_irregular = generate_traffic("irregular", total_points)

# Evaluate A2C on all patterns
df_bursty = evaluate_with_a2c(traffic_bursty, a2c_model)
df_periodic = evaluate_with_a2c(traffic_periodic, a2c_model)
df_irregular = evaluate_with_a2c(traffic_irregular, a2c_model)
```

10.5 Summary Reporting and Visualization

Prints concise summaries and renders a three-panel comparison (latency, cost, SLA violation).

```
# Summary print
def summarize(df, label):
    print(f"--- {label} ---")
    print(f"Average Latency: {df['Avg_Latency'].mean():.2f} ms")
    print(f"Average Cost: ${df['Avg_Cost'].mean():.6f}")
    print(f"SLA Violation Rate: {df['SLA_Violation'].mean():.2f}%\n")

summarize(df_bursty, "Bursty Traffic")
summarize(df_periodic, "Periodic Traffic")
summarize(df_irregular, "Irregular Traffic")

# Visualization
patterns = ["Bursty", "Periodic", "Irregular"]
latencies = [df_bursty["Avg_Latency"].mean(), df_periodic["Avg_Latency"].mean(), df_irregular["Avg_Latency"].mean()]
costs = [df_bursty["Avg_Cost"].mean(), df_periodic["Avg_Cost"].mean(), df_irregular["Avg_Cost"].mean()]
violations = [df_bursty["SLA_Violation"].mean(), df_periodic["SLA_Violation"].mean(), df_irregular["SLA_Violation"].mean()]

x = np.arange(len(patterns))

# Subplots in one row
fig, axs = plt.subplots(1, 3, figsize=(18, 5))
bar_width = 0.6

# Latency subplot
axs[0].bar(x, latencies, color='skyblue', width=bar_width)
axs[0].set_xticks(x)
axs[0].set_xticklabels(patterns)
axs[0].set_ylabel("Latency (ms)")
axs[0].set_title("Average Latency")
axs[0].grid(axis='y')
for i, val in enumerate(latencies):
    axs[0].text(i, val + 1, f"{val:.2f}", ha='center', fontsize=9)

# Cost subplot
axs[1].bar(x, costs, color='lightgreen', width=bar_width)
axs[1].set_xticks(x)
axs[1].set_xticklabels(patterns)
axs[1].set_ylabel("Cost (USD)")
axs[1].set_title("Average Cost")
axs[1].grid(axis='y')
for i, val in enumerate(costs):
    axs[1].text(i, val + 0.002, f"{val:.5f}", ha='center', fontsize=9)

# SLA Violation subplot
axs[2].bar(x, violations, color='salmon', width=bar_width)
axs[2].set_xticks(x)
axs[2].set_xticklabels(patterns)
axs[2].set_ylabel("Violation Rate (%)")
axs[2].set_title("SLA Violation Rate")
axs[2].grid(axis='y')
for i, val in enumerate(violations):
    axs[2].text(i, val + 1, f"{val:.2f}%", ha='center', fontsize=9)

