Network Security

Topic: Final Report

Instructor: **Prof. Kanwal Ahmed**

Name:	Mohammad Hamim
ID:	202280090114
Department:	School of International Education
College:	School of Computer and Artificial Intelligence
Major:	Software Engineering
Batch:	2022

Real-Time Malicious URL Detection Using Static Lexical and Structural Features

Mohammad Hamim – 202280090114 [Course: Network Security]
Software Engineering, School of Computer Science and Artificial Intelligence, Zhengzhou University
mohammadhamim@stu.zzu.edu.cn

Abstract

Malicious URLs are a primary vehicle for phishing campaigns, malware distribution, and website defacement, and they frequently evade traditional blacklist-based defenses due to rapid domain churn, URL shortening, and deliberate string obfuscation. This paper presents a practical, real-time malicious URL detection framework that performs **multi-class classification** of URLs into **benign, phishing, malware, and defacement** using only **static lexical and structural characteristics** extracted directly from the URL string. A curated set of **27 features** is engineered to capture length properties, special-character usage, token statistics, abnormal host patterns, IP literal usage, directory depth, URL shortening indicators, and suspicious keyword presence. A **Random Forest** classifier is trained on a large corpus of **651,191 URLs** aggregated from public sources (ISCX URL 2016, PhishTank, PhishStorm, Malware Domain List). To reduce real-world false positives, the system integrates an "intelligent whitelist override" covering 150+ trusted domains across 12 categories, forcing benign classification when a trusted domain match is detected. The deployed prototype is implemented in Python with Streamlit and provides interactive prediction outputs, confidence scoring, and visual analytics. Experimental evaluation reports approximately **98.5% classification accuracy, with an average end-to-end runtime of ~18 ms per URL** and throughput of **~60 URLs/sec**, satisfying the project's objective of sub-50 ms latency for real-time screening. The GitHub Repository is present in: https://github.com/md-hameem/Malicious-URL-Detection-Using-Machine-Learning . The project is Deployed and ready to be tested on: https://malicious-url-detection-hamim.streamlit.app/ .

Keywords: Malicious URL Detection, Phishing Detection, Lexical Feature Engineering, Random Forest Classifier, Cybersecurity, Web Security, Machine Learning, Real-Time URL Analysis

I. INTRODUCTION

Malicious Uniform Resource Locators (URLs) remain one of the most scalable and cost-effective mechanisms for delivering cyberattacks, because they can be distributed across multiple channels—email, SMS ("smishing"), social platforms, advertisements, QR codes, and compromised websites—while requiring minimal attacker infrastructure. At Internet scale, defenders must evaluate vast numbers of links continuously: Google's Safe Browsing program reports scanning billions of URLs per day and **discovering** thousands of new malicious sites daily, illustrating the sustained prevalence of malicious link activity. [1] Similarly, the Verizon Data Breach Investigations **Report (DBIR) continues** to rank phishing and web-based delivery as leading initial access vectors in real-world breaches, underscoring the practical need for automated URL-based screening. [2]

The threat landscape is also characterized by high volume and rapid evolution. The Anti-Phishing Working Group (APWG) observed 1,003,924 phishing attacks in Q1 2025, the highest quarterly total since late 2023, emphasizing that link-based identity theft remains persistent at global scale. [3] APWG further documents the rise of QR-code-driven phishing, where malicious QR codes route victims to phishing sites or malware, often evading traditional URL filtering because the URL is encountered only at scan-time on mobile devices. These developments show that real-time URL screening is not only relevant to email security but also to emerging channels where users are rapidly redirected to attacker infrastructure.

A. Limitations of Traditional Defenses

A widely used baseline defense is blacklist matching—blocking URLs or domains known to be malicious. While effective against previously observed threats, blacklist strategies are **inherently reactive** and struggle with "first-seen" attacks. Modern adversaries exploit this window via rapid domain churn, disposable hosting, and automated domain generation, creating URLs that are operational for only hours or days—often shorter than the time needed for reporting, verification, and distribution of updated blacklist entries. Surveys in malicious URL detection highlight this fundamental limitation: blacklists provide high precision on known threats but weak recall on novel malicious URLs. [4]

Another practical limitation is that many security controls that go beyond blacklists—such as content-based scanning (fetching a page, running scripts, rendering, or sandboxing)—can add latency and operational risk. Visiting a malicious site can trigger drive-by downloads, browser exploitation, or unwanted interactions with attacker infrastructure. As a result, many real-time scenarios (email gateways, chat moderation, QR scanning apps, corporate proxies) prefer **static, non-interactive analysis** that avoids dereferencing URLs but still provides meaningful suspicion scoring.

B. Machine Learning for URL Screening

Machine learning approaches aim to generalize beyond known bad indicators by learning patterns that correlate with malicious intent. Early work such as Ma et al. ("Beyond Blacklists") demonstrated that statistical models can detect malicious sites using properties of suspicious URLs, improving coverage compared to pure blacklist strategies. [5] Subsequent research and surveys have organized the field into three broad feature families: (i) **lexical** (string-based), (ii) **host-based** (DNS/WHOIS/IP reputation), and (iii) **content-based** (HTML/JS/page behavior). Lexical/structural methods are particularly appealing for real-time pipelines because they can operate without external lookups and without fetching content. [4]

Lexical URL analysis leverages the empirical observation that malicious URLs often exhibit distinguishable structural signals: unusually long hostnames, excessive delimiters, digit-heavy tokens, obfuscated paths, deceptive subdomains, embedded brand keywords, or inconsistent protocol/host structures. For example, attackers may embed a trusted brand name as a subdomain within an attacker-controlled domain (e.g., `login.microsoft.com.<attacker-domain>`), or rely on multiple redirects/shorteners to hide the true destination. These patterns can be captured through engineered features and learned by supervised models such as decision trees, Random Forests, gradient boosting, or deep sequence networks.

C. Scope and Objective of This Work

This work formulates malicious URL detection as a **multi-class classification** task with four outcome categories: **Benign, Phishing, Malware, and Defacement**. The system is designed for both predictive strength and operational practicality, with explicit goals of achieving **>95% accuracy** and **sub-50 ms** per-URL latency for real-time usage. Unlike approaches that require crawling or reputation lookups, this implementation intentionally restricts itself to **static lexical and structural feature extraction**, ensuring that classification can be performed safely and quickly on any URL string.

To support robust learning, the model is trained on a large dataset (651,191 URLs) aggregated from widely used public sources (including ISCX URL 2016, PhishTank, PhishStorm, and Malware Domain List). This dataset includes a realistic class imbalance where benign URLs dominate (~65.7%), reflecting real-world traffic distributions, while still providing substantial representation across multiple malicious categories.

D. Proposed Approach Overview

The proposed pipeline consists of three major components:

1. **Lexical–Structural Feature Engineering**: Each URL is mapped to a compact vector of 27 numeric features, including length-based metrics, character-count statistics for special symbols, and binary/structural indicators such as the presence of IP literals, shortening services, abnormal host patterns, directory depth, embedded domain patterns, and suspicious token cues.
2. **Ensemble Learning via Random Forest:** A Random Forest classifier is trained on the engineered feature vectors to predict one of the four classes. Random Forests are well-suited for heterogeneous tabular features, provide strong generalization in practice, and offer interpretability through feature importance—useful for understanding which lexical signals dominate the decision process. [12]
3. **False-Positive Mitigation with a Whitelist Override:** Because operational trust and user experience degrade rapidly with false alarms, the system includes an "intelligent whitelist" that checks URLs against **150+ trusted domains** across multiple categories (search, social, banking, government, etc.). If `hostname.endswith(trusted_domain)`, the URL is forced to **Benign** with 100% confidence prior to ML prediction. This is intended to reduce disruptive false positives for widely used legitimate services.

