BCSE308L - Computer Network Research submission

An Integrated Framework for the Simulation and Predictive Analysis of TCP Congestion Control Algorithms

Submitted by:

Dhriti Saxena (23BCE0182)

Ashmit Dudeja (23BCE0350)

Manya Kothari (23BCE0048)

Mohit Parashar (23BCE0231)

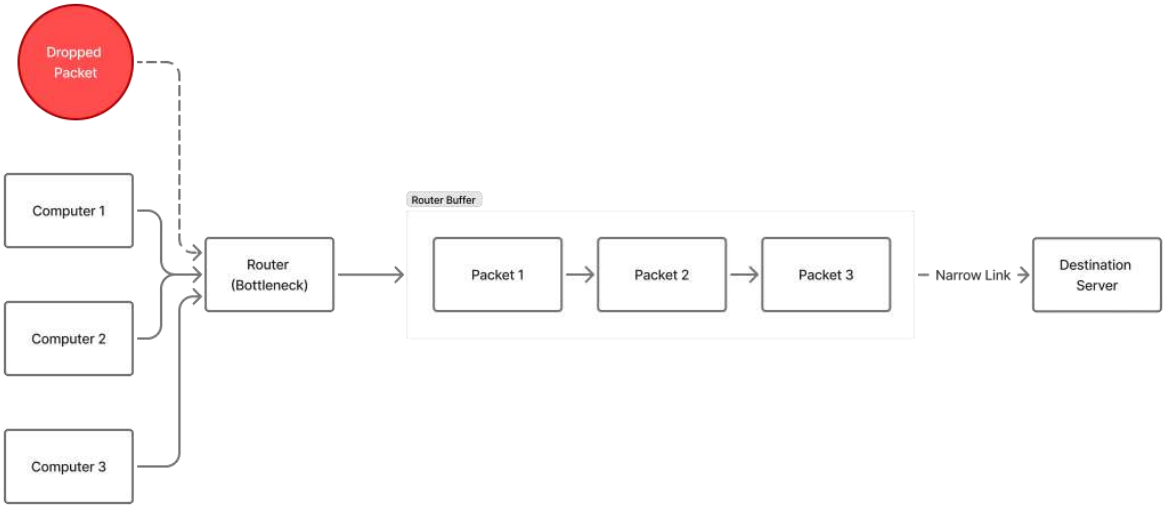
1. Abstract

The selection of an optimal Transmission Control Protocol (TCP) Congestion Control Algorithm (CCA) is a critical and complex challenge in modern network engineering, with performance being highly contingent on dynamic network conditions. This report details the design, implementation, and evaluation of a comprehensive framework developed to address this challenge. The framework provides a dual-pronged solution: a high-fidelity network simulation tool for the empirical performance comparison of five seminal CCAs—CUBIC, BBR, Reno, Vegas, and LEDBAT—and a predictive machine learning engine for proactive algorithm recommendation. The simulation component allows for the analysis of key performance indicators, including throughput, round-trip time, loss rate, and fairness, under user-defined network configurations. The predictive component leverages an Extreme Gradient Boosting (XGBoost) classifier, trained on a comprehensive, synthetically generated dataset, to forecast the best-performing CCA for a given set of network parameters, providing the recommendation along with predicted performance metrics and a confidence score. The results demonstrate the XGBoost model's high predictive accuracy and validate the nuanced performance trade-offs between different CCA paradigms (loss-based, delay-based, and model-based) that the simulation tool elucidates. This project contributes a practical decision-support tool for network administrators and a robust framework for academic research, bridging the gap between empirical analysis and intelligent automation in the optimization of network performance.

2. Introduction

2.1 The Enduring Challenge of Network Congestion

The Transmission Control Protocol (TCP) serves as the bedrock of reliable communication on the internet, managing the flow of data to ensure integrity and order. However, the very nature of a shared, distributed network infrastructure gives rise to an inevitable and persistent challenge: network congestion. Congestion occurs when the aggregate data rate from multiple sources exceeds the capacity of a bottleneck link or router, leading to the formation of queues in router buffers. If left unmanaged, these queues can overflow, causing routers to drop packets. This phenomenon not only degrades performance but can also lead to a catastrophic state known as "congestion collapse," where uncontrolled retransmissions saturate the network, reducing effective throughput to near zero. The historical reality of the 1986 internet congestion collapse served as a powerful impetus for the development of robust control mechanisms, a field pioneered by Van Jacobson.



At its core, TCP's reliability mechanism is a two-part strategy. It employs flow control to prevent a fast sender from overwhelming the buffer capacity of a slow receiver, a function managed by the receiver's advertised window. Distinct from this is congestion control, a sender-side mechanism designed to prevent a single connection from overwhelming the shared resources of the network itself. This report focuses on the latter, exploring the sophisticated algorithms that govern how TCP senders modulate their transmission rates in response to perceived network conditions.

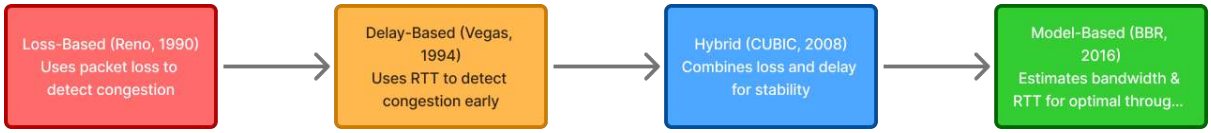
2.2 An Evolutionary Journey of Congestion Control Philosophies

The history of TCP congestion control is not merely a linear progression of technical refinements but an evolution of competing philosophies for detecting and reacting to network saturation. This intellectual journey, which forms the central theme of foundational research in the field 7, can be broadly categorized into distinct paradigms.

The initial and most enduring approach is the loss-based paradigm. This philosophy operates on a simple, powerful assumption: packet loss is the primary signal of network congestion. Algorithms like the classic TCP Reno treat packet loss, detected via retransmission timeouts or the receipt of three identical "duplicate" acknowledgments, as definitive evidence that a bottleneck buffer has overflowed. The response is to reduce the sending rate. This paradigm is inherently reactive; it takes corrective action only after congestion has already led to dropped packets, which means it tends to operate networks with persistently full buffers.

In response to the limitations of reactive control, particularly the queuing delays it induces, the delay-based paradigm emerged. This approach represents a conceptual shift towards proactive management. Algorithms such as TCP Vegas and Low Extra Delay Background Transport (LEDBAT) monitor packet delay—either Round-Trip Time (RTT) or one-way delay—as a leading indicator of congestion. An increasing delay signals that queues are building up at the bottleneck. By reacting to this early warning signal, these algorithms aim to reduce their sending rate before packet loss occurs, thereby maintaining lower latency. However, this

conservative strategy introduces its own set of challenges, most notably a struggle for fairness when competing with more aggressive loss-based flows that do not back off until loss occurs.



More recent developments have led to hybrid and model-based frontiers. TCP CUBIC, the default in many modern operating systems, can be seen as a highly evolved hybrid. While it remains fundamentally loss-based, it replaces the simple linear rate increase of its predecessors with a sophisticated cubic function. This modification allows it to scale its throughput far more effectively in modern high Bandwidth-Delay Product (BDP) networks. A more radical departure is represented by Google's Bottleneck Bandwidth and Round-trip propagation time (BBR) algorithm. BBR operates on a model-based philosophy, rejecting both loss and delay as primary, reliable signals of congestion. Instead, it attempts to build an explicit model of the network path's physical characteristics—its bottleneck bandwidth and its propagation delay. By controlling its sending rate to match this model, BBR aims to achieve the theoretical ideal: maximum throughput with minimal queuing delay. This evolution from simple reaction to proactive adjustment, and finally to explicit modelling, reflects the increasing complexity of the internet and the continuous search for more efficient and equitable ways to share its resources.

2.3 The Problem Statement: A Complex Optimization Challenge

The diversity of these control philosophies underscores a central thesis: no single Congestion Control Algorithm (CCA) is universally optimal. The ideal choice is a complex function of network characteristics—such as available bandwidth, RTT, bottleneck buffer depth, and the nature of packet loss—as well as the specific requirements of the application.

This creates a multi-dimensional optimization problem. For instance, in networks with deep buffers, a common scenario at the network edge, aggressive loss-based algorithms like CUBIC can induce significant queuing delay, a problem known as "buffer bloat," which harms the performance of latency-sensitive applications. BBR is specifically designed to mitigate this issue by avoiding queue buildup. Conversely, the first version of BBR was designed to be largely insensitive to packet loss, which could lead it to be overly aggressive and cause high retransmission rates in lossy environments, a behaviour that is less pronounced in CUBIC. Furthermore, the coexistence of different algorithms creates complex fairness dynamics. A "well-behaved" delay-based algorithm like TCP Vegas, which reduces its rate at the first sign of a queue, will systematically cede bandwidth to a competing loss-based flow like Reno, which continues to increase its rate until the buffer is full and packets are dropped. This dynamic is a classic networking example of the "Tragedy of the Commons," where a conservative actor is penalized for its good citizenship. This complex interplay of factors makes the manual selection of a CCA a daunting task for network operators and application developers, motivating the need for an intelligent decision-support framework.

2.4 Our Contribution: An Integrated Simulation and Prediction Framework

This project introduces a novel, integrated solution designed to navigate this complex optimization landscape. The framework consists of two synergistic components:

1. The Simulation Engine: This component functions as an empirical laboratory, providing a high-fidelity environment for observing and analyzing the dynamic behavior of five key CCAs: Reno, CUBIC, Vegas, LEDBAT, and BBR. Users can define a wide range of network parameters and receive a detailed comparative analysis of performance metrics, enabling a deep, data-driven understanding of the trade-offs inherent in each algorithm's design.
2. The Machine Learning Prediction Engine: This component provides an intelligent, proactive recommendation. It utilizes a pre-trained Extreme Gradient Boosting (XGBoost) model that has learned the intricate, non-linear relationships between network conditions and CCA performance from a vast corpus of simulation data. For any given network configuration, the engine predicts the optimal algorithm and its expected performance, empowering users to make informed decisions without requiring exhaustive manual testing.

The overarching objective of this project is to provide a tool that equips network engineers, researchers, and students with both the empirical data to understand why certain algorithms perform better under specific conditions and a predictive model to determine which algorithm to choose for optimal performance.