# Overall layout
fig.suptitle("A2C Evaluation Across Traffic Patterns", fontsize=14)
plt.tight_layout(rect=[0, 0, 1, 0.95])
plt.show()
```

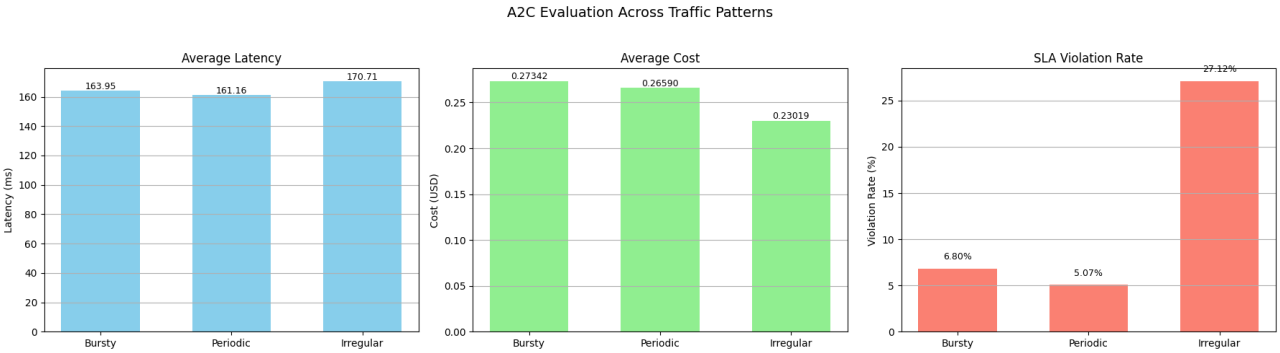


Fig 05 – A2C Evaluation Across Traffic Patterns

11 A2C Hyperparameter Sensitivity Analysis

11.1 Experiment Setup

Imports required packages and establishes the experiment context for testing learning rate (α), discount factor (γ), and entropy coefficient (ent).

```
import pandas as pd
import matplotlib.pyplot as plt
from stable_baselines3 import A2C
from stable_baselines3.common.vec_env import DummyVecEnv
```

11.2 Experiment Function (train_a2c_with_params)

Defines a reusable routine that builds a vectorized environment, instantiates an A2C model with the supplied hyperparameters, performs a short rollout (300 steps), and returns the reward series for later smoothing and comparison.

```
# --- Define the experiment function ---
def train_a2c_with_params(forecast_series, learning_rate, gamma, ent_coef, label):
    env = DummyVecEnv([lambda: ServerlessTuningEnv(forecast_series)])
    model = A2C("MlpPolicy", env, learning_rate=learning_rate, gamma=gamma,
                ent_coef=ent_coef, verbose=0)

    rewards = []
    obs = env.reset()
    for _ in range(300):
        action, _ = model.predict(obs)
        obs, reward, done, info = env.step(action)
        rewards.append(reward[0])
    return rewards, label
```

11.3 Hyperparameter Configurations (param_configs)

Specifies the grid to explore: three learning rates, three γ values, and three entropy coefficients. Each config includes a label and color for plotting.

```
# --- Define hyperparameter configurations ---
param_configs = [
    {"learning_rate": 0.01, "gamma": 0.99,  "ent_coef": 0.01, "label": "α=0.01",  "color": "blue"},
    {"learning_rate": 0.1,  "gamma": 0.99,  "ent_coef": 0.01, "label": "α=0.1",   "color": "dodgerblue"},
    {"learning_rate": 0.001,"gamma": 0.99,  "ent_coef": 0.01, "label": "α=0.001", "color": "navy"},

    {"learning_rate": 0.01, "gamma": 0.9,   "ent_coef": 0.01, "label": "γ=0.9",   "color": "green"},
    {"learning_rate": 0.01, "gamma": 0.99,  "ent_coef": 0.01, "label": "γ=0.99",  "color": "forestgreen"},
    {"learning_rate": 0.01, "gamma": 0.999, "ent_coef": 0.01, "label": "γ=0.999", "color": "darkgreen"},

    {"learning_rate": 0.01, "gamma": 0.99,  "ent_coef": 0.0,  "label": "Ent=0.0", "color": "red"},
    {"learning_rate": 0.01, "gamma": 0.99,  "ent_coef": 0.01, "label": "Ent=0.01","color": "tomato"},
    {"learning_rate": 0.01, "gamma": 0.99,  "ent_coef": 0.1,  "label": "Ent=0.1", "color": "darkred"},
]
```

11.4 Run Experiments

Loops through each configuration, trains/evaluates with the helper, applies a 10-episode rolling mean to smooth noise, and stores series + metadata for plotting.