E. Contributions

The main contributions of this paper are:

- **A practical, static URL detection pipeline** that avoids DNS resolution and web crawling, enabling safe deployment in latency-sensitive settings.
- **A 27-feature lexical–structural representation** explicitly designed to capture common obfuscation and deception strategies used in malicious URLs.
- **A multi-class Random Forest model** trained on a large-scale corpus (651,191 URLs) covering benign, phishing, malware, and defacement categories.
- **An end-to-end deployable implementation** with automated testing and runtime benchmarks; reported performance includes approximately **98.5% accuracy, around 18 ms** average latency per URL, and ~60 URLs/sec throughput under test conditions.
- **A usability-oriented whitelist safeguard** designed to minimize false positives for major trusted platforms and reduce alert fatigue in day-to-day use.

F. Paper Organization

The remainder of this paper is organized as follows: Section II reviews related work in malicious URL detection and motivates lexical-only modeling. Section III describes the dataset and preprocessing. Section IV details the 27-feature engineering pipeline. Section V explains the Random Forest model and training procedure. Section VI presents experimental results and performance evaluation. Section VII discusses limitations, security considerations (including whitelist safety), and future work. Finally, Section XI concludes.

II. RELATED WORK

Research on malicious URL detection has evolved from reactive blacklist matching to proactive learning-based systems that attempt to generalize to previously unseen threats. The literature is commonly organized by **feature sources** (lexical, host-based, content-based) and **learning paradigms** (batch ML, online/streaming, deep learning), with important practical considerations such as latency, safety, and concept drift. [4]

A. Beyond Blacklists and Early Learning-Based URL Screening

One of the most influential early works, Ma et al. (KDD 2009), demonstrated that statistical learning can detect malicious websites using properties derived from URLs and related signals, improving coverage beyond what blacklists can provide. ([Computer Science Department at UCSD][2]) Their key insight is that attackers often produce URLs with structural and lexical artifacts that differ from benign URLs, and these artifacts can be exploited by supervised learning to **identify suspicious links before** they appear on public blocklists.

B. Lexical and Structural Feature Engineering for Malicious URLs

A large portion of subsequent work focuses on **lexical analysis**—features extracted purely from the URL string such as length statistics, delimiter counts, digit ratios, token patterns, path depth, and suspicious substrings—because it avoids dereferencing potentially harmful content and supports low-latency deployment. The survey by Sahoo et al. (2017) highlights lexical features as among the most scalable representations, especially for real-time applications that cannot afford crawling or extensive host lookups. [4]

A representative example of lexical modeling is Mamun et al. (NSS 2016), which presented malicious URL detection using lexical analysis and is strongly associated with the ISCX-URL2016 / URL2016 dataset ecosystem widely used in academic benchmarking. [6] Their work reinforced that string-level artifacts (e.g., abnormal length distributions, special character patterns, and deceptive token usage) can provide sufficient signal for separating benign from multiple malicious categories.

From an applied perspective, Joshi et al. (2019) reported a lexical-feature-based ensemble approach for malicious URL detection in email pipelines, emphasizing that static features can provide fast verdicts without crawling and that bagging/ensemble methods can be effective on noisy, unstructured URL strings. [4] This is especially relevant for production environments where decision latency and operational safety are primary constraints.

Overall, lexical approaches occupy a practical "sweet spot": they are generally **safer** than content-based scanning, **faster** than host-reputation pipelines, and can still achieve strong predictive performance if the feature set captures attacker obfuscation strategies.

C. Host-Based and Reputation-Driven Detection

Another major branch relies on **host-based** signals such as WHOIS registration attributes, DNS behavior, IP reputation, and domain age. These features can be highly informative—newly registered domains, unusual DNS patterns, or fast-changing infrastructure often correlate with malicious activity—but they introduce costs and constraints: DNS/WHOIS lookups add latency, can fail due to rate limits or privacy restrictions, and may not be feasible for offline or strictly isolated environments.

Work in adjacent areas such as **fast-flux** detection often leverages DNS response patterns and temporal/spatial resolution features to identify malicious domains that rotate IP addresses rapidly. [10] While fast-flux detection targets domain infrastructure rather than URL strings directly, it illustrates the broader theme that host-level telemetry can materially improve detection when available—at the expense of extra dependencies and runtime overhead.

Similarly, research on **DGA (Domain Generation Algorithm)** detection explores feature engineering and ML/DL methods to identify algorithmically generated domains used for command-and-control. [11] Although DGA detection is not identical to full URL classification (it focuses on the domain token), it informs malicious URL detection by highlighting the value of character-level distributions, n-gram statistics, and sequential patterns—particularly for attacker-generated strings.

D. Content- and Behavior-Based URL Classification

Content-based approaches fetch the landing page and analyze HTML structure, JavaScript behavior, redirection chains, embedded resources, and sometimes dynamic execution traces in a sandbox. These methods can detect threats that are "lexically clean" (e.g., compromised benign domains serving malicious content) because they examine the payload and behavior rather than relying on string patterns alone. However, content-based analysis is associated with three practical drawbacks:

4. **Safety risk**: dereferencing unknown URLs can expose scanners to drive-by attacks and exploitation.
5. **Latency and cost**: rendering, JS execution, and sandboxing are computationally expensive and slow.
6. **Operational complexity**: crawlers must handle bot detection, geo/IP-based cloaking, and dynamic content.

The broad trade-off is well summarized in survey literature: content-based signals may improve recall in some scenarios but complicate deployment and can be unsuitable for strict real-time decision points. [4]

E. Deep Learning and Learned URL Representations

Deep learning approaches aim to reduce reliance on hand-crafted lexical features by learning representations directly from raw URL strings. A prominent example is URLNet (Le et al., 2018), which uses convolutional neural networks over both character- and word-level components of URLs to capture sequential patterns and token semantics, reporting improvements over classical bag-of-words and manually engineered baselines. [8] This line of work addresses three recurrent limitations of traditional lexical methods: reliance on expert-designed features, difficulty capturing sequential character patterns, and reduced robustness to adversarial string perturbations.

More recent literature also explores hybrid CNN/LSTM or transformer-style models for phishing/URL classification, typically emphasizing character-level modeling to capture obfuscation patterns. [21] Despite strong accuracy reports in some studies, deep learning methods can introduce heavier inference costs and may require careful dataset curation to avoid leakage and to ensure robustness against distribution shift—factors that matter in real-time deployments and in educational projects where compute resources are constrained.

F. Streaming, Online Learning, and Concept Drift

A practical reality for malicious URL detection is concept drift: attacker tactics, shortening services, URL templates, and infrastructure choices evolve continuously, causing static models to degrade over time. Survey work explicitly calls out drift and adversarial adaptation as open challenges, motivating online learning, periodic retraining, and monitoring strategies. [4]

Systems designed for operational pipelines often emphasize fast, incremental decisioning. For example, **PhishStorm** is positioned as a phishing detection approach built with streaming analytics considerations, reflecting the broader need to handle continuous flows of URLs rather than fixed offline batches. [9] While methodologies vary (lexical heuristics, search-relatedness features, or streaming classifiers), the shared objective is to maintain detection performance under evolving threat conditions.

G. Usability, False Positives, and Hybrid Safeguards

Across many academic studies, the primary focus is maximizing detection metrics (accuracy, AUC, recall) under benchmark datasets; however, real-world usability is often limited by **false positives** that disrupt benign workflows and reduce user trust. Production-aligned work frequently combines ML with deterministic safeguards (e.g., allowlists, reputation caches, policy rules) to reduce operational friction. The practical emphasis on low latency and safe static analysis in lexical-only systems aligns with such hybrid deployment needs.