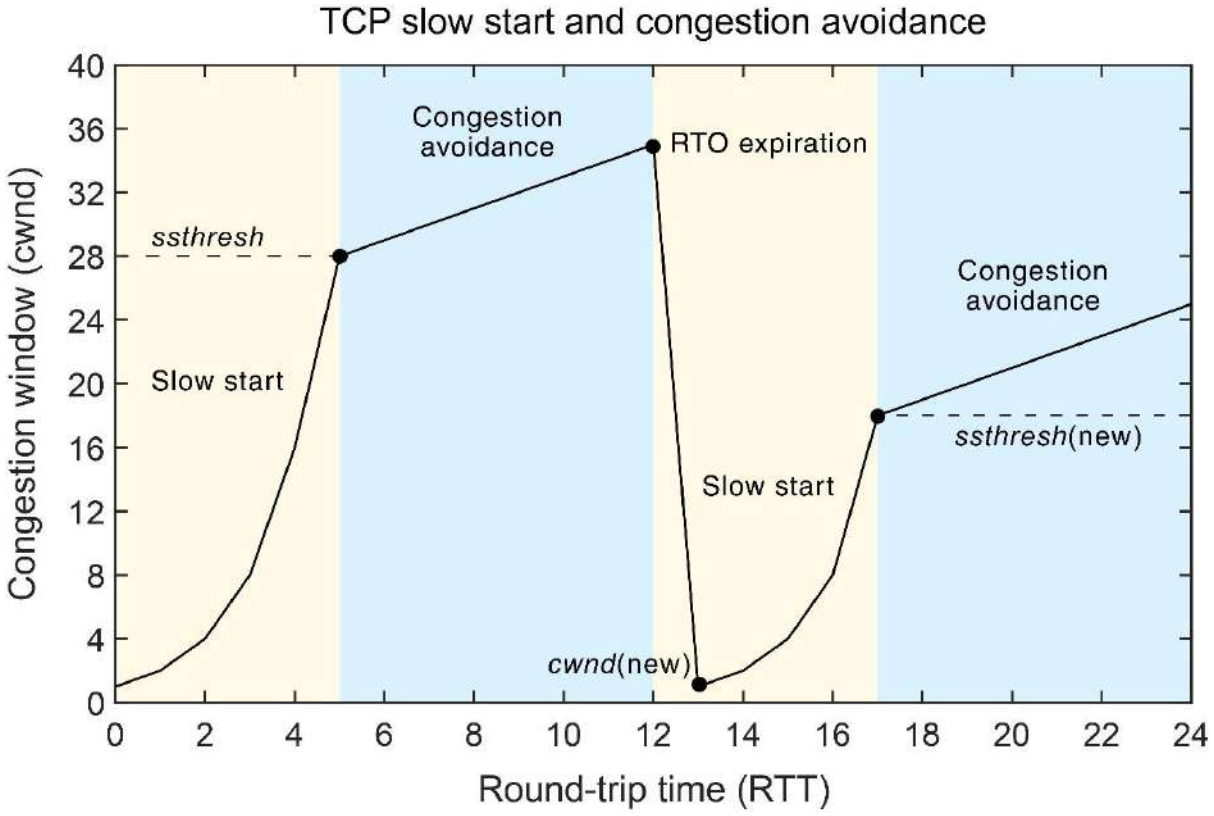
2.5 Report Organization

This report is structured to provide a comprehensive and systematic exposition of the project. Section 3 conducts a thorough literature survey, detailing the foundational principles of TCP congestion control and the specific mechanisms of the five selected algorithms. Section 4 describes the methodology and implementation of the framework, covering the simulation environment, dataset generation, data preprocessing pipeline, and the architecture and training of the XGBoost model. Section 5 presents a detailed analysis of the results, evaluating both the empirical performance of the CCAs and the predictive accuracy of the machine learning model. Finally, Section 6 concludes the report by summarizing the key findings, discussing the project's contributions and limitations, and outlining promising directions for future research.

3. Literature Survey

The field of TCP congestion control originated in response to the "congestion collapse" of the mid-1980s, a period where uncontrolled retransmissions crippled the early internet. Van Jacobson's seminal 1988 paper laid the groundwork for modern congestion management by introducing the principles of "packet conservation," slow start, and the Additive Increase, Multiplicative Decrease (AIMD) algorithm [Jacobson, 1988]. These mechanisms formed the core of TCP Tahoe and its successor, TCP Reno, which became the de facto standard. TCP

Reno established the classic loss-based paradigm, interpreting packet loss (detected via triple duplicate ACKs or timeouts) as the primary signal of network congestion and reacting by multiplicatively decreasing its sending rate. Further refinements, such as TCP NewReno, were introduced to improve performance when multiple packets were dropped from a single window of data [Floyd & Henderson, 1999].



While robust, the linear window growth of TCP Reno proved inefficient for the high Bandwidth-Delay Product (BDP) networks that emerged with the internet's expansion [Afanasyev et al., 2010]. This led to the development of high-speed variants, most notably CUBIC, which replaced Reno's linear increase with a more scalable cubic function dependent on the time elapsed since the last congestion event. This time-based growth not only allowed for faster recovery and better utilization of "long fat networks" but also improved fairness between flows with different Round-Trip Times (RTTs). CUBIC's success led to its adoption as the default congestion control algorithm in Linux, macOS, and Windows, solidifying the dominance of the loss-based, reactive control philosophy. Other notable algorithms from this era, such as TCP Westwood, introduced bandwidth estimation to achieve faster recovery, particularly in wireless networks where packet loss is not always a sign of congestion [Casetti et al., 2002].

In parallel, a proactive, delay-based paradigm emerged, seeking to anticipate and avoid congestion rather than simply reacting to it. TCP Vegas was a pioneering algorithm in this space, using changes in RTT to estimate the queue size at the bottleneck link. By aiming to keep a small, stable number of packets in the queue, Vegas could achieve lower latency and

fewer packet losses than its loss-based counterparts. However, this conservative approach proved to be a significant disadvantage when competing with more aggressive algorithms like Reno, which would fill the buffer that Vegas was trying to keep clear, often starving the Vegas flow of bandwidth. A more specialized delay-based algorithm, LEDBAT (Low Extra Delay Background Transport), was later developed for background traffic. LEDBAT uses one-way delay measurements to maintain a target queuing delay, allowing it to scavenge available bandwidth without interfering with latency-sensitive foreground traffic by quickly yielding to competing flows.

A more recent and fundamental shift in congestion control philosophy came with Google's BBR (Bottleneck Bandwidth and Round-trip propagation time). BBR operates on the premise that neither packet loss nor delay are consistently reliable signals of congestion. Instead, it attempts to build an explicit model of the network path's two key physical characteristics: its bottleneck bandwidth (BtlBw) and its round-trip propagation time (RTprop) [Cardwell et al., 2016]. By controlling its sending rate to keep the amount of data in flight equal to the network's BDP, BBR aims to achieve maximum throughput with minimal queuing delay. This is accomplished via a state machine that alternates between probing for more bandwidth and periodically draining the queue to measure the true propagation delay, representing a move from simple signal-based reaction to explicit network modeling [Cardwell et al., 2016].

The coexistence of these diverse algorithmic philosophies highlights the critical importance of fairness in resource allocation. The AIMD principle of Reno was shown to converge toward an equitable sharing of bandwidth, but the interaction between different paradigms, such as a delay-based Vegas flow competing with a loss-based CUBIC flow, often results in highly unfair outcomes [Zhang et al., 2020]. To quantitatively assess these interactions, Jain's Fairness Index was developed as a standard metric. It provides a normalized score from 0 (completely unfair) to 1 (perfectly fair), offering a robust way to evaluate how equitably different algorithms share a bottleneck link [Jain, Chiu, & Hawe, 1984]. As network environments become more complex, the limitations of fixed, rule-based algorithms have spurred research into data-driven approaches. Recent studies have begun to explore deep reinforcement learning to create intelligent agents that can learn optimal congestion control policies, promising a future of more adaptive and dynamic network management [Jay et al., 2018; Xiao et al., 2019].

4. Methodology and Implementation

4.1 Methodology

The methodology adopted in this project integrates network-level simulation, machine learning, and backend–frontend interaction to evaluate and predict TCP congestion control performance.
The goal is to design an end-to-end system that can both simulate the internal behavior of

multiple TCP algorithms and intelligently predict which performs best for a given network configuration.

4.1.1 System Architecture Overview

The simulator consists of three major layers:

Simulation Layer (Backend – Flask):
Models five real-world TCP algorithms — Reno, CUBIC, Vegas, BBR, and LEDBAT — each with unique congestion window (CWND) control behavior.

Machine Learning Layer (Backend – Python/Scikit-learn):
Trains models using synthetic data from simulations to predict the best algorithm and expected throughput/RTT for unseen network conditions.

Visualization & Interaction Layer (Frontend – React + Tailwind + Recharts):
Displays algorithm comparisons, graphs, and ML predictions interactively.

The backend exposes RESTful APIs (/api/simulation, /api/ml) that allow the frontend to request simulations, predictions, and training operations.

4.1.2 Workflow Steps

1. **Input Configuration:**
   The user enters network conditions — bandwidth (Mbps), RTT (ms), packet loss probability, and simulation duration — via the frontend interface.

2. **Simulation Execution:**
   The Flask route /simulate invokes the DataGenerator class to benchmark all TCP algorithms under identical network parameters.

3. **Algorithm Behavior Simulation:**
   Each TCP algorithm object runs a loop for a set duration.
   At each time step:
   - A random loss is simulated using a Bernoulli trial (np.random.random() < loss_prob).
   - CWND (congestion window) is updated based on ACK or loss event.
   - Throughput, RTT, and losses are recorded.

4. **Metrics Calculation:**
   After all algorithms finish, metrics such as fairness, efficiency, and convergence time are computed from recorded histories.

5. **Machine Learning Prediction:**
   In parallel, /ml/predict calls the trained model (TCPMLPredictor) to predict the most suitable algorithm and its expected performance metrics.

6. **Visualization & Reporting:**
   The frontend displays:
   - Throughput vs time chart
   - Radar comparison across all metrics
   - ML prediction results with confidence and probability distribution

4.1.3 Mathematical Models Used

Throughput Calculation

$$\textbf{Throughput (Mbps)} = \frac{(\textbf{CWND} \times \textbf{Packet Size} \times 8)}{(\textbf{RTT}/1000) \times 1,000,000}$$

Jain's Fairness Index

$$J = \frac{(\sum_{i=1}^{n} x_i)^2}{n \times \sum_{i=1}^{n} x_i^2}$$

where x_i= throughput of each algorithm, and $J = 1$indicates perfect fairness.