```
# --- Run experiments ---
results = []
for config in param_configs:
    # Extract only the required keys for training
    train_args = {k: config[k] for k in ['learning_rate', 'gamma', 'ent_coef', 'label']}

    # Train and collect results
    rewards, label = train_a2c_with_params(forecast_series, **train_args)
    smoothed = pd.Series(rewards).rolling(window=10).mean()

    results.append((smoothed, label, config["color"]))
```

11.5 Visualization and Best Configuration Annotation

Plots all smoothed reward curves on one figure and auto-annotates the configuration achieving the highest smoothed reward.

```
# --- Run experiments ---
results = []
for config in param_configs:
    # Extract only the required keys for training
    train_args = {k: config[k] for k in ['learning_rate', 'gamma', 'ent_coef', 'label']}

    # Train and collect results
    rewards, label = train_a2c_with_params(forecast_series, **train_args)
    smoothed = pd.Series(rewards).rolling(window=10).mean()

    results.append((smoothed, label, config["color"]))

# --- Plot results ---
plt.figure(figsize=(14, 6))

# Track best config info
best_value = float('-inf')
best_ep = -1
best_label = ''
best_color = ''

for smoothed, label, color in results:
    plt.plot(smoothed, label=label, color=color)

    # Check max reward value in this line
    max_val = smoothed.max()
    max_ep = smoothed.idxmax()

    if max_val > best_value:
        best_value = max_val
        best_ep = max_ep
        best_label = label
        best_color = color
```

```
# Annotate the best config
plt.annotate(
    f"Best: {best_label}",
    xy=(best_ep, best_value),
    xytext=(best_ep + 10, best_value + 2),
    color=best_color,
    arrowprops=dict(arrowstyle="->", color=best_color),
    fontsize=9,
    bbox=dict(boxstyle="round,pad=0.3", facecolor="white", edgecolor=best_color)
)

plt.title("A2C Reward Trends Across Hyperparameter Variations")
plt.xlabel("Episode")
plt.ylabel("Smoothed Reward")
plt.legend(loc='upper left', ncol=3, fontsize='small')
plt.grid(True)
plt.tight_layout()
plt.show()
```

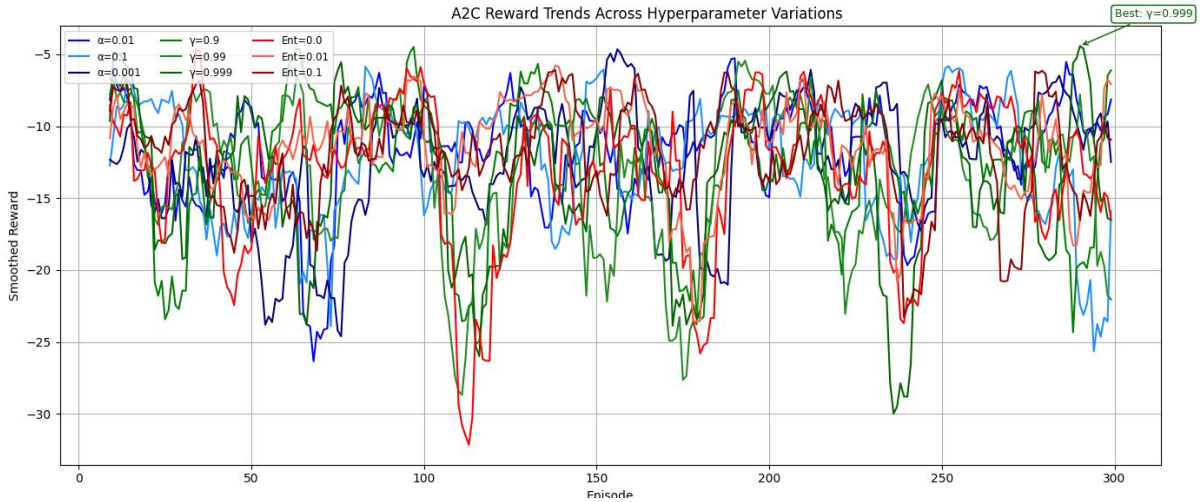


Fig 05 – A2C Hyperparameter Sensitivitys

12 Cost–Latency Trade-off Simulation

12.1 Dependencies and Cost-Prioritized Environment

Defines a variation of the environment where cost is foremost in the reward, but where latency is still penalized. Action is being decoded through the step function, latency/cost is simulated, observation is normalized, and cost-weighted reward is calculated.
(Assumes core imports like numpy as np and pandas as pd are already available from earlier sections.).