At the same time, hybrid designs introduce their own risks: naïve allowlisting logic can be abused if domain boundary checks are not robust (e.g., suffix tricks and deceptive subdomain structures). Consequently, modern secure implementations typically recommend registrable-domain validation (Public Suffix List awareness) and careful normalization before applying allow/deny rules—an important engineering consideration when combining ML with whitelist overrides.

H. Positioning of the Present Work

Motivated by the above trade-offs, the present work follows the widely validated **lexical–structural** line of research for safe, low-latency URL screening, while adopting an **ensemble classifier (Random Forest)** consistent with strong tabular performance in prior applied studies. [4] Unlike content-based pipelines, it avoids dereferencing URLs; unlike host-based pipelines, it does not require DNS/WHOIS access at inference time. The system also incorporates an explicit usability safeguard (trusted-domain whitelist override) and a full deployment prototype, positioning it as both an ML study and an end-to-end security tool implementation.

III. DATASET AND PROBLEM DEFINITION

A. Problem Definition

This work addresses **malicious URL detection** as a supervised **multi-class classification** problem. Given an input URL string (u), the task is to predict a discrete label (y) from four security-relevant categories: y ∈ {Benign, Phishing, Malware, Defacement. The goal is to learn a classifier (f(·)) such that: hat{y = f(u) where (hat{y) is the predicted class. In addition to the class label, the deployed system reports a **confidence score** derived from the model's estimated class probabilities.

A central design constraint is **safe, real-time decisioning**: the system must classify URLs using only intrinsic characteristics of the URL string (lexical/structural properties) and must not require visiting the destination site, in order to avoid exposure to potentially harmful content and to meet strict latency targets. This requirement is consistent with the project objectives of **< 50 ms** per-URL prediction latency and string-only feature extraction.

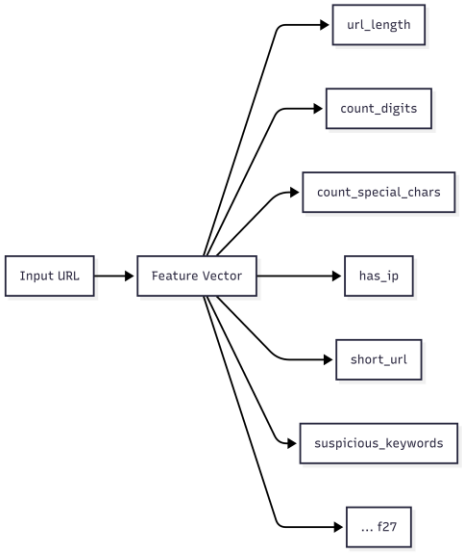

Figure 01: Feature Vector Representation

B. Threat Model and Scope

We assume an adversary can generate or manipulate URLs to:

I. impersonate trusted brands (phishing),
II. host or distribute executable payloads (malware), or
III. point to compromised sites that have been altered to display unauthorized content (defacement).

The detection model operates **purely on the URL text**, which means it targets attacker behaviors that manifest in the string itself (e.g., abnormal structure, suspicious tokens, obfuscating delimiters, unusual length distributions). Host-level reputation signals (WHOIS age, DNS patterns, IP reputation) and content-level signals (HTML/JS behavior) are considered out of scope at inference time to preserve safety and speed.

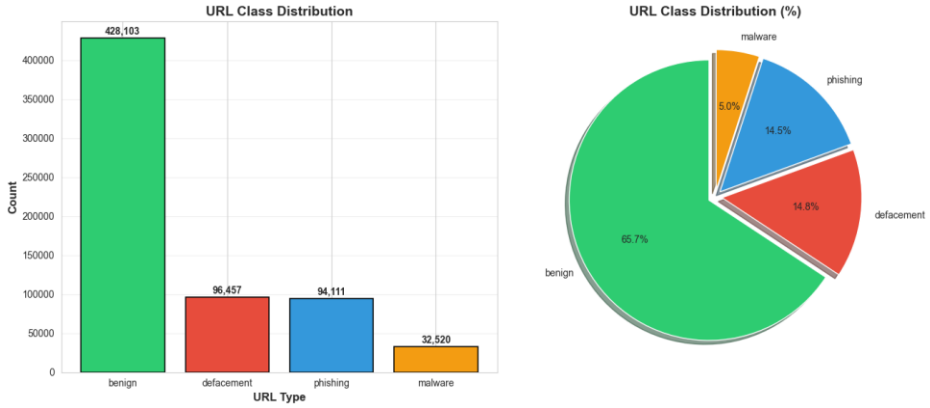

Figure 02: URL Class Distribution

C. Dataset Provenance and Collection

The experiments are built on a large, publicly available dataset distributed on Kaggle as the "**Malicious URLs Dataset**" curated by Siddhant Baldota, which aggregates URLs from multiple security data sources. [20] The reported upstream sources include:

- **ISCX URL 2016 (URL-2016):** A widely cited academic dataset and associated study focusing on detecting malicious URLs via lexical analysis. [17]

PhishTank: A community-driven clearinghouse providing phishing URL intelligence and an API for research and integration. [18] **PhishStorm:** A research line emphasizing real-time phishing detection from URL characteristics (often discussed in streaming/real-time detection contexts). [9] **Malware Domain List**: A threat-intelligence source listing domains/URLs associated with malware activity, used to enrich malicious samples. [19]

This multi-source aggregation is important because it increases diversity across malicious behaviors and reduces reliance on a single collection methodology, improving the likelihood that the learned classifier generalizes beyond one narrow distribution.

D. Dataset Schema and Label Semantics

The Kaggle dataset contains **651,191 URL instances**. Each instance consists of a raw URL string and a categorical label indicating the URL type. The four label meanings used in this work are:

Benign: safe, legitimate URLs.

- **Phishing:** URLs imitating legitimate entities to steal credentials or sensitive data.
- **Malware:** URLs that host, deliver, or facilitate distribution of malicious software.
- **Defacement:** URLs associated with compromised pages altered to show unauthorized content.

These four categories match the project objective of multi-class classification across benign, phishing, malware, and defacement.

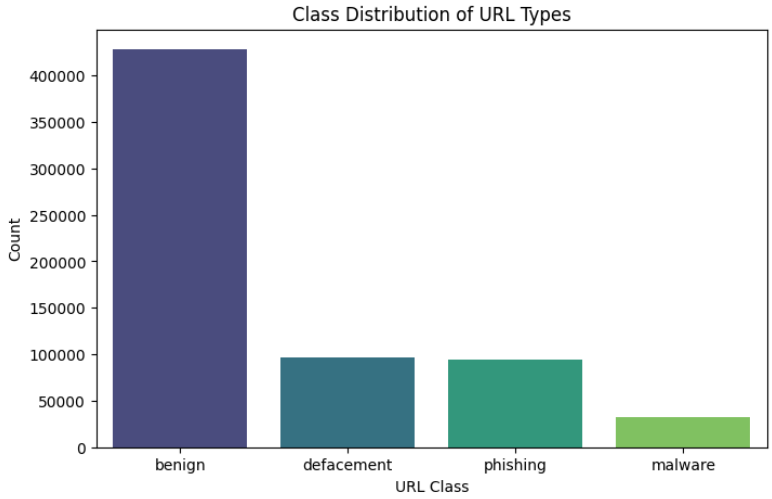

Figure 03: Class Distribution URL Types

E. Class Distribution and Imbalance Analysis

The dataset is **class-imbalanced**, reflecting realistic web traffic where benign URLs dominate. The distribution is:

Benign: 428,103 (65.7%)

- **Defacement:** 96,457 (14.8%)
- **Phishing:** 94,111 (14.5%)
- **Malware:** 32,520 (5.0%)

This imbalance has two implications:

- Accuracy alone can be misleading, because a classifier biased toward the majority class can still score high overall accuracy. Therefore, IEEE-quality reporting should include class-wise precision/recall/F1 and macro-averaged measures in addition to accuracy.
- The minority class (malware, 5.0%) requires careful attention, since operational security systems often value high recall on rare but high-impact categories.