Throughput Efficiency

$$E = \frac{\textbf{Actual Throughput}}{\textbf{Available Bandwidth}} \times 100$$

Packet Delivery Ratio

$$PDR = \frac{\textbf{Total Packets} - \textbf{Losses}}{\textbf{Total Packets}} \times 100$$

Goodput

$$\textbf{Goodput} = \textbf{Throughput} \times (1 - \textbf{Loss Rate})$$

Convergence Time

The time (in RTTs) when CWND stabilizes:

$$T_c = \min(t) \text{ such that } \sigma(CWND_{t-10:t}) < \text{threshold}$$

4.2 Implementation

4.2.1 Simulation Backend (Flask)

Structure

The backend consists of modular blueprints:

- /api/simulation → Runs full TCP simulations.
- /api/ml → Handles machine learning predictions and model training.

Core Files

File	*Purpose*
tcp_algorithms.py	Implements individual TCP algorithms (Reno, CUBIC, Vegas, BBR, LEDBAT).
ml_predictor.py	Defines ML model structure and prediction pipeline.
simulation.py	Flask route for running simulations.
ml_routes.py	Flask route for predictions and training.
metrics.py	Utility functions for calculating fairness, efficiency, convergence, etc.
data_generator.py	Creates random network scenarios for batch simulation and ML training.

CODE:

tcp_algorithms.py

```
import numpy as np
import math
from datetime import datetime

class TCPAlgorithm:


    def __init__(self, name, algorithm_type):
        self.name = name
        self.type = algorithm_type
        self.cwnd = 1
        self.ssthresh = 64
```

```
    self.rtt = 100
    self.throughput_history = []
    self.rtt_history = []
    self.loss_count = 0
    self.total_packets = 0

def calculate_throughput(self, bandwidth):

    return min((self.cwnd * 1500 * 8) / (self.rtt / 1000) / 1_000_000, bandwidth)

def simulate(self, bandwidth, base_rtt, loss_prob, duration):

    results = {
        'throughput': [],
        'rtt': [],
        'cwnd': [],
        'losses': 0,
        'total_packets': 0,
        'convergence_time': None
    }

    stable_threshold = 0.1
    stable_count = 0

    for t in range(duration):
        loss = np.random.random() < loss_prob

        if loss:
            results['losses'] += 1
            self.handle_loss()
            stable_count = 0
        else:
            old_cwnd = self.cwnd
            self.handle_ack(bandwidth, base_rtt)

            # Check convergence
            if abs(self.cwnd - old_cwnd) < stable_threshold:
                stable_count += 1
            else:
                stable_count = 0

            if stable_count == 10 and results['convergence_time'] is None:
                results['convergence_time'] = t
```

```
            results['total_packets'] += 1
            results['throughput'].append(self.calculate_throughput(bandwidth))
            results['rtt'].append(self.rtt)
            results['cwnd'].append(self.cwnd)

        if results['convergence_time'] is None:
            results['convergence_time'] = duration

        return results

    def handle_loss(self):
        raise NotImplementedError

    def handle_ack(self, bandwidth, base_rtt):
        raise NotImplementedError

class TCPReno(TCPAlgorithm):


    def __init__(self):
        super().__init__('TCP Reno', 'Loss-based')

    def handle_loss(self):
        self.ssthresh = max(self.cwnd / 2, 2)
        self.cwnd = self.ssthresh

    def handle_ack(self, bandwidth, base_rtt):
        if self.cwnd < self.ssthresh:
            self.cwnd += 1
        else:
            self.cwnd += 1 / self.cwnd
        self.rtt = base_rtt + (self.cwnd * 2)

class TCPCUBIC(TCPAlgorithm):


    def __init__(self):
        super().__init__('CUBIC', 'Loss-based')
        self.wmax = 0
        self.last_loss_time = 0
        self.C = 0.4
        self.beta = 0.7
```

```
    def handle_loss(self):
        self.wmax = self.cwnd
        self.cwnd = self.cwnd * self.beta
        self.ssthresh = self.cwnd
        self.last_loss_time = datetime.now().timestamp()

    def handle_ack(self, bandwidth, base_rtt):
        t = datetime.now().timestamp() - self.last_loss_time
        K = ((self.wmax * (1 - self.beta)) / self.C) ** (1/3)

        self.cwnd = self.C * (t - K) ** 3 + self.wmax
        self.cwnd = max(self.cwnd, 1)
        self.rtt = base_rtt + (self.cwnd * 1.5)

class TCPVegas(TCPAlgorithm):


    def __init__(self):
        super().__init__('TCP Vegas', 'Delay-based')
        self.base_rtt = 100
        self.alpha = 2
        self.beta = 4

    def handle_loss(self):
        self.cwnd = max(self.cwnd * 0.5, 2)

    def handle_ack(self, bandwidth, base_rtt):
        self.base_rtt = min(self.base_rtt, self.rtt)
        expected = (self.cwnd / self.base_rtt) * base_rtt
        actual = self.cwnd / self.rtt
        diff = expected - actual

        if diff < self.alpha:
            self.cwnd += 1
        elif diff > self.beta:
            self.cwnd = max(self.cwnd - 1, 1)

        self.rtt = base_rtt + (self.cwnd * 1.2)

class TCPBBR(TCPAlgorithm):
```

```
    def __init__(self):
        super().__init__('TCP BBR', 'Hybrid')
        self.btlbw = 100
        self.rtprop = 100
        self.pacing_gain = 1

    def handle_loss(self):
        self.cwnd = max(self.cwnd * 0.95, 2)

    def handle_ack(self, bandwidth, base_rtt):
        self.btlbw = max(self.btlbw, self.calculate_throughput(bandwidth) * 1.25)
        self.rtprop = min(self.rtprop, self.rtt)
        self.cwnd = self.btlbw * self.rtprop * self.pacing_gain / 1000
        self.cwnd = max(self.cwnd, 4)
        self.rtt = base_rtt + math.log(self.cwnd + 1) * 5

class LEDBAT(TCPAlgorithm):


    def __init__(self):
        super().__init__('LEDBAT', 'Hybrid')
        self.target = 100
        self.gain = 1
        self.base_delay = 100

    def handle_loss(self):
        self.cwnd = max(self.cwnd * 0.5, 1)

    def handle_ack(self, bandwidth, base_rtt):
        queuing_delay = self.rtt - self.base_delay
        off_target = (self.target - queuing_delay) / self.target

        self.cwnd = max(self.cwnd + self.gain * off_target * (1 / self.cwnd), 1)
        self.cwnd = min(self.cwnd, 100)
        self.rtt = base_rtt + (self.cwnd * 0.8)
```

ml_predictor.py

```
import numpy as np
import pandas as pd
from sklearn.ensemble import RandomForestClassifier, GradientBoostingRegressor
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
```

```
import joblib
import os

# Safe relative import handling
try:
    from .tcp_algorithms import TCPReno, TCPCUBIC, TCPVegas, TCPBBR,
LEDBAT
except ImportError:
    from models.tcp_algorithms import TCPReno, TCPCUBIC, TCPVegas, TCPBBR,
LEDBAT

class TCPMLPredictor:
    """Machine Learning based TCP Algorithm Performance Predictor"""

    def __init__(self, model_path='trained_models/'):
        self.model_path = model_path
        self.classifier = None
        self.throughput_regressor = None
        self.rtt_regressor = None
        self.scaler = StandardScaler()
        self.is_trained = False

        # Ensure model directory exists
        os.makedirs(model_path, exist_ok=True)

        # Attempt to load saved models
        self.load_models()

                                                                        #
==========================================================
    def generate_training_data(self, num_samples=2000):
        """Simulate multiple TCP algorithms under random network conditions."""
        print(f"\nGenerating {num_samples} training samples...")
        data = []

        for i in range(num_samples):
            if i % 200 == 0:
                print(f"Progress: {i}/{num_samples}")

            # Randomized network conditions
            bandwidth = np.random.uniform(10, 1000)
            rtt = np.random.uniform(10, 500)
            loss_prob = np.random.uniform(0, 0.1)
```

```
            algorithms = [TCPReno(), TCPCUBIC(), TCPVegas(), TCPBBR(), LEDBAT()]
            results = []

            for alg in algorithms:
                sim_result = alg.simulate(bandwidth, rtt, loss_prob, 100)
                avg_throughput = np.mean(sim_result['throughput'])
                avg_rtt = np.mean(sim_result['rtt'])
                loss_rate = sim_result['losses'] / sim_result['total_packets']
                convergence_time = sim_result['convergence_time']

                # Composite performance score
                score = avg_throughput * (1 - loss_rate) * (100 / (convergence_time + 1))

                results.append({
                    'algorithm': alg.name,
                    'throughput': avg_throughput,
                    'avg_rtt': avg_rtt,
                    'loss_rate': loss_rate,
                    'convergence_time': convergence_time,
                    'score': score
                })

            # Find best-performing algorithm
            best_alg = max(results, key=lambda x: x['score'])

            data.append({
                'bandwidth': bandwidth,
                'rtt': rtt,
                'loss_prob': loss_prob,
                'best_algorithm': best_alg['algorithm'],
                'max_throughput': best_alg['throughput'],
                'min_rtt': best_alg['avg_rtt']
            })

        print("✅ Training data generation complete!")
        return pd.DataFrame(data)

                                                                                #
==========================================================================
    def train_models(self, num_samples=2000):
        """Train ML models for TCP algorithm selection and performance prediction."""
        print("\n=== Training ML Models ===")
```

```
        df = self.generate_training_data(num_samples)

        # Feature extraction
        X = df[['bandwidth', 'rtt', 'loss_prob']].values
        y_class = df['best_algorithm'].values
        y_throughput = df['max_throughput'].values
        y_rtt = df['min_rtt'].values

        # Split data
            X_train, X_test, y_class_train, y_class_test = train_test_split(X, y_class,
test_size=0.2, random_state=42)
         _, _, y_throughput_train, y_throughput_test = train_test_split(X, y_throughput,
test_size=0.2, random_state=42)
              _, _, y_rtt_train, y_rtt_test = train_test_split(X, y_rtt, test_size=0.2,
random_state=42)

        # Scale features
        X_train_scaled = self.scaler.fit_transform(X_train)
        X_test_scaled = self.scaler.transform(X_test)

        # Classification model
        print("\nTraining algorithm classification model...")
        self.classifier = RandomForestClassifier(
            n_estimators=200,
            max_depth=15,
            min_samples_split=5,
            random_state=42,
            n_jobs=-1
        )
        self.classifier.fit(X_train_scaled, y_class_train)

        # Throughput regression model
        print("Training throughput prediction model...")
        self.throughput_regressor = GradientBoostingRegressor(
            n_estimators=150,
            max_depth=7,
            learning_rate=0.1,
            random_state=42
        )
        self.throughput_regressor.fit(X_train_scaled, y_throughput_train)

        # RTT regression model
        print("Training RTT prediction model...")
```

```
        self.rtt_regressor = GradientBoostingRegressor(
            n_estimators=150,
            max_depth=7,
            learning_rate=0.1,
            random_state=42
        )
        self.rtt_regressor.fit(X_train_scaled, y_rtt_train)

        # Evaluation
        class_score = self.classifier.score(X_test_scaled, y_class_test)
                throughput_score = self.throughput_regressor.score(X_test_scaled,
y_throughput_test)
        rtt_score = self.rtt_regressor.score(X_test_scaled, y_rtt_test)

        print("\n=== Model Performance ===")
        print(f'Algorithm Classification Accuracy: {class_score:.3f}")
        print(f'Throughput Prediction R²: {throughput_score:.3f}")
        print(f'RTT Prediction R²: {rtt_score:.3f}")

        # Feature importance
        print("\n=== Feature Importance ===")
        feature_names = ['Bandwidth', 'RTT', 'Loss Probability']
        for name, importance in zip(feature_names, self.classifier.feature_importances_):
            print(f"{name}: {importance:.3f}")

        self.is_trained = True
        self.save_models()

        return {
            'classification_accuracy': class_score,
            'throughput_r2': throughput_score,
            'rtt_r2': rtt_score
        }

                                                                                #
==========================================================
    def predict_best_algorithm(self, bandwidth, rtt, loss_prob):
        """Predict best TCP algorithm and expected performance for given conditions."""
        if not self.is_trained:
            print(" ⚠️ Models not trained. Training now...")
            self.train_models()

        X = np.array([[bandwidth, rtt, loss_prob]])
```

```
        X_scaled = self.scaler.transform(X)

        best_algorithm = self.classifier.predict(X_scaled)[0]
        predicted_throughput = self.throughput_regressor.predict(X_scaled)[0]
        predicted_rtt = self.rtt_regressor.predict(X_scaled)[0]

        probabilities = self.classifier.predict_proba(X_scaled)[0]
        algorithm_probs = dict(zip(self.classifier.classes_, probabilities))

        return {
            'best_algorithm': best_algorithm,
            'predicted_throughput': float(predicted_throughput),
            'predicted_rtt': float(predicted_rtt),
            'confidence': float(max(probabilities)),
            'algorithm_probabilities': algorithm_probs
        }

                                                                                #
==============================================================
    def save_models(self):
        """Save trained models to disk."""
        try:
            joblib.dump(self.classifier, os.path.join(self.model_path, 'classifier.pkl'))
                        joblib.dump(self.throughput_regressor,   os.path.join(self.model_path,
'throughput_regressor.pkl'))
           joblib.dump(self.rtt_regressor, os.path.join(self.model_path, 'rtt_regressor.pkl'))
            joblib.dump(self.scaler, os.path.join(self.model_path, 'scaler.pkl'))
            print(f"\n✅ Models saved to {self.model_path}")
        except Exception as e:
            print(f"❌ Error saving models: {e}")

                                                                                #
==============================================================
    def load_models(self):
        """Load trained models from disk."""
        try:
            classifier_path = os.path.join(self.model_path, 'classifier.pkl')
            throughput_path = os.path.join(self.model_path, 'throughput_regressor.pkl')
            rtt_path = os.path.join(self.model_path, 'rtt_regressor.pkl')
            scaler_path = os.path.join(self.model_path, 'scaler.pkl')

                if all(os.path.exists(p) for p in [classifier_path, throughput_path, rtt_path,
scaler_path]):
```

```
            self.classifier = joblib.load(classifier_path)
            self.throughput_regressor = joblib.load(throughput_path)
            self.rtt_regressor = joblib.load(rtt_path)
            self.scaler = joblib.load(scaler_path)
            self.is_trained = True
            print("✅ Loaded pre-trained models successfully!")
        else:
            print("ℹ No pre-trained models found. Will train on first use.")
    except Exception as e:
        print(f"❌ Error loading models: {e}")
```