```python
from stable_baselines3 import A2C
from stable_baselines3.common.vec_env import DummyVecEnv

# --- Custom environment prioritizing cost ---
class CostPriorityEnv(ServerlessTuningEnv):
    def step(self, action):
        # Decode action to get actual configuration
        memory, timeout, concurrency = self._decode_action(action)

        # Simulate latency: inverse relation with memory and concurrency
        latency = max(100, 300 - memory / 16 - concurrency * 10)

        # Simulate cost: function of memory, timeout, concurrency
        cost = (memory / 1024) * timeout * concurrency * 0.002  # Adjust unit price as needed

        # SLA violation threshold
        sla_violated = latency > 200

        # Forecast for this step
        forecast = self.forecast_sequence[self.current_step]
        normalized_forecast = forecast / self.max_forecast

        # Normalize for state space
        normalized_latency = latency / 300
        normalized_cost = cost / 0.1  # assuming 0.1 is upper bound

        self.state = np.array([normalized_forecast, normalized_latency, normalized_cost])

        # Reward prioritizes cost, slightly penalizes latency
        reward = - (0.7 * cost + 0.3 * latency / 300)

        self.current_step += 1
        done = self.current_step >= len(self.forecast_sequence)

        return self.state, reward, done, False, {}
```

12.2 A2C Training in Cost-Prioritized Environment

Build the cost-prioritized environment and train an A2C policy to minimize the cost and maintain the latency at an acceptable range.

```python
# --- Train A2C in cost-prioritized environment ---
cost_env = DummyVecEnv([lambda: CostPriorityEnv(forecast_series)])
cost_model = A2C("MlpPolicy", cost_env, verbose=0)
cost_model.learn(total_timesteps=10_000)
```

12.3 Evaluation Routine and Execution

Evaluates the trained cost-prioritized policy and logs stepwise metrics (config, latency, cost, reward, SLA).

```python
# --- Evaluate the cost-prioritized agent ---
def evaluate_cost_model(model, forecast_seq):
    env = CostPriorityEnv(forecast_seq)
    vec_env = DummyVecEnv([lambda: env])
    obs = vec_env.reset()

    logs = []
    done = False
    step = 0
    while not done:
        action, _ = model.predict(obs, deterministic=True)
        next_obs, reward, done, info = vec_env.step(action)
        scalar_action = int(action[0]) if isinstance(action, np.ndarray) else int(action)
        memory, timeout, concurrency = env._decode_action(scalar_action)

        latency = next_obs[0][1] * 300
        cost = next_obs[0][2] * 0.1
        forecast_val = forecast_seq[min(step, len(forecast_seq)-1)]
        logs.append({
            "step": step,
            "forecast": forecast_val,
            "memory": memory,
            "timeout": timeout,
            "concurrency": concurrency,
            "latency": latency,
            "cost": cost,
            "reward": reward[0],
            "sla_violated": latency > 200
        })
        obs = next_obs
        step += 1

    return pd.DataFrame(logs)

df_cost_priority = evaluate_cost_model(cost_model, forecast_series)
```

12.4 Baseline (Default A2C) for Comparison

Generates a comparison DataFrame using the **original** A2C setup.
(Assumes a helper evaluate_strategy_a2c(...) exists in the notebook, consistent with earlier sections.)