F. Data Preparation and Experimental Protocol

To build a deployable multi-class classifier, the training pipeline follows a standard supervised learning protocol:

- **Label Encoding:** The categorical target labels are encoded into numeric classes for model training and then mapped back to human-readable labels for inference.
- **Train/Test Split:** The dataset is partitioned using an **80/20 split** into training and held-out test subsets.
- **Static Feature Extraction:** Each URL is transformed into a fixed-length vector via lexical and structural feature engineering; importantly, these features are "purely lexical and safe to extract without network interaction."

These steps do not change the "string-only" safety property of the system, but they improve evaluation rigor and reproducibility in an IEEE-style manuscript.

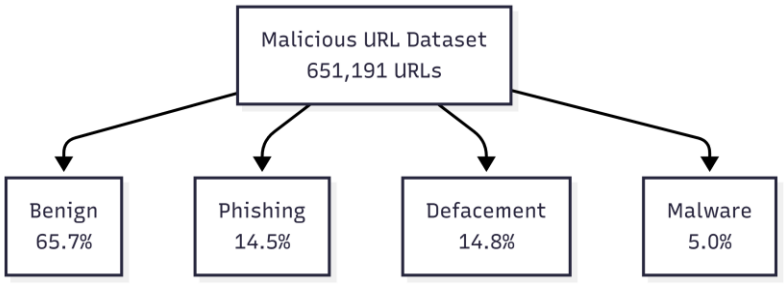

Figure 04: Dataset distribution

IV. FEATURE ENGINEERING

This system extracts a **fixed set of 27 lexical and structural features** from each URL string. The feature design is explicitly **static** (no DNS queries, no page fetching) and therefore "safe to extract without network interaction," while still capturing common attacker obfuscation patterns.

A. Design Principles

Feature engineering was guided by four deployment-oriented principles:

1. **Safety:** Avoid dereferencing unknown links to eliminate exposure to drive-by content and reduce operational risk.
2. **Low latency:** Use features computable in (O(|u|)) time (linear in URL length), dominated by simple counts and a small number of regex checks.
3. **Generalization:** Prefer structural indicators (lengths, delimiter usage, IP-in-host, shorteners) that tend to remain informative even as specific domains change—an approach consistent with lexical URL detection literature.
4. **Deterministic ordering:** Maintain an **exact, consistent feature order** between training and inference ("EXACT ORDER FROM MODEL") to prevent mismatched columns during prediction. (This is essential for scikit-learn style tabular models.)

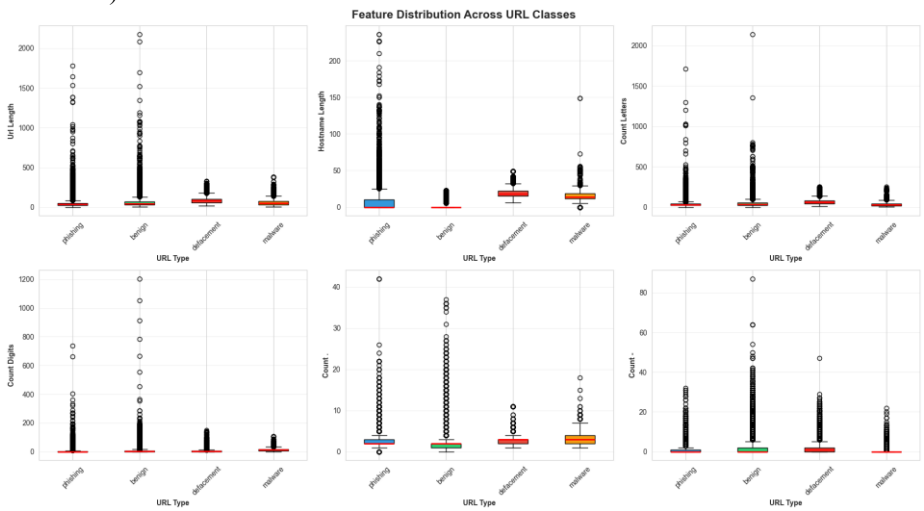

Figure 05: Feature extraction process

B. URL Parsing and Minimal Normalization

A URL (u) is decomposed into its components using a standard parser (scheme, hostname/netloc, path, query). Hostname and path are required for several structural features such as `hostname_length`, directory depth, and first-directory length.

TLD extraction Correctly identifying the top-level domain (TLD/public suffix) is non-trivial because naive dot-splitting fails on multi-part suffixes (e.g., `.co.uk`). Libraries based on the Public Suffix List handle these cases robustly. In this implementation, TLD length is computed via a dedicated TLD utility (with safe fallback to 0 if parsing fails), ensuring the pipeline remains robust under malformed URLs.

Handling malformed inputs When parsing fails (e.g., empty host, missing path segments, uncommon formats), feature functions return safe defaults (typically 0), preventing runtime errors and ensuring every URL yields a complete 27-dimensional vector.

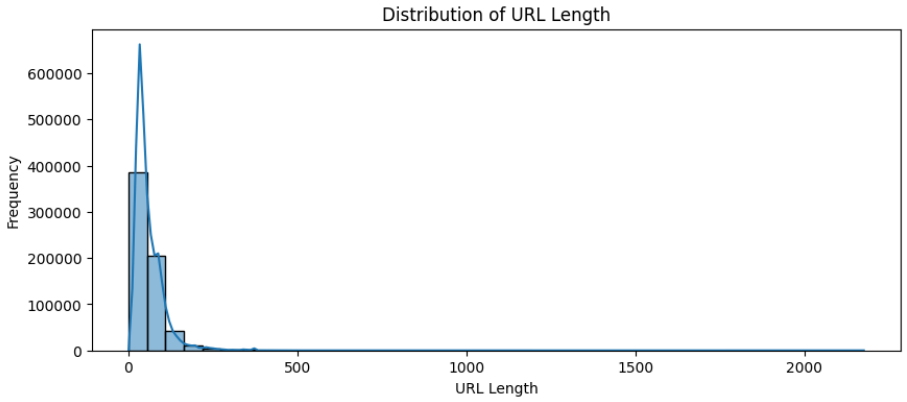

Figure 06: Distribution of URL Length

C. Feature Set Overview (27 Features)

The extracted features follow the project specification: **4 length-based, 15 count-based, and 8 structural/binary** indicators.