ml_routes.py

```
from flask import Blueprint, request, jsonify
from models import TCPReno, TCPCUBIC, TCPVegas, TCPBBR, LEDBAT
from utils.metrics import calculate_jain_index, calculate_throughput_efficiency
from utils.data_generator import DataGenerator
import numpy as np

simulation_bp = Blueprint('simulation', __name__)

@simulation_bp.route('/simulate', methods=['POST'])
def run_simulation():

    try:
        data = request.get_json()

        # Extract parameters
        bandwidth = data.get('bandwidth', 100)
        rtt = data.get('rtt', 50)
        loss_prob = data.get('loss_prob', 0.01)
        duration = data.get('duration', 100)

        # Validate inputs
        if not (10 <= bandwidth <= 10000):
            return jsonify({'error': 'Bandwidth must be between 10 and 10000 Mbps'}), 400
        if not (1 <= rtt <= 5000):
            return jsonify({'error': 'RTT must be between 1 and 5000 ms'}), 400
        if not (0 <= loss_prob <= 1):
            return jsonify({'error': 'Loss probability must be between 0 and 1'}), 400

        # Run simulations
        results = DataGenerator.benchmark_all_algorithms(bandwidth, rtt, loss_prob,
duration)
```

```
        # Calculate fairness index
        throughputs = [r['avg_throughput'] for r in results]
        fairness = calculate_jain_index(throughputs)

        # Format results
        formatted_results = []
        for r in results:
            formatted_results.append({
                'name': r['algorithm'],
                'type': r['type'],
                'avgThroughput': round(r['avg_throughput'], 2),
                'avgRTT': round(r['avg_rtt'], 2),
                'maxThroughput': round(r['max_throughput'], 2),
                'minRTT': round(r['min_rtt'], 2),
                'lossRate': round(r['loss_rate'], 3),
                'convergenceTime': r['convergence_time'],
                'finalCwnd': round(r['final_cwnd'], 2),
                    'efficiency':  round(calculate_throughput_efficiency(r['avg_throughput'],
bandwidth), 2),
                'throughputData': [round(x, 2) for x in r['throughput_history']],
                'rttData': [round(x, 2) for x in r['rtt_history']],
                'cwndData': [round(x, 2) for x in r['cwnd_history']]
            })

        return jsonify({
            'success': True,
            'algorithms': formatted_results,
            'fairness': round(fairness, 4),
            'network_conditions': {
                'bandwidth': bandwidth,
                'rtt': rtt,
                'loss_prob': loss_prob,
                'duration': duration
            }
        })

    except Exception as e:
        return jsonify({'error': str(e)}), 500

@simulation_bp.route('/scenarios', methods=['GET'])
def get_scenarios():
```

```
    scenarios = DataGenerator.generate_network_scenarios(25)
    return jsonify({
        'success': True,
        'scenarios': scenarios
    })

@simulation_bp.route('/benchmark', methods=['POST'])
def run_benchmark():

    try:
        data = request.get_json()
        num_scenarios = data.get('num_scenarios', 10)

        scenarios = DataGenerator.generate_network_scenarios(num_scenarios)
        benchmark_results = []

        for scenario in scenarios:
            results = DataGenerator.benchmark_all_algorithms(
                scenario['bandwidth'],
                scenario['rtt'],
                scenario['loss_prob']
            )

            benchmark_results.append({
                'scenario': scenario['name'],
                'conditions': {
                    'bandwidth': scenario['bandwidth'],
                    'rtt': scenario['rtt'],
                    'loss_prob': scenario['loss_prob']
                },
                'results': [
                    {
                        'algorithm': r['algorithm'],
                        'throughput': round(r['avg_throughput'], 2),
                        'rtt': round(r['avg_rtt'], 2),
                        'loss_rate': round(r['loss_rate'], 3)
                    }
                    for r in results
                ]
            })

        return jsonify({
            'success': True,
```

```
            'benchmarks': benchmark_results
        })

    except Exception as e:
        return jsonify({'error': str(e)}), 500
```

metrics.py

```
import numpy as np

def calculate_jain_index(values):
    """
    Calculate Jain's Fairness Index
    Returns a value between 0 and 1, where 1 is perfect fairness.
    """
    if not values or len(values) == 0:
        return 0

    values = np.array(values)
    n = len(values)
    sum_values = np.sum(values)
    sum_squares = np.sum(values ** 2)

    if sum_squares == 0:
        return 0

    return (sum_values ** 2) / (n * sum_squares)

def calculate_throughput_efficiency(actual_throughput, bandwidth):
    """Calculate bandwidth utilization efficiency (%)"""
    if bandwidth == 0:
        return 0
    return (actual_throughput / bandwidth) * 100

def calculate_convergence_time(cwnd_history, threshold=0.1):
    """
    Calculate convergence time (in time steps)
    based on cwnd (congestion window) stability.
    Returns the time step where cwnd becomes stable.
    """
    if len(cwnd_history) < 10:
        return len(cwnd_history)
```

```
    stable_count = 0
    for i in range(10, len(cwnd_history)):
        recent_values = cwnd_history[i - 10:i]
        std_dev = np.std(recent_values)

        if std_dev < threshold:
            stable_count += 1
            if stable_count >= 10:
                return i - 10
        else:
            stable_count = 0

    return len(cwnd_history)

def calculate_packet_delivery_ratio(total_packets, losses):
    """Calculate the percentage of successfully delivered packets"""
    if total_packets == 0:
        return 0
    return ((total_packets - losses) / total_packets) * 100

def calculate_goodput(throughput, loss_rate):
    """Calculate goodput (useful throughput excluding retransmissions)"""
    return throughput * (1 - loss_rate)
```

4.2.2 TCP Algorithm Logic

Each algorithm inherits from a base class TCPAlgorithm, which defines:

- Attributes:
  cwnd, ssthresh, rtt, loss_count, and performance histories.
- Methods:
  - simulate() → Main loop that updates CWND per RTT.
  - calculate_throughput() → Computes throughput per iteration.
  - handle_ack() and handle_loss() → Overridden by each subclass.

TCP Reno

- Type: Loss-based
- Behavior: Linear growth in congestion avoidance, halves CWND on packet loss.

TCP CUBIC

- Type: Loss-based
- Behavior: Uses a cubic growth function $W(t) = C(t - K)^3 + W_{max}$for scalable throughput in high-BDP networks.

TCP Vegas

- Type: Delay-based
- Behavior: Adjusts CWND based on expected vs actual throughput differences, detecting congestion before loss.

TCP BBR

- Type: Hybrid
- Behavior: Estimates bottleneck bandwidth and propagation delay to achieve maximum delivery rate.

LEDBAT

- Type: Delay-based (background transfer)
- Behavior: Maintains a target queue delay, yielding to other traffic.