```python
# --- Evaluate original A2C agent for comparison ---
df_a2c_default = evaluate_strategy_a2c(
    env=ServerlessTuningEnv(forecast_sequence=forecast_series),
    strategy='ml+rl'
)
```

12.5 Summary Reporting

To bring out the trade-off, printed values aggregate metrics where both the default and cost-prioritized policies are aggregated.

```python
# --- Summary print ---
def summarize(df, label):
    print(f"--- {label} ---")
    print(f"Avg Latency: {df['latency'].mean():.2f} ms")
    print(f"Avg Cost: ${df['cost'].mean():.6f}")
    print(f"SLA Violation Rate: {df['sla_violated'].mean() * 100:.2f}%\n")

summarize(df_a2c_default, "Original A2C")
summarize(df_cost_priority, "Cost-Prioritized A2C")
```

12.6 Comparative Visualization — Default vs Cost-Prioritized A2C

Renders side-by-side bar charts for **average latency**, **average cost**, and **SLA violation rate** to illustrate the trade-off between the default A2C policy and the cost-prioritized variant.

```
# --- Plot comparison ---
import matplotlib.pyplot as plt

models = ["Original A2C", "Cost-Prioritized A2C"]

plt.figure(figsize=(12, 4))

# For latency chart
plt.subplot(1, 3, 1)
plt.bar(models, [df_a2c_default["latency"].mean(), df_cost_priority["latency"].mean()], color='skyblue')
plt.ylabel("Latency (ms)")
plt.title("Avg Latency")

# For Cost chart
plt.subplot(1, 3, 2)
plt.bar(models, [df_a2c_default["cost"].mean(), df_cost_priority["cost"].mean()], color='mediumseagreen')
plt.ylabel("Cost (USD)")
plt.title("Avg Cost")

# For SLA chart
plt.subplot(1, 3, 3)
plt.bar(models, [df_a2c_default["sla_violated"].mean()*100, df_cost_priority["sla_violated"].mean()*100], color='salmon')
plt.ylabel("SLA Violation (%)")
plt.title("SLA Violation Rate")