Table I — Engineered Lexical/Structural Feature Set (27 total)

Group	Feature	Definition / Computation
Length-based	url_length	Total number of characters in the full URL string |u|
Length-based	hostname_length	Length of hostname (domain/netloc)
Length-based	fd_length	Length of the first directory token in the URL path (0 if none)
Length-based	tld_length	Length of extracted TLD/public suffix (0 if unavailable)
Count-based	count_letters	Number of alphabetic characters in u
Count-based	count_digits	Number of numeric digits in u
Count-based	count_@	Count of '@'
Count-based	count_?	Count of '?'
Count-based	count_-	Count of '-'
Count-based	count_=	Count of '='
Count-based	count_.	Count of '.'
Count-based	count_#	Count of '#'
Count-based	count_%	Count of '%'
Count-based	count_+	Count of '+'
Count-based	count_$	Count of '$'
Count-based	count_!	Count of '!'
Count-based	count_*	Count of '*'
Count-based	count_,	Count of ','
Count-based	count_slashes	Count of '//' occurrences in u

Structural/Binary	count_dir	Directory depth (number of '/' in path)
Structural/Binary	count_embed_domain	Count of '//' tokens inside the path (proxy for embedded URL patterns)
Structural/Binary	count_www	Count/presence of 'www' in URL
Structural/Binary	has_ip	1 if hostname matches IPv4 literal regex, else 0
Structural/Binary	abnormal_url	1 if hostname is missing or does not appear in the full URL as expected, else 0
Structural/Binary	short_url	1 if URL matches known shortening services, else 0
Structural/Binary	https	1 if scheme indicates HTTPS, else 0
Structural/Binary	suspicious	1 if URL contains suspicious tokens (e.g., login, bank, account), else 0

Note: Note: the project report groups “counts of special characters” together (including / /) and lists structural/binary indicators such as IP-in-host, abnormality, shorteners, HTTPS, and suspicious keywords.

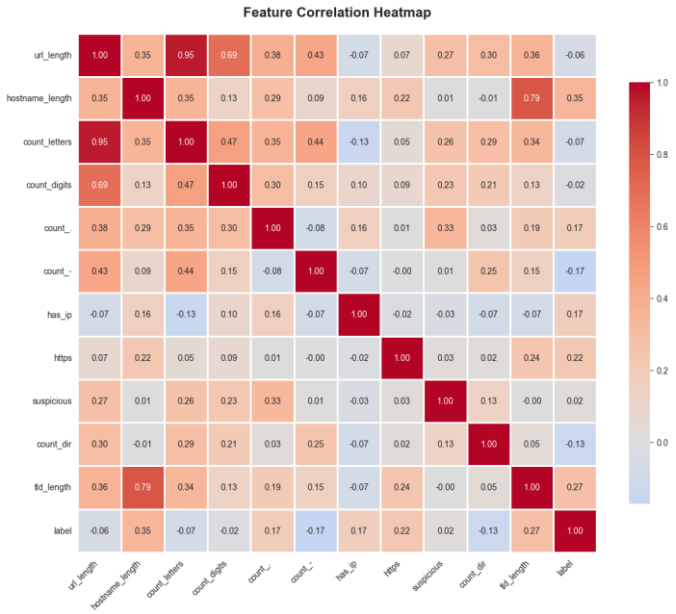

Figure 07: Feature vector representation

D. Mathematical Formulation

Let (u) be the URL string, (h) the extracted hostname, and (p) the path string.

Length features url_length=|u|, hostname_length=|h|, fd_length=|p_1|, tld_length=|TLD(h)|. where (p_1) is the first directory token after splitting (p) on “/”.

Count features For a character/token (c): count_c=#{i : u[i]=c. For the substring token “//”: count_slashes=#{occurrences of “//” in u.

Structural/binary indicators has_ip= begin{cases 1 & if hostname matches IPv4 regex 0 & otherwise end{cases short_url= begin{cases 1 & if u matches known shortener patterns 0 & otherwise end{cases suspicious= begin{cases 1 & if u contains suspicious keywords (login/bank/account/…) 0 & otherwise end{cases

E. Why These Features Work (Security Rationale)

Lexical URL detection is effective because malicious URLs frequently reveal intent through **structure and composition**, even when the final payload is hidden behind redirects. Seminal and widely cited work shows that suspicious URLs exhibit detectable lexical/host-pattern differences compared to benign URLs, enabling proactive classification beyond blacklists.

Key attacker behaviors captured by the features include:

Obfuscation and deception via length and delimiters: Excessive URL length, heavy use of punctuation (`-`, `_`, `%`, `=`, `?`) and deep paths are common in phishing and exploit kits to evade simple filters and to mimic legitimate query strings.

- **IP-literal hosting (`has_ip`):** Direct IP addressing is frequently used by malware hosting and temporary infrastructures to avoid domain-based reputation.
- **Shortener abuse (`short_url`):** Short links conceal the true destination and are widely abused in phishing campaigns; detecting common shortening services is therefore operationally valuable.
- **Embedded URL/redirect artifacts (`count_slashes`, `count_embed_domain`):** Attackers often embed a full URL inside a path or parameter (e.g., `.../http://evil.com/...`) to create multi-stage redirection; counting "//" patterns helps flag such constructions.
- **Social-engineering tokens (`suspicious`):** Words such as "login," "signin," "bank," "account," "update," etc., frequently appear in credential-harvesting pages; simple keyword cues remain strong baseline signals.

Deep learning approaches such as URLNet attempt to learn these patterns directly from raw URL sequences, but classical engineered features remain attractive in low-latency, resource-constrained deployments due to interpretability and speed.

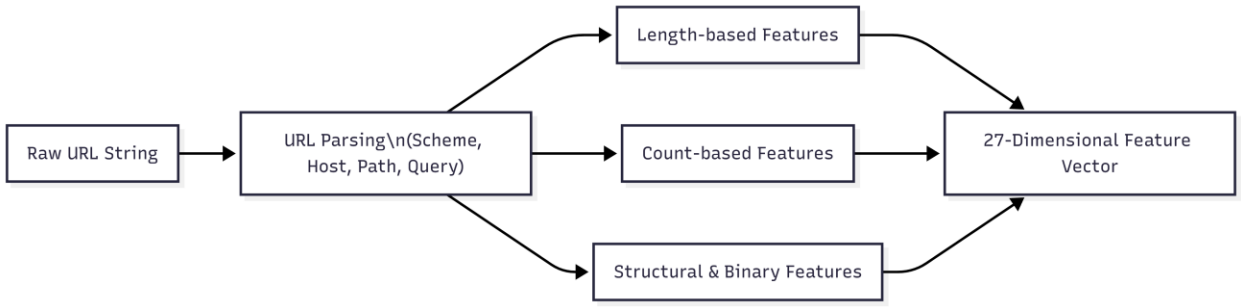

Figure 08: Lexical Feature Extraction Process

F. Extraction Algorithm and Complexity

Algorithm 1 — Feature Extraction (per URL)

a. Parse URL into hostname (h) and path (p).
b. Compute length features: `url_length`, `hostname_length`, `fd_length`, `tld_length`.
c. Compute count features using linear scans / `count()` operations for the specified characters and tokens.
d. Compute structural/binary flags using small regex checks and parser-based logic.
e. Output a 27-dimensional vector in the **same feature order** used during training.

Time complexity: Each count and regex check is bounded by (O(|u|)); with a constant number of features, total runtime is (O(|u|)) per URL, supporting real-time classification. This aligns with the project's stated goals of low per-URL latency and real-time usability.

V. PROPOSED MODEL

A. Overview of the Detection Model

The proposed system performs **multi-class URL classification** into four categories—**Benign, Phishing, Malware, Defacement**—using a supervised ML pipeline trained on **651,191 URLs**. The model operates on **a 27-dimensional lexical–structural** feature vector extracted from the raw URL string (no DNS resolution and no web content fetching).

At inference time, the decision logic is executed in the following order:

I. **Whitelist check** (trusted-domain override)
II. **Feature extraction** (27 features)
III. **Random Forest prediction** (class 0–3)
IV. **Confidence calculation** using the model's predicted probability for the selected class

This ordering is designed to minimize false positives for common trusted services (via allowlist override) while retaining ML-based generalization for unknown or suspicious URLs.