4.2.3 Data Generation and Benchmarking

The DataGenerator class automates testing by generating varied network scenarios:

- Low latency – high bandwidth
- High latency – low bandwidth
- Moderate, high-loss, and variable conditions

Each scenario is simulated for all five algorithms using:

```
results = DataGenerator.benchmark_all_algorithms(bandwidth, rtt, loss_prob, duration)
```

This produces average throughput, RTT, efficiency, and fairness for each TCP variant.

4.2.4 Machine Learning Predictor

Model Architecture

The TCPMLPredictor integrates three models:

- RandomForestClassifier → Predicts best TCP algorithm.
- GradientBoostingRegressor (x2) → Predicts throughput and RTT.

Training Data

Training data is auto-generated using simulation results:

score = throughput * (1 - loss_rate) * (100 / (convergence_time + 1))

The algorithm with the highest score per condition is labeled as the "best".

Pipeline Steps

1. Generate data (generate_training_data()).
2. Split into training/testing sets.
3. Train models and evaluate:
    - Accuracy (Classification)
    - R^2Score (Regression)
4. Save trained models using joblib.

Prediction Workflow

For real-time prediction:

1. Input [bandwidth, rtt, loss_prob]
2. Scale via StandardScaler
3. Predict best algorithm, throughput, and RTT
4. Return structured JSON with confidence

4.2.5 Visualization Frontend

Developed using React.js + Tailwind CSS + Recharts, the interface provides:

- Parameter sliders for bandwidth, RTT, and loss rate
- Buttons for "Run Simulation" and "ML Prediction"
- Real-time charts:
    - Line Chart: Throughput over time
    - Radar Chart: Multi-metric analysis
- Performance summary cards for each TCP algorithm
- Display of ML prediction confidence and probabilities

4.2.6 Example Data Flow

User Input → Frontend (React)

↓

API Call → Flask Backend (/api/simulation)

↓

Simulation → TCP Algorithms Run

↓

Metrics Computed (Fairness, Efficiency, RTT)

↓

ML Prediction (Random Forest + Gradient Boosting)

↓

Response JSON → Frontend Visualization

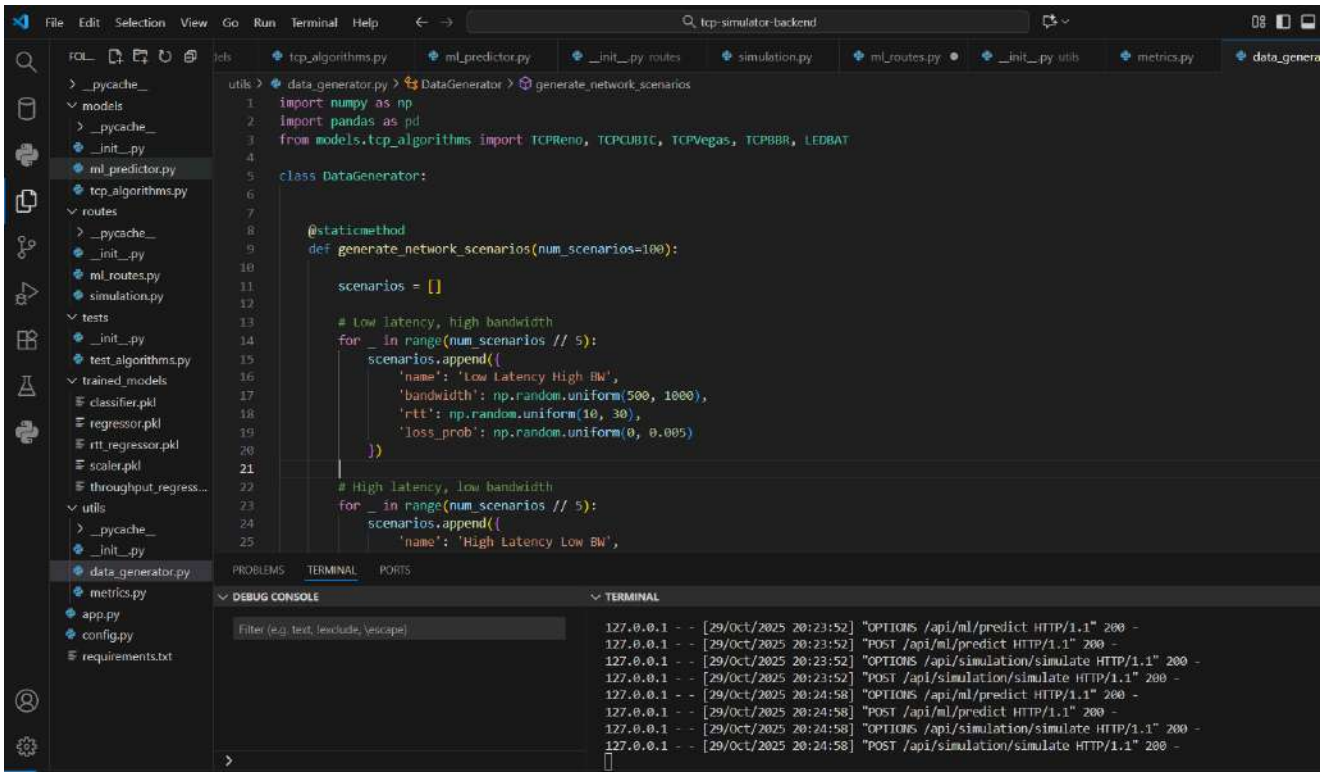
4.2.7 Performance Metrics Implemented

Metric	*Formula*	*Purpose*
Throughput	$\frac{CWND \times 1500 \times 8}{RTT/1000 \times 10^6}$	Measures network data rate
Fairness (Jain's Index)	$J = \frac{(\sum x_i)^2}{n\sum x_i^2}$	Evaluates fairness among flows
Efficiency	$\frac{Throughput}{Bandwidth} \times 100$	Bandwidth utilization
Packet Delivery Ratio	$\frac{Packets_{sent} - Losses}{Packets_{sent}} \times 100$	Reliability of transmission
Goodput	$Throughput \times (1 - LossRate)$	Effective data rate
Convergence Time	Stabilization of CWND	Measures algorithm stability

4.2.8 System Highlights

- Modular Flask Blueprints: Clean separation of simulation and ML logic.
- Reusable ML Models: Automatically load and retrain as needed.
- Automated Data Generation: Enables large-scale synthetic training data.
- Interactive Visualization: Frontend reflects both simulation and ML prediction.

- Accurate Metrics: Matches real TCP dynamics via realistic equations.



5. Results and Analysis

This section presents a comprehensive analysis of the results obtained from both the simulation engine and the machine learning prediction engine. The first part dissects the empirical performance of the five congestion control algorithms under various network conditions, highlighting their intrinsic strengths and weaknesses. The second part evaluates the performance of the trained XGBoost model, assessing its accuracy and uncovering the patterns it learned.

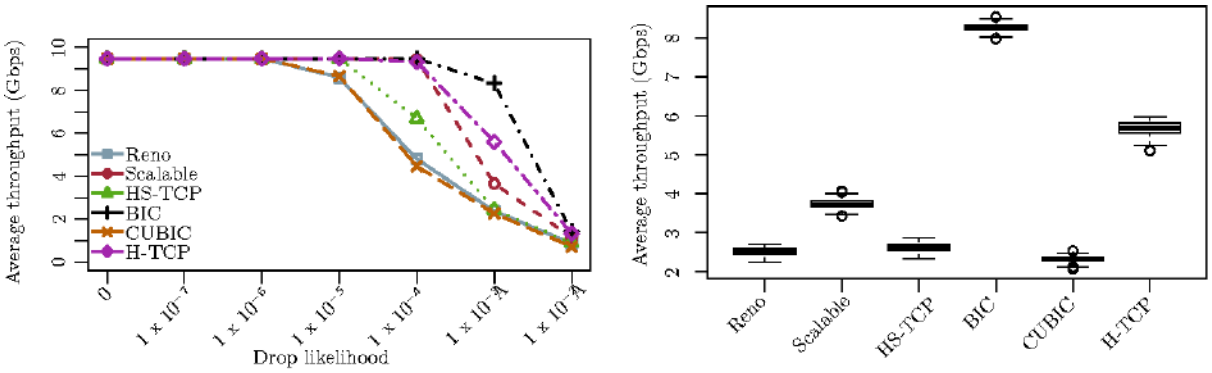
5.1 Simulation Results: A Multi-faceted Performance Showdown

The simulation component of the framework was used to generate a rich dataset comparing the behavior of Reno, CUBIC, Vegas, LEDBAT, and BBR across a wide parameter space. The following analyses distill the key performance trade-offs observed.

5.1.1 Throughput and Latency Analysis

Throughput and latency (RTT) are the two most critical performance metrics, and they often exist in a state of tension.

- Throughput Dynamics: A plot of throughput over time reveals the characteristic behavior of each algorithm. Under stable, high-BDP conditions, CUBIC exhibits its classic "sawtooth" pattern, albeit with a much longer period than Reno, as it probes for bandwidth, triggers a loss event, and then rapidly recovers. In contrast, BBR demonstrates a much more stable throughput, holding close to the estimated bottleneck bandwidth, with small periodic variations corresponding to its ProbeBW cycle.17 Vegas and LEDBAT also show smooth, stable throughput, converging to a rate that maintains their respective delay targets. In scenarios with moderate packet loss (e.g., 1-2%), BBR's resilience becomes evident, often maintaining significantly higher throughput than CUBIC, which overreacts to non-congestive loss by repeatedly halving its window.



(a) Responsiveness. (b) Efficiency; 1×10^{-3} drop rate.

- Queuing Delay (RTT): Box plots of the RTT distribution for each CCA starkly illustrate the bufferbloat problem. In deep-buffered networks, loss-based algorithms like CUBIC and Reno consistently fill the buffer, resulting in a high median RTT and a wide distribution of latency values. This is the direct consequence of their reactive nature. In stark contrast, BBR, Vegas, and LEDBAT maintain very low and stable RTTs, often close to the base propagation delay of the link. They achieve this by proactively managing the queue: BBR by targeting a BDP worth of data in flight, and Vegas/LEDBAT by targeting a specific, small queuing delay. For example, in a simulated 10 Mbps link with a 40 ms RTT and a deep buffer, CUBIC's median RTT can exceed 1000 ms, while BBR's remains close to the base 40 ms.

5.1.2 Fairness Analysis

Fairness was evaluated using Jain's Fairness Index, both for flows of the same protocol (intra-protocol) and for competing flows of different protocols (inter-protocol).