plt.tight_layout()
plt.show()
```

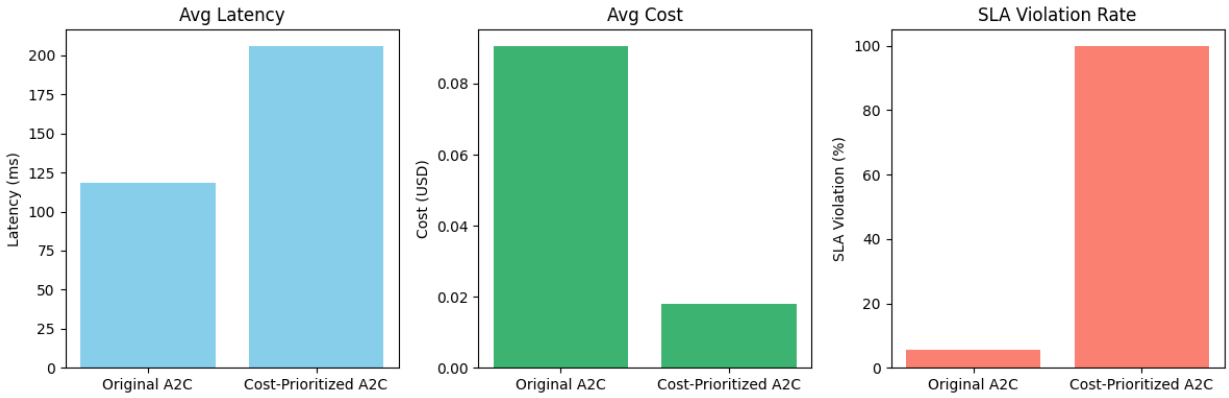


Fig 06 - Default vs Cost-Prioritized A2C

13 AWS Lambda PoC

13.1 Purpose

Deploy a basic-level Lambda function (Python 3.10) (Van Rossum and Drake; 2009) in Cloud9 with the help of the Boto3 SDK (PYTHON) (Amazon Web Services, n.d.) and monitor deployment through CloudWatch metrics to test the behavior of the run-time. The PoC replicates the decision logic of the notebook at a light production-ready level.

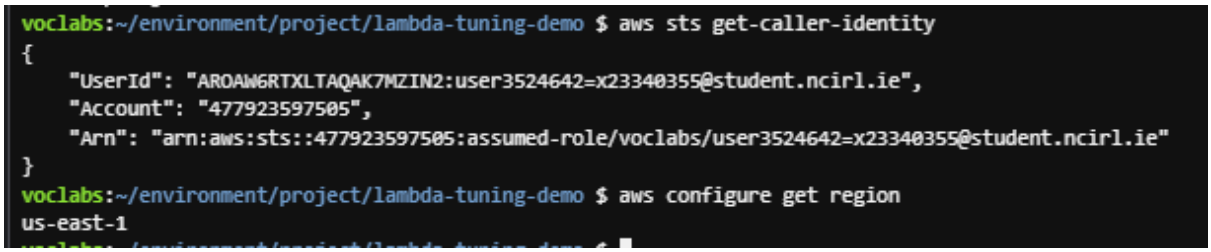
13.2 Prerequisites

- **Cloud9** environment with AWS CLI v2 configured.

- AWS account and following IAM permissions will be needed as minimum. *lambda:UpdateFunctionConfiguration, cloudwatch:GetMetricStatistics*.

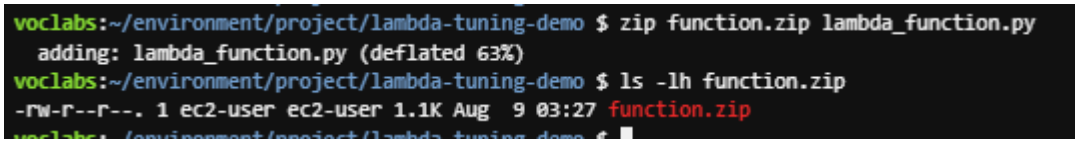
- Files available in the workspace:
  - lambda_function.py (Lambda handler)
  - cloudwatch_metric_fetch.py (local script to retrieve metrics)

13.3 Verify AWS context (account & region)



13.4 Package the Lambda function

Zips only the handler file (no external deps assumed; boto3 is available in the Lambda runtime).



13.5 Create the Lambda function

This step is only for first-time deployment

First, creates the function with Python 3.10 runtime and basic handler entry point "lambda_function.lambda_handler" using the existing IAM role.

```
aws lambda create-function \
  --function-name lambda-tuning-demo \
  --runtime python3.10 \
  --role arn:aws:iam::477923597505:role/LabRole \
  --handler lambda_function.lambda_handler \
  --zip-file fileb://function.zip
```

13.6 Optional runtime configuration

Adjust memory, timeout, and (optionally) reserved concurrency to match evaluation scenarios.

```
aws lambda update-function-configuration \
  --function-name lambda-tuning-demo \
  --memory-size 384 \
```

```
  --timeout 10

# (Optional) reserved concurrency
aws lambda put-function-concurrency \
  --function-name lambda-tuning-demo \
  --reserved-concurrent-executions 2
```

13.7 Traffic & Metrics Script (Cloud9 Terminal)

Creates a short burst of invocations, waits for CloudWatch aggregation, then calls the function once more to return a **decision payload** (avg duration, invocations, decision, new memory, update status).

```
--Bash--
# cloudwatch_metric_fetch.py  (runs as a bash script in Cloud9)
for i in {1..15}; do
  aws lambda invoke --function-name lambda-tuning-demo --payload '{}' /dev/null
done
sleep 180
aws lambda invoke --function-name lambda-tuning-demo output.json
cat output.json
```

Notes

- The initial 15 invocations generate recent traffic.
- sleep 180 (~3 minutes) allows CloudWatch to aggregate metrics at 1-minute granularity.
- The final invoke prints the decision JSON to output.json.

13.8 Execute the Script

Run the script from the Cloud9 terminal. If needed, call with bash explicitly.