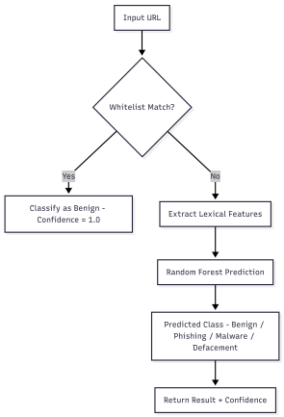

Figure 09: Real-time URL pipeline

B. Random Forest Classifier for Multi-Class URL Labeling

The core classifier is a Random Forest, i.e., an ensemble of decision trees where each tree is trained on a bootstrapped sample of the training set and split decisions are made using randomized feature selection at each node. In this work, Random Forest is chosen because it (i) performs strongly on heterogeneous tabular features, (ii) handles non-linear decision boundaries without heavy preprocessing, (iii) offers interpretability through feature importance, and (iv) supports fast inference suitable for real-time pipelines.

Model choice in the project: Random Forest Classifier.

Configuration note (reported): "Ensemble of decision trees (n_estimators = 100 typically)."

Formally, let (x ∈ mathbb{R^{27) be the engineered feature vector for a URL. A Random Forest contains (T) trees ({h_t_{t=1^{T). Each tree outputs a class label (h_t(x) ∈ {0,1,2,3). The forest prediction is the majority vote: hat{y = modeleft(h_1(x), h_2(x), ..., h_T(x)right). For probability outputs (used for confidence), the forest estimates class posterior probabilities by averaging the per-tree class distributions.

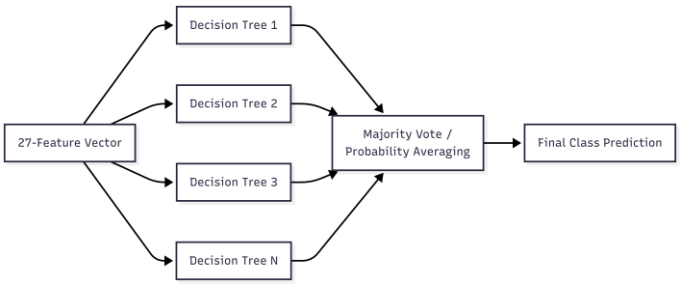

Figure 10: Random Forest architecture

C. Training Strategy and Preprocessing

The training pipeline follows a standard supervised protocol:

- **Train/Test split: 80/20** split for evaluation.
- **Target encoding: Label Encoding** is applied to map string labels (Benign/Phishing/Malware/Defacement) into numeric class IDs required by the classifier.
- **Feature representation**: Each URL is converted into a fixed-length **27-feature** vector prior to learning.

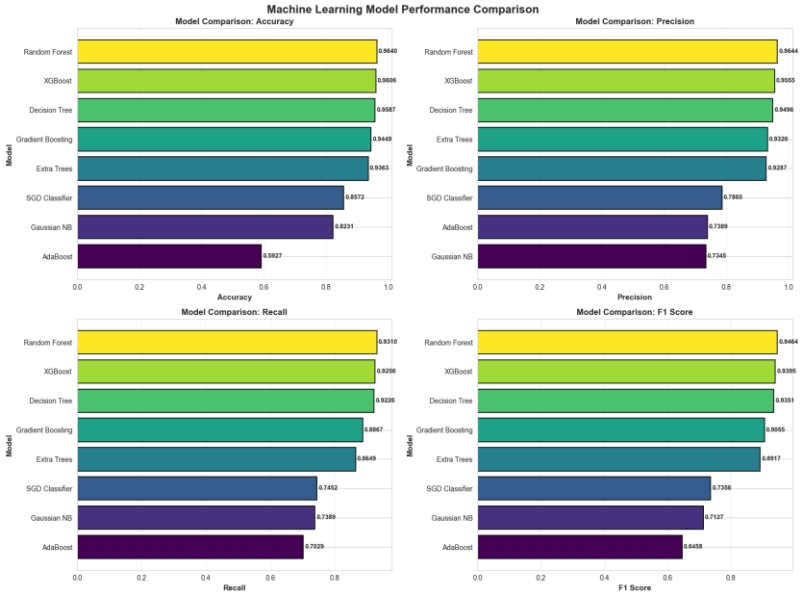

Figure 11: Machine Learning Model Performance Comparison

D. Confidence Score and Decision Output

The deployed application returns:

- predicted class label, and
- **confidence score** computed as the model's probability assigned to the predicted class.

If the whitelist rule triggers, the system **forces Benign with 100% confidence** before ML inference: if hostname.endswith(trusted_domain) Rightarrow label=Benign,;confidence=1.0 This explicit override is part of the deployed decision policy.

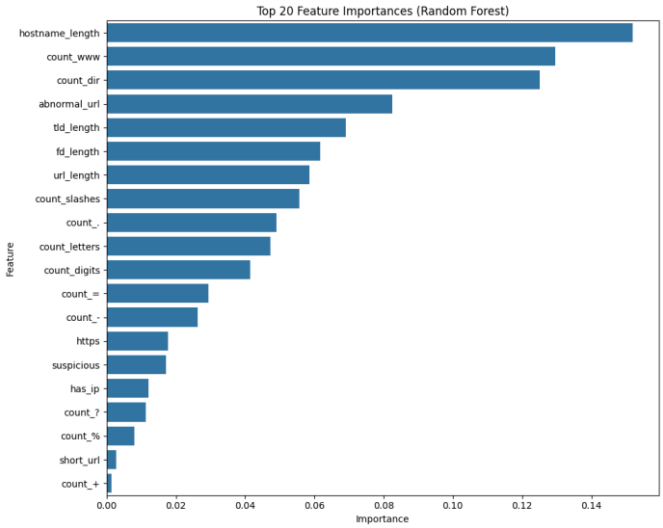

Figure 12: Feature Importance (RF)

E. Model Serialization and Deployment

To support production-style reuse (UI + tests + batch execution), the trained artifacts are persisted as follows:

- **Model file**: `final_random_forest_model.pkl`
- **Label encoder**: `label_encoder.pkl`
- **Serialization method**: Python `pickle`

A dedicated test (`test_load_models`) verifies the model and encoder load correctly at runtime, supporting robustness and preventing silent deployment failures.

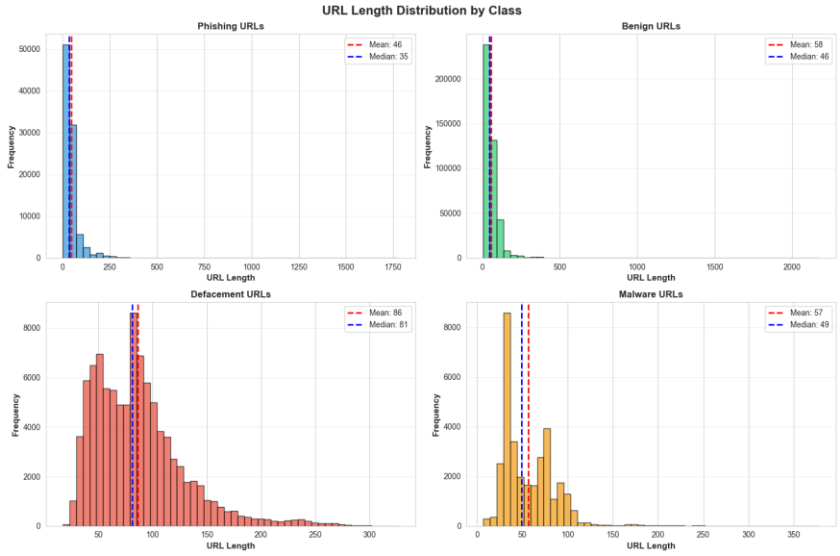

Figure 13: URL Length Statistics by Class

F. Computational Cost and Real-Time Suitability

The model's runtime suitability is supported by the design:

- Feature extraction is based on deterministic counts/regex/parsing over the URL string (constant number of operations per feature set).
- Random Forest inference typically scales with the number of trees and tree depth; in practice, this is efficient for small tabular inputs like 27 features.