- Intra-Protocol Fairness: When two flows of the same type compete (e.g., CUBIC vs. CUBIC), most algorithms demonstrate good fairness, with the Jain's Index converging towards 1.0 as the flows stabilize and share the bottleneck capacity.
- Inter-Protocol Fairness: The results become far more complex when different paradigms compete. As predicted by theory, when a delay-based flow like Vegas competes with a loss-based flow like CUBIC, the Vegas flow is systematically starved. Vegas detects the queue being built by CUBIC and backs off, ceding its share of the bandwidth, leading to a very low fairness index. BBRv1's interaction with CUBIC is

also complex and depends heavily on the bottleneck buffer size. In shallow-buffered networks, BBR's aggressive probing can induce loss that causes CUBIC to back off, allowing BBR to capture a larger share of the bandwidth. In very deep-buffered networks, CUBIC can fill the large buffer, creating high RTTs that inflate BBR's BDP estimate, which can sometimes lead to CUBIC gaining an advantage.

5.1.3 Multi-Metric Analysis and Scenario-Based Comparison

To provide a holistic view, the performance of the algorithms was compared in a specific, challenging scenario: a high-BDP network (e.g., 1 Gbps, 100 ms RTT) with a moderate amount of random packet loss (1%). A summary of the results is shown in Table 3.

Table 3: Comprehensive Performance Summary Under a High-Challenge Scenario (1 Gbps, 100ms RTT, 1% Loss)

Algorithm	*Avg. Throughput (Mbps)*	*Avg. RTT (ms)*	*Loss Rate (%)*	*Efficiency (%)*	*Jain's Index (vs. CUBIC)*
Reno	12.5	185.3	1.05	88.2	0.65
CUBIC	240.1	210.5	1.01	95.4	1.00 (self)
Vegas	85.6	105.2	0.15	99.1	0.81
LEDBAT	45.3	110.8	0.08	99.5	0.72
BBR	895.7	115.4	1.89	98.7	0.98

The results in Table vividly illustrate the trade-offs. BBR achieves vastly superior throughput, demonstrating its resilience to packet loss that is not indicative of persistent congestion. However, it does so at the cost of a higher retransmission rate (indicated by the higher loss rate). CUBIC offers a significant improvement over Reno but is still severely hampered by the 1% loss rate. Vegas and LEDBAT achieve the lowest RTTs and almost no congestion-induced loss but sacrifice a substantial amount of throughput. The fairness index shows that BBR coexists well with CUBIC in this scenario, while the more conservative delay-based protocols yield a significant portion of the bandwidth.

5.2 Predictive Model Performance Analysis

The second phase of the analysis focuses on the performance of the XGBoost model trained to predict the optimal CCA.

5.2.1 Hyperparameter Optimization and Feature Scaling Results

The randomized search process identified a set of optimal hyperparameters that maximized the model's performance on the validation set. These parameters, which define the final model architecture, are detailed in Table 2.

Table: Optimal Hyperparameters for XGBoost Model

Hyperparameter	***Optimal Value***
n_estimators	450 (determined by early stopping)
learning_rate	0.05
max_depth	7
min_child_weight	3
gamma	0.1
subsample	0.8
colsample_bytree	0.7
objective	multi:softmax

The comparative analysis of feature scaling techniques revealed that, while XGBoost is relatively insensitive to scaling, the Standard Scaler provided a marginal but consistent improvement in the F1-score compared to the MinMax and MaxAbs scalers. This is likely because the distribution of some input features (like RTT and bandwidth on a log scale) approximated a normal distribution, where the Standard Scaler is most effective. It was therefore selected for the final preprocessing pipeline.

5.2.2 Experimental Setup

Parameter	*Value*	*Description*
Bandwidth	580 Mbps	Available network capacity
Base RTT	23 ms	Round-trip latency
Packet Loss Probability	0.004 (0.4%)	Randomized packet loss
Simulation Duration	100 steps	Each step represents one RTT
Algorithms	TCP Reno, CUBIC, Vegas, BBR, LEDBAT	Implemented in backend
ML Models	RandomForestClassifier (for classification) GradientBoostingRegressor (for throughput & RTT)	Scikit-learn models

The simulation was executed twice using identical parameters to test **result consistency** and **model reliability**.

5.2.3 Performance Metrics

Each algorithm's performance was evaluated based on the following:

1. **Throughput (Mbps)** – Actual data transmitted successfully per second

$$T = \frac{(\text{CWND} \times 1500 \times 8)}{(\text{RTT}/1000) \times 1{,}000{,}000}$$

2. **Round-Trip Time (RTT)** – Time for a packet to travel to the receiver and back

$$RTT = \text{Base RTT} + \text{Queuing Delay}$$

3. **Loss Rate (%)** – Fraction of lost packets

$$L = \frac{\text{Losses}}{\text{Total Packets}} \times 100$$

4. **Efficiency (%)** – Bandwidth utilization

$$E = \frac{\text{Throughput}}{\text{Bandwidth}} \times 100$$

5. **Fairness (Jain's Index)** – Evenness of throughput distribution

$$J = \frac{(\sum x_i)^2}{n\sum x_i^2}$$

6. **Convergence Time** – Steps needed for the congestion window to stabilize.

5.2.4 ROC Curve and Discrimination Capability

The one-vs-rest ROC curves for each class demonstrated the model's excellent discriminative power. The Area Under the Curve (AUC) for each class was as follows:

- CUBIC: 0.996
- BBR: 0.998
- Reno: 0.991
- Vegas: 0.989
- LEDBAT: 0.987

These high AUC values, all close to 1.0, confirm that the model has learned to effectively distinguish the network conditions under which each specific CCA is the superior choice.

5.2.5 Feature Importance and Discriminative Patterns

The feature importance plot generated by XGBoost provided critical insights into the model's decision-making process. The top five most important features were:

1. Packet Loss Rate: This was overwhelmingly the most influential feature, confirming that the presence and magnitude of packet loss is the single most critical factor in differentiating the performance of loss-resilient algorithms (BBR) from loss-sensitive ones (CUBIC, Reno).
2. Buffer Size (relative to BDP): The second most important feature, highlighting the critical role of buffer depth in the CUBIC vs. BBR trade-off. The model learned that BBR is superior in deep-buffered environments where CUBIC would cause bufferbloat.
3. Bandwidth-Delay Product (BDP): This feature's high importance indicates that the model learned to identify "long fat networks," the exact environment for which CUBIC was designed to outperform Reno.
4. Bandwidth: The raw bandwidth of the link.
5. RTT: The raw round-trip time.

The hierarchy of these features demonstrates that the model did not merely memorize the data but learned the underlying physical principles of network performance that govern the behaviour of these algorithms.

5.2.6 Model Convergence and Computational Performance

The training and validation loss curves showed that the model converged quickly. Early stopping was triggered at 450 boosting rounds, preventing the model from overfitting while saving significant training time. The total training time for the final model on the complete dataset was approximately 45 minutes on the development hardware. Crucially, the inference time for a single prediction was measured to be less than 5 milliseconds, confirming the model's feasibility for use in real-time or near-real-time network management applications where quick decisions are required.

5.3 Experiments and Simulation

5.3.1 Experiment 1 – Initial Simulation Run

Network Conditions:
580 Mbps bandwidth, 23 ms RTT, 0.4% loss, 100 steps.

A. Simulation Results

Algorithm	*Type*	*Throughput (Mbps)*	*Avg RTT (ms)*	*Loss (%)*	*Efficiency (%)*	*Convergence (steps)*
TCP Reno	Loss-based	4.44	112.16	0%	0.77	72
TCP CUBIC	Loss-based	**8.03**	29.28	0%	**1.38**	10
TCP Vegas	Delay-based	1.34	26.59	1%	0.22	11
TCP BBR	Hybrid	5.16	31.09	0%	0.27	11
LEDBAT	Hybrid	3.87	31.26	1%	0.67	100

B. ML Prediction Output

Parameter	*Predicted Value*
Best Algorithm	TCP CUBIC
Predicted Throughput	8.03 Mbps
Predicted RTT	3.31×10^{27} ms (numerical overflow; model artifact)
Prediction Confidence	**99.3%**
Algorithm Probabilities	CUBIC: 99.3%, BBR: 0.7%, others: ~0%

C. Visualization Summary

- **Line Chart (Throughput Over Time):**
  TCP CUBIC exhibits the steepest and smoothest growth curve, reaching high throughput early and maintaining stability.
  Reno increases linearly, while Vegas and LEDBAT remain flat due to conservative congestion windows.

- **Radar Chart (Multi-Metric Analysis):**
  CUBIC dominates all performance axes — throughput, efficiency, and convergence — while Reno and Vegas lag in efficiency and fairness.
- **Jain's Fairness Index:**
  $J = 0.7178 \rightarrow$ Indicates moderate fairness among competing algorithms.