```
bash cloudwatch_metric_fetch.py
# or
chmod +x cloudwatch_metric_fetch.py
./cloudwatch_metric_fetch.py
```

13.9 Expected Decision Payload (Example)

The below JSON message confirms the successful run and indicates the “measured load”, “policy decision”, with applied configuration.

{"avg_duration": 249.04500000000002, "invocations": 4.0, "decision": "Low load", "new_memory": 256, "update_status": "Updated memory to 256 MB"}

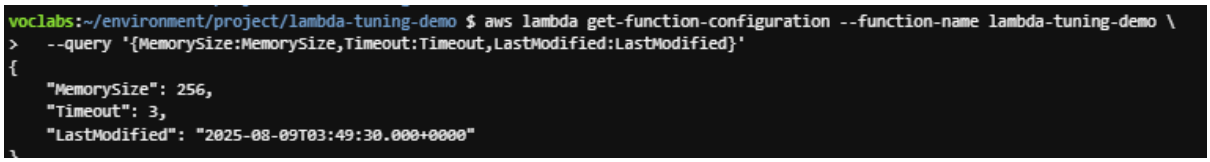
Field meanings:

- avg_duration — Average function duration (ms) over the recent window used by the handler logic.
- invocations — Recent request count considered by the policy.
- decision — Policy classification (e.g., Low/Medium/High load).
- new_memory — Target memory size selected by the policy.
- update_status — Result from UpdateFunctionConfiguration.

13.10 Confirm Configuration Change

Verify the function’s memory size was updated as reported in the JSON.

aws lambda get-function-configuration --function-name lambda-tuning-demo \
  --query '{MemorySize:MemorySize,Timeout:Timeout,LastModified:LastModified}'



13.11 Troubleshooting

Verify the function’s memory size was updated as reported in the JSON.

- No JSON fields / empty output → Ensure the last line in the script runs aws lambda invoke ... output.json and cat output.json.
- Stale metrics / zero invocations → Increase sleep to 240–300 seconds to allow metric refresh; re-run.
- AccessDenied → Verify Cloud9 credentials and that the IAM role used by the function allows logs:CreateLogStream, logs:PutLogEvents, and that the Cloud9 user has lambda:InvokeFunction and lambda:GetFunctionConfiguration.
- Memory not updating → Confirm the Lambda execution role allows lambda:UpdateFunctionConfiguration if the update call is made inside the handler. If updates are made by the Cloud9 user, ensure that user’s permissions include the same action.

References

Chen, T. and Guestrin, C. (2016). Xgboost: A scalable tree boosting system, Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16), ACM, San Francisco, CA, USA, 13–17 August 2016, pp. 785–794. Available at: https://doi.org/10.1145/2939672.2939785 [Accessed 22 June 2025].

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V. and Vanderplas, J. (2011). Scikit-learn: Machine learning in python, Journal of Machine Learning Research 12: 2825–2830. Available at: https://www.jmlr.org/papers/volume12/pedregosa11a/pedregosa11a.pdf [Accessed 01 July 2025].

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J. and Zaremba, W. (2016). Openai gym, arXiv preprint arXiv:1606.01540. Available at: https://arxiv.org/abs/1606.01540 [Accessed 20 June 2025].

Van Rossum, G. and Drake, F. (2009). Python 3 Reference Manual, CreateSpace, Scotts Valley, CA. Available at: https://dl.acm.org/doi/book/10.5555/1593511 [Accessed 04 June 2025].

Amazon Web Services (n.d.). Boto3 documentation, Available at: https://boto3.amazonaws.com/v1/documentation/api/latest/index.html. [Accessed 24 Jun. 2025].