The project's benchmarking reports approximately **~18 ms average latency per URL** and **~60 URLs/sec throughput** in the full system (feature extraction + inference + reporting overhead).

VI. Whitelist Override for False-Positive Mitigation

A. Motivation and Design Goal

In operational URL filtering, **false positives** (blocking/flagging legitimate, high-traffic platforms) quickly reduce user trust and lead to alert fatigue. To address this, the system introduces an **Intelligent Whitelist System**: a hard-coded override that checks an input URL against a curated set of 150+ trusted domains across **12 categories** before running machine learning. The project objective explicitly targets "0% false positives for major trusted platforms (e.g., Google, Facebook, Banking sites)."

B. Whitelist Scope and Categorization

The whitelist is organized across **12 functional** categories—Search Engines, Social Media, Tech Giants, Developer Platforms, Media, E-commerce, Education, Cloud Services, News, Email, Banking, and Government—reflecting the most common "everyday" destinations where false alarms would be most disruptive.

This categorization is beneficial for maintainability because it allows:

fast auditing (e.g., verify that “Banking” entries are current),

- targeted updates (e.g., add/remove platforms by category), and
- policy tuning (e.g., stricter treatment for user-generated content platforms).

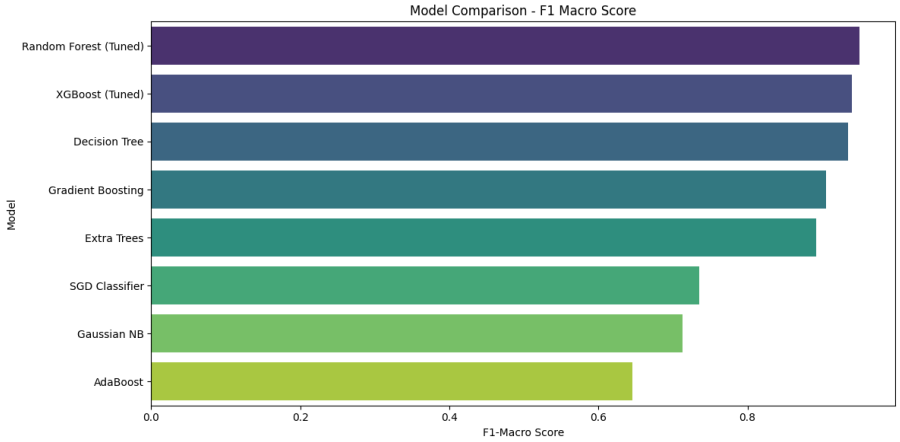

Figure 14: Model Comparison - F1 Macro Score

C. Decision Policy and Pipeline Placement

The whitelist is executed **as the first stage** in the real-time analysis engine:

- whitelist check → 2) feature extraction → 3) Random Forest prediction → 4) confidence calculation.

Whitelist rule (as implemented in the project specification): If `hostname.endswith(trusted_domain)` → force **Benign** with **100% confidence**, bypassing ML inference.

This creates a two-tier decision policy:

- **Tier 1 (Deterministic allow):** known trusted domains → “Benign”
- **Tier 2 (Predictive model):** everything else → ML classification

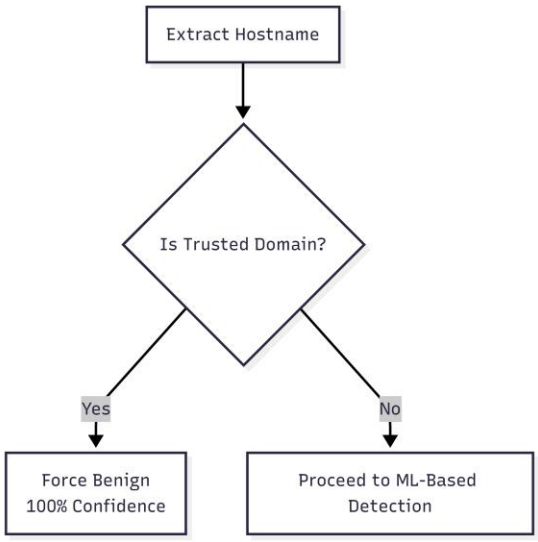

Figure 15: Whitelist logic

D. Implementation Details and Recommended Hardening

1) URL parsing and canonicalization

Correct allowlisting must operate on the **canonical hostname** extracted from the URL's authority component (per URI generic syntax), rather than naive substring checks on the full URL string. [13] Recommended preprocessing steps:

- Parse using a standards-compliant URL parser to extract hostname. [13]
- Normalize hostname:
- lowercase (DNS hostnames are case-insensitive),
- remove trailing dot (e.g., `example.com.`),
- strip ports (e.g., `example.com:443`),
- apply IDN handling (convert Unicode domains to ASCII-compatible encoding / punycode) where applicable (important for internationalized domains and homograph-risk reduction). [13]

These steps ensure the whitelist decision is consistent and not bypassed or mis-triggered by formatting variations.

2) Avoid unsafe `endswith()` allowlist matches

A raw `hostname.endswith("google.com")` check can incorrectly allow domains like `notgoogle.com` (because the string suffix matches). This is a known class of allowlist bug when using `startsWith/endsWith` style checks without enforcing correct boundary characters. [24]

Safer match rule (recommended IEEE-grade implementation): Let `h` be the normalized hostname and `d` a trusted domain. Allow only if:

- `h == d`, or
- `h.endswith("." + d)` (dot-boundary enforced)

This dot-boundary requirement is explicitly highlighted in security guidance: when `endsWith` must be used, you must include the "`.`" boundary to avoid inaccurate matches. [24]

3) Registrable-domain validation using the Public Suffix List

Many modern services operate under "shared registrable spaces" (e.g., `.github.io`, `.blogspot.com`) where subdomains are **user-controlled**. Treating such spaces as universally trusted is risky.

A robust approach is to compute the registrable domain (eTLD+1) using the Public Suffix List (PSL)—a maintained set of suffix rules used broadly in browsers and security tooling. [15] Python libraries such as tldextract perform this separation (subdomain, domain, suffix) using PSL rules. [16]

Recommended refinement:

- Maintain whitelist entries at the **registrable domain** level (e.g., `google.com`, `bank.com`).
- Optionally treat "private suffixes" (e.g., platform-hosted domains) cautiously; PSL-aware tooling can distinguish public suffix and (optionally) private suffix handling. [16]

This reduces the risk of mistakenly allowlisting attacker-controlled subdomains that live under a shared hosting provider.

E. Security Trade-offs and Threat Analysis

While the whitelist reduces false positives, it introduces **intentional** blind spots. Key risks include:

- **Compromised trusted domains:** If a whitelisted domain is compromised or hosts malicious content, the allowlist bypass prevents ML from flagging it.
- **Trusted platforms with user-generated content:** Even if the parent domain is reputable, user content (e.g., shared documents, pages, redirects) may be abused for phishing delivery.
- **Domain-matching pitfalls:** Naive suffix matching can allow attacker-owned lookalike domains unless dot-boundary and registrable-domain logic are enforced. [24]
- **Staleness risk:** Whitelists must be maintained; PSL guidance notes the list changes frequently and stale lists can introduce security and privacy issues. [15]

Operationally, this means the whitelist should be treated as a policy control, not a purely technical guarantee of safety.