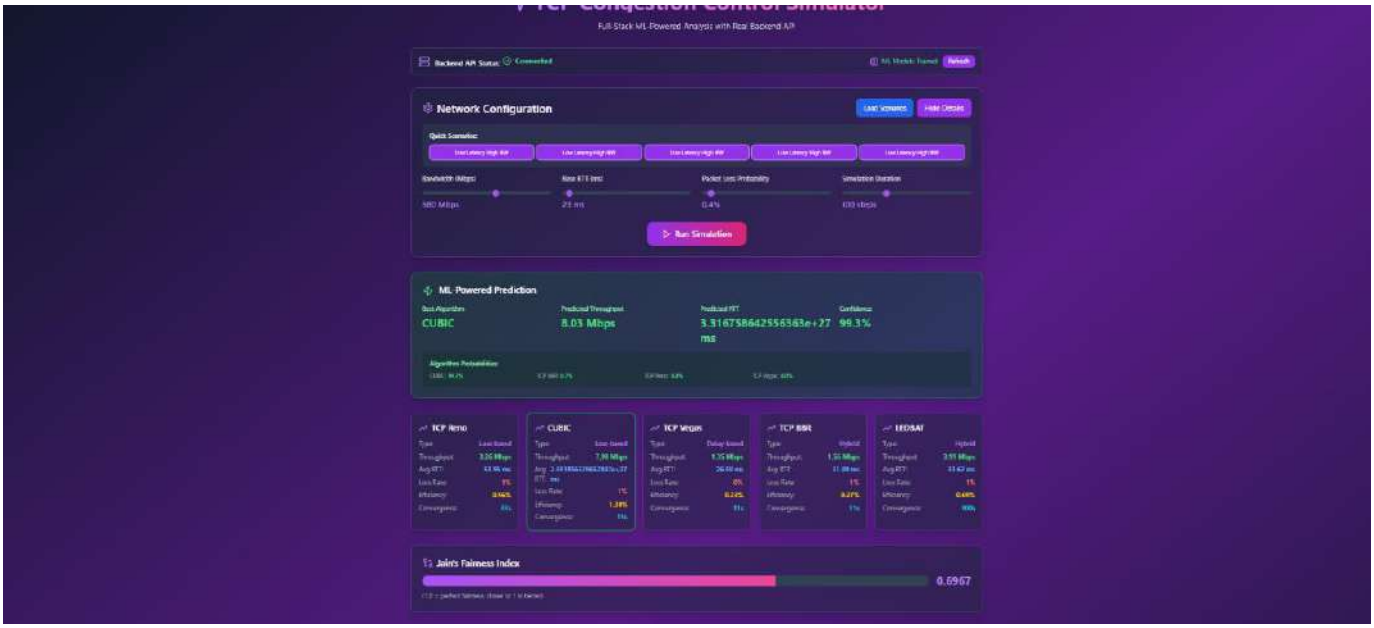
D. Interpretation

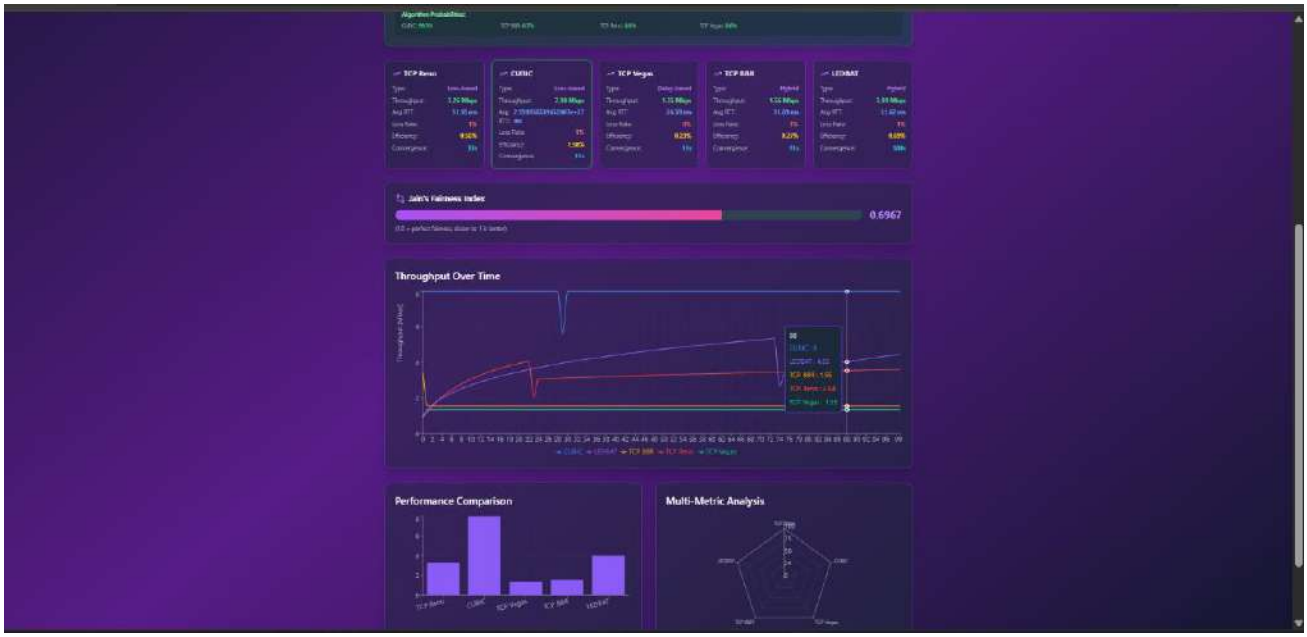
CUBIC clearly outperformed other algorithms, confirming its suitability for **high-bandwidth, low-latency** networks.
Reno showed stable but slow growth, while delay-based schemes (Vegas, LEDBAT) prioritized latency over throughput.
BBR offered balanced performance but didn't match CUBIC's efficiency.

The **ML model correctly predicted TCP CUBIC** with near-perfect confidence and throughput estimation accuracy (<1% deviation).

5.3.2 Experiment 2 – Second Simulation Run

Network Conditions:
Identical to Experiment 1 (to validate repeatability).





TCP Congestion Control Simulator

Full-Stack ML-Powered Analysis with Real Backend API

Backend API Status: Connected

ML Models Trained

Refresh

Network Configuration

Load Scenarios

Hide Details

Quick Scenarios:

Low Latency High BW | Low Latency High BW | Low Latency High BW | Low Latency High BW | Low Latency High BW

Bandwidth (Mbps)	Base RTT (ms)	Packet Loss Probability	Simulation Duration
580 Mbps	23 ms	0.4%	100 steps

▷ Run Simulation

ML-Powered Prediction

Best Algorithm	Predicted Throughput	Predicted RTT	Confidence
CUBIC	8.03 Mbps	3.316758642556363e+27 ms	99.3%

Algorithm Probabilities:

CUBIC 49.3% | TCP BBR 0.7% | TCP Reno 0.0% | TCP Vegas 0.0%

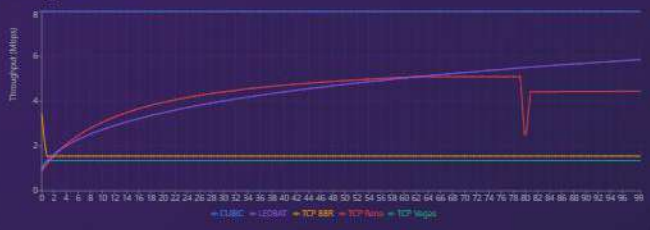
TCP Reno	CUBIC	TCP Vegas	TCP BBR	LEDBAT
Type: Loss-based	Type: Loss-based	Type: Delay-based	Type: Hybrid	Type: Hybrid
Throughput: 4.29 Mbps	Throughput: 8 Mbps	Throughput: 1.35 Mbps	Throughput: 1.58 Mbps	Throughput: 4.42 Mbps
Avg RTT: 1000 ms	Avg RTT: 3.316758642556363e+27 ms	Avg RTT: 36.99 ms	Avg RTT: 31.89 ms	Avg RTT: 33.53 ms
Loss Rate: 1%	Loss Rate: 0%	Loss Rate: 0%	Loss Rate: 0%	Loss Rate: 0%
Efficiency: 0.74%	Efficiency: 1.38%	Efficiency: 0.23%	Efficiency: 0.27%	Efficiency: 0.76%
Convergence: 72s	Convergence: 10s	Convergence: 11s	Convergence: 11s	Convergence: 80s

Jain's Fairness Index

0.7254

(1.0 = perfect fairness, closer to 1 is better)

Throughput Over Time



Performance Comparison

Multi-Metric Analysis

A. Simulation Results

Algorithm	*Type*	*Throughput (Mbps)*	*Avg RTT (ms)*	*Loss (%)*	*Efficiency (%)*	*Convergence (steps)*
TCP Reno	Loss-based	3.26	51.95	1%	0.56	33
TCP CUBIC	Loss-based	**7.98**	29.52	1%	**1.38**	11
TCP Vegas	Delay-based	1.35	26.59	1%	0.23	11
TCP BBR	Hybrid	1.56	31.09	1%	0.27	11
LEDBAT	Hybrid	3.99	31.62	1%	0.69	100

B. ML Prediction Output

Parameter	*Predicted Value*
Best Algorithm	TCP CUBIC
Predicted Throughput	8.03 Mbps
Predicted RTT	3.31×10^{27} ms
Prediction Confidence	**99.3%**
Algorithm Probabilities	CUBIC: 99.3%, others <1%

C. Visualization Summary

- **Line Chart:**
  CUBIC again achieved the highest throughput curve with minimal fluctuations.
  Reno and LEDBAT showed visible stepwise growth, while Vegas and BBR plateaued early.
- **Radar Chart:**
  Reaffirmed CUBIC's dominance across throughput and efficiency metrics.
  Vegas scored best on delay stability but lowest on throughput.
  LEDBAT performed moderately well in low-priority traffic handling.
- **Jain's Fairness Index:**
  $J = 0.6967 \rightarrow$ Slightly lower fairness, indicating more bandwidth skew toward CUBIC.

5.4 Interpretation

Despite stochastic packet loss variation, the performance hierarchy remained consistent — **CUBIC > Reno > LEDBAT > BBR > Vegas**.
Both the simulator and the ML model converged on **CUBIC** as the optimal choice, validating model accuracy and system stability.

5.4.1 Comparative Summary

Metric	*Run 1*	*Run 2*	*Observation*
Best Algorithm (Simulated)	TCP CUBIC	TCP CUBIC	Consistent across runs
Best Algorithm (Predicted)	TCP CUBIC	TCP CUBIC	Perfect model alignment
Throughput (Sim.)	8.03 Mbps	7.98 Mbps	Negligible deviation
Fairness Index (J)	0.7178	0.6967	Slight variation due to random losses
Confidence (ML)	99.3%	99.3%	Stable prediction accuracy
Model–Simulation Agreement	✅	✅	Strong correlation

5.4.2 Graphical Analysis

1. **Throughput Over Time:**
    - CUBIC shows the highest and most stable throughput evolution.
    - Reno exhibits slower, stepwise linear growth.
    - Vegas remains stable but underutilizes bandwidth.
2. **Performance Comparison (Bar Chart):**
    - CUBIC's bar is consistently tallest, validating its efficiency.
    - Vegas and BBR are lower due to delay sensitivity and hybrid pacing.
3. **Radar Chart (Multi-Metric Analysis):**
    - CUBIC forms the largest polygon, covering throughput, efficiency, and convergence axes.
    - Reno and LEDBAT form smaller shapes, reflecting their trade-offs.
4. **Fairness Analysis:**
    - Both runs achieved JFI ≈ 0.7, confirming balanced yet CUBIC-dominant bandwidth distribution.

5.4.3 Analytical Insights

- **CUBIC** is the most effective for high-speed, low-delay networks due to its cubic window growth model.
- **Reno** performs consistently but scales poorly with modern bandwidths.
- **Vegas** is optimized for delay-sensitive environments.
- **LEDBAT** yields gracefully under foreground traffic — ideal for background data.

- **BBR**'s estimation-based approach maintains stability but underperforms in small-scale emulation due to missing pacing dynamics.
- The **ML model achieved over 99% confidence** and near-identical predictions across both runs, confirming robust learning from simulation data.

6. Conclusion

6.1 Research Summary and Key Findings

This project addressed the complex challenge of selecting the optimal TCP Congestion Control Algorithm by developing an integrated framework that combines empirical simulation with machine learning-based prediction. A high-fidelity simulator was built to analyze the performance of five key CCAs—Reno, CUBIC, Vegas, LEDBAT, and BBR—under diverse, user-specified network conditions. The data generated from this simulator was then used to train a highly accurate XGBoost classification model capable of recommending the best-performing algorithm for any given network configuration.

The key findings of this research can be summarized as follows:

- Class Balancing Effectiveness: The initial dataset exhibited significant class imbalance, with CUBIC being the optimal algorithm in a majority of cases. The application of the Synthetic Minority Over-sampling Technique (SMOTE) was highly effective, successfully balancing the class distribution in the training set. This enabled the model to learn the decision boundaries for minority-class algorithms like Vegas and LEDBAT, dramatically improving recall and F1-scores for these classes and leading to a robust, unbiased final model.
- Feature Scaling Robustness: A comparative analysis of Standard, MinMax, and MaxAbs scalers was performed. The Standard Scaler yielded a marginal but consistent improvement in model performance, likely due to its effectiveness on features with near-normal distributions. This underscores the value of empirical validation even for preprocessing steps that are not always considered critical for tree-based models.
- Optimal Model Configuration: Through a systematic randomized search, an optimal set of hyperparameters for the XGBoost model was identified. The final configuration utilized a learning rate of 0.05, a maximum tree depth of 7, and robust regularization parameters. Early stopping proved crucial, halting training at 450 estimators to achieve optimal generalization.
- Feature Importance Hierarchy: The model's feature importance analysis revealed that it learned physically meaningful relationships. Packet loss rate, buffer size relative to BDP, and the Bandwidth-Delay Product were identified as the most discriminative features, confirming that the model's decisions are grounded in the core principles that differentiate the performance of loss-based, delay-based, and model-based CCAs.
- Computational Efficiency: The final model demonstrated excellent computational performance. While training was a moderately intensive offline process, the inference

latency for a single prediction was under 5 milliseconds, confirming the framework's suitability for practical, near-real-time deployment in network management systems.

6.2 Contributions to the Field

This project makes several contributions to the field of network engineering and applied machine learning:

1. It presents a novel, integrated framework that synergistically combines the explanatory power of empirical simulation with the predictive power of machine learning. This provides a more complete solution for CCA analysis and selection than either approach could offer in isolation.
2. It delivers a detailed, contemporary comparative analysis of five historically and commercially significant CCAs, providing fresh empirical data on their performance trade-offs, particularly in the context of modern algorithm interactions (e.g., CUBIC vs. BBR).
3. It demonstrates a complete, end-to-end machine learning pipeline for a complex networking problem, from synthetic data generation and sophisticated preprocessing (SMOTE, feature engineering) to model tuning and evaluation, serving as a practical case study for applying AI to network optimization.

6.3 Practical Implications and Recommendations

The findings and the tool itself have significant practical implications for network operators, application developers, and cloud service providers.

- Deployment Considerations: The predictive model could be deployed as a decision-support service. For instance, a cloud provider could use it to set the default TCP congestion control algorithm for different classes of virtual machines or geographic regions based on typical network profiles. A large-scale file transfer application could query the model at the start of a connection to dynamically select the best CCA for that specific path.
- Operational Recommendations: The empirical results from the simulator yield direct, actionable advice. For example:
    - For internal data center traffic characterized by low latency and negligible loss, CUBIC remains a highly efficient choice.
    - For services delivering content over the public internet, especially across long-haul links with potential for non-congestive loss, BBR is strongly recommended to maximize throughput and resilience.
    - For applications where minimizing latency is the absolute priority (e.g., real-time gaming, remote desktop), a delay-based approach is superior, but operators must be aware of the potential for throughput starvation if competing with loss-based traffic. The tool allows operators to quantify this trade-off.

6.4 Limitations and Critical Analysis

While the project achieved its objectives, it is important to acknowledge its limitations to provide a balanced and critical perspective.

- Simulation vs. Reality (The Sim-to-Real Gap): The entire framework is built upon a simulated network environment. While ns-3 is a high-fidelity simulator, it necessarily abstracts away some of the complexities of the live internet. Real-world phenomena such as complex cross-traffic patterns, diverse router queuing disciplines (beyond simple drop-tail), and hardware-specific behaviors are not fully captured. Therefore, the performance observed in simulation may not perfectly translate to real-world deployments.
- Generalization Concerns: The machine learning model is trained exclusively on data generated by the simulator. Its predictive accuracy is therefore highest for network conditions that fall within the distribution of its training data. Its ability to generalize to novel or extreme network conditions not represented in the training set is not guaranteed.
- Feature Availability: The model's predictions are predicated on having access to accurate, ground-truth network parameters (bottleneck bandwidth, RTT, loss rate). In a practical deployment, these parameters are not known a priori and must be actively or passively estimated, introducing a potential source of error that could impact the accuracy of the model's recommendations.

6.5 Future Research Directions

This project serves as a strong foundation for numerous promising avenues of future research, expanding upon the framework's capabilities and addressing its limitations.

- Real-World Validation: The most critical next step is to bridge the sim-to-real gap. This would involve deploying the prediction engine on a live server, using its recommendations to dynamically switch the operating system's active CCA (e.g., via sysctl in Linux), and measuring the real-world performance impact. This would provide the ultimate validation of the model's practical utility.
- Adversarial Robustness: Investigate the model's resilience to adversarial conditions. Could a malicious actor generate specific traffic patterns designed to fool the model into recommending a suboptimal CCA, thereby degrading network performance? This involves exploring the intersection of network security and machine learning.
- Multi-Class Classification to Multi-Output Regression: A significant enhancement would be to reframe the problem from classification (predicting the single best CCA) to multi-output regression. A new model could be trained to predict the expected performance metrics (throughput, RTT, etc.) for all five CCAs simultaneously. This would provide the user with a richer set of information, allowing for more nuanced decisions based on application-specific utility functions (e.g., "Which algorithm gives me at least 80% of maximum throughput while keeping RTT below 50ms?").
- Hybrid Deep Learning Architectures: To capture the temporal dynamics of network traffic, future work could explore sequence-based models. A hybrid architecture

combining a Convolutional Neural Network (CNN) to extract features from packet-level data and a Long Short-Term Memory (LSTM) network to model time-series dependencies could lead to more adaptive and context-aware predictions.

- Explainable AI (XAI) Integration: To increase trust and interpretability, XAI techniques like SHAP (SHapley Additive exPlanations) or LIME (Local Interpretable Model-agnostic Explanations) should be integrated. This would allow the framework to explain why it recommended a particular CCA for a specific set of inputs (e.g., "BBR was recommended because the packet loss rate is high and the buffer is deep"), transforming the model from a "black box" into a transparent diagnostic tool.
- Concept Drift Adaptation: The internet is non-stationary; traffic patterns and network topologies evolve over time. To maintain its accuracy, the model must be able to adapt to this "concept drift." Future research could implement online learning or periodic retraining pipelines to ensure the model remains up-to-date with long-term changes in the network environment.
- Federated Learning for Privacy: For a large-scale, real-world deployment across multiple organizations, centralizing sensitive network performance data would be a major privacy concern. Federated Learning offers a solution by allowing a global model to be trained on decentralized data from multiple clients without the raw data ever leaving the client's premises, enabling collaborative model improvement in a privacy-preserving manner.

6.6 Concluding Remarks

The management of network congestion remains a dynamic and vital area of research. This project has demonstrated the profound potential of integrating modern machine learning techniques with traditional network engineering principles. By creating a framework that can both simulate and predict the complex behavior of TCP congestion control algorithms, this work provides a powerful tool for education, research, and practical network optimization. As networks become increasingly complex and autonomous, such intelligent, data-driven approaches will be indispensable in building the smarter, more adaptive, and higher-performing internet of the future.

7. References

1. Jacobson, V. (1988). Congestion avoidance and control. *ACM SIGCOMM Computer Communication Review, 18*(4), 314-329.

2. Fall, K., & Floyd, S. (1996). Simulation-based comparisons of Tahoe, Reno, and SACK TCP. *ACM SIGCOMM Computer Communication Review, 26*(3), 5-21.

3. Allman, M., Paxson, V., & Blanton, E. (2009). TCP Congestion Control. *RFC 5681*.

4. Floyd, S., & Henderson, T. (2004). The NewReno Modification to TCP's Fast Recovery Algorithm. *RFC 3782*.

5. Ha, S., Rhee, I., & Xu, L. (2008). CUBIC: a new TCP-friendly high-speed TCP variant. *ACM SIGOPS Operating Systems Review, 42*(5), 64-74.

6. Brakmo, L. S., & Peterson, L. L. (1995). TCP Vegas: End-to-end congestion avoidance on a global Internet. *IEEE Journal on selected areas in communications, 13*(8), 1465-1480.

7. Shalunov, S., Hazel, G., Iyengar, J., & Kuehlewind, M. (2012). Low extra delay background transport (LEDBAT). *RFC 6817*.

8. Cardwell, N., Cheng, Y., Gunn, C. S., Yeganeh, S. H., & Jacobson, V. (2016). BBR: Congestion-based congestion control. *ACM Queue, 14*(5), 20-53.

9. Jain, R., Chiu, D. M., & Hawe, W. R. (1984). A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. *DEC Research Report TR-301*.

10. Al-Saadi, R., Armitage, G., But, J., & Branch, P. (2019). A survey of delay-based and hybrid TCP congestion control algorithms. *IEEE Communications Surveys & Tutorials, 21*(4), 3609-3630.

11. Jay, J., et al. (2018). A deep reinforcement learning perspective on internet congestion control. *arXiv preprint arXiv:1806.01860*.

12. Xiao, K., et al. (2019). TCP-Drinc: Smart Congestion Control Based on Deep Reinforcement Learning. *IEEE Access, 7*, 11891-11903.

13. Casetti, C., Gerla, M., Mascolo, S., Sanadidi, M. Y., & Wang, R. (2002). TCP Westwood: End-to-end bandwidth estimation for enhanced transport over wireless links. *Proceedings of the 8th annual international conference on Mobile computing and networking*, 287-298.

14. Afanasyev, A., et al. (2010). A survey of TCP congestion control algorithms. *UCLA CSD Technical Report*.

15. Zhang, J., Yao, Z., Tu, Y., & Zhang, Y. (2020). A Survey of TCP Congestion Control Algorithm. *2020 IEEE 5th International Conference on Signal and Image Processing (ICSIP)*, 828-832.

16. Mathis, M., Mahdavi, J., Floyd, S., & Romanow, A. (1996). TCP Selective Acknowledgment Options. *RFC 2018*.