F. Testing and Verification

The project includes explicit automated tests covering whitelist behavior:

`test_google_prediction` validates allowlisting on a known trusted domain,

- `test_whitelist_override` performs batch tests across major domains (e.g., YouTube, GitHub),
- additional tests verify model loading, feature extraction correctness, malicious URL detection, and performance metrics.

This is important because allowlist bugs are high-impact: a single incorrect match rule can silently permit malicious domains.

G. Recommended Enhancements for an IEEE-grade Deployment

To make the whitelist mechanism safer while preserving its usability benefits:

- **Use boundary-safe domain matching** (`h == d` or `h.endswith("." + d)`). [24]
- **Adopt PSL-based registrable-domain extraction** and store whitelist entries at eTLD+1. [15]
- **Introduce "conditional allowlisting"** for user-generated platforms:
- allow domain, but still run ML if paths/queries contain **credential keywords** or redirects, or if the URL uses unusual embedding patterns.
- **Log whitelist hits** (domain, timestamp, URL sample) to support auditing and to detect abuse patterns.
- **Periodic review/update** of the whitelist list and PSL cache, since suffix rules and platform domains evolve. [15]

VII. SYSTEM ARCHITECTURE AND IMPLEMENTATION

A. Architectural Overview

The proposed solution is implemented as a **modular, end-to-end malicious URL detection system** consisting of four primary layers: **(i) presentation/UI, (ii) real-time analysis engine, (iii) model + feature services, and (iv) quality assurance/reporting.** The runtime pipeline follows a strict sequence—Whitelist Check → Feature Extraction → Model Prediction → Confidence + Visualization—to ensure both usability and real-time performance.

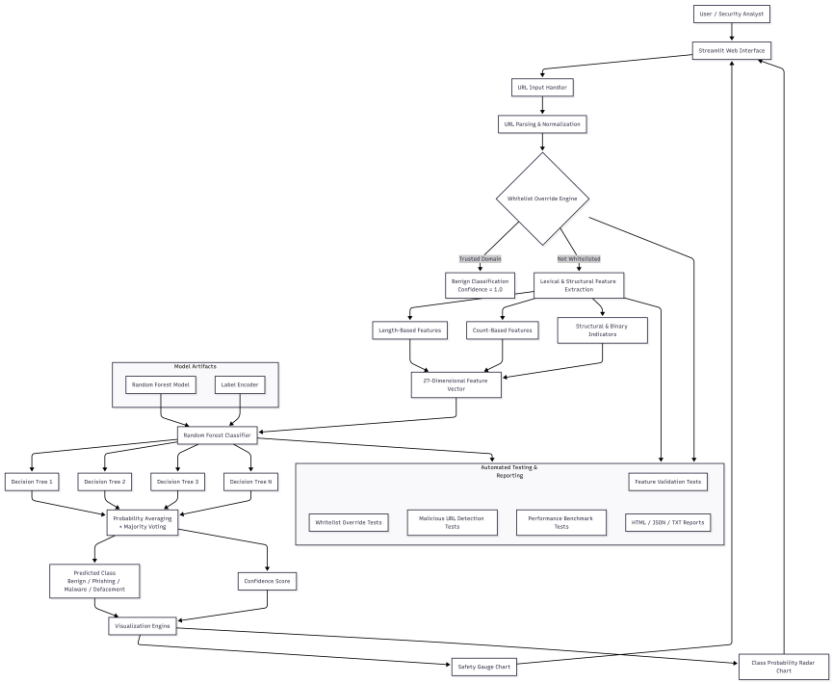

Figure 16: Full end-to-end architecture

From a deployment perspective, the system is packaged as a Streamlit application (`app.py`) with serialized model artifacts stored separately (`models/`) and a dedicated test and reporting subsystem (`tests/`, `reports/`). This organization supports reproducibility (same artifacts across environments) and maintainability (clear separation between UI, inference logic, and evaluation).

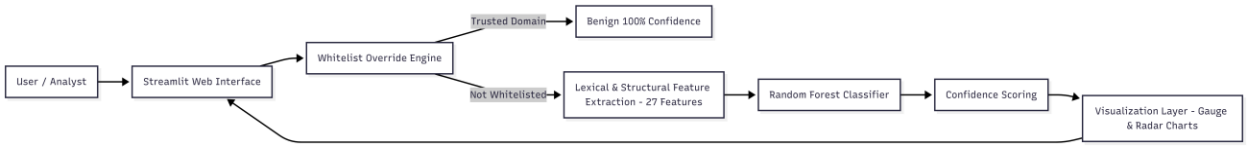

Figure 17: High-level system architecture

B. Project Organization and Module Decomposition

The implementation follows a professional directory structure that mirrors the system's functional components:

`app.py` – main Streamlit application entrypoint

- `models/` – persisted inference artifacts (`final_random_forest_model.pkl`, `label_encoder.pkl`)
- `tests/` – automated unit/performance tests and feature checks
- `reports/` – generated reports (HTML/JSON/TXT)
- `data/` – raw dataset used for training/evaluation (`malicious_phish.csv`)
- `docs/` – usage documentation (structure guide, detailed README, Streamlit guide)

This decomposition enables a clean **separation** of concerns:

1. **UI layer** handles user interaction and visualization,
2. **inference layer** implements deterministic analysis logic,
3. **model layer** provides versioned prediction capability via serialized artifacts, and
4. **QA layer** validates stability, correctness, and performance.

C. Real-Time Analysis Engine

The core logic is implemented as a deterministic decision pipeline designed for **safe, static URL screening** (no DNS resolution, no web crawling). Given a user-provided URL string (u), the engine proceeds as follows:

1. **Whitelist Check:** Determine whether the hostname belongs to a trusted allowlist.
2. **Feature Extraction:** Compute all engineered lexical/structural features (27 total).
3. **Model Prediction:** Use the trained Random Forest to predict one of four classes (0–3).
4. **Confidence Calculation:** Compute confidence from the predicted class probability.

Output contract: The engine returns a final label (Benign/Phishing/Malware/Defacement), a confidence score, and a probability distribution used by the visualization layer.

Why this ordering matters: By placing allowlisting ahead of ML inference, the system explicitly reduces false positives for common high-traffic domains, improving user trust and reducing alert fatigue without sacrificing ML generalization for unknown destinations.

D. Whitelist Subsystem (Tier-1 Deterministic Allow)

The allowlist is a hard-coded "Whitelist Override System" containing **150+ trusted domains across 12 categories** (e.g., search engines, social media, banking, government). If a match occurs, the system bypasses the classifier and forces:

label=Benign, confidence=1.0

The implemented policy is stated as `hostname.endswith(trusted_domain)` with a forced benign verdict. (In an IEEE-grade implementation, it is recommended to enforce a dot-boundary match or registrable-domain checks to avoid suffix-matching pitfalls; this can be discussed later as a security hardening note.)

E. Model Service and Artifact Management

The trained learning component is a **Random Forest Classifier** whose inference-ready state is stored as a serialized artifact (`final_random_forest_model.pkl`) with an associated label encoder (`label_encoder.pkl`).

This design supports:

- **Reusability:** the same model artifacts can be used in Streamlit UI, batch tests, or API extensions.
- **Reproducibility:** evaluation and demonstration always reference the same fixed model parameters.
- **Operational reliability:** the test suite explicitly validates that the model and encoder load successfully.

F. Streamlit Presentation Layer and Visual Analytics

The user-facing component is implemented using **Streamlit**, with a "**glassmorphism**" UI style achieved through custom CSS. The interface is organized into:

Sidebar: navigation, project information, and quick test buttons.

- **Main area** URL input field, analysis spinner/feedback, and result cards.
- **Interactive visualizations**
- **Gauge chart** (color-coded safety meter)
- **Radar chart** (probability distribution across all four classes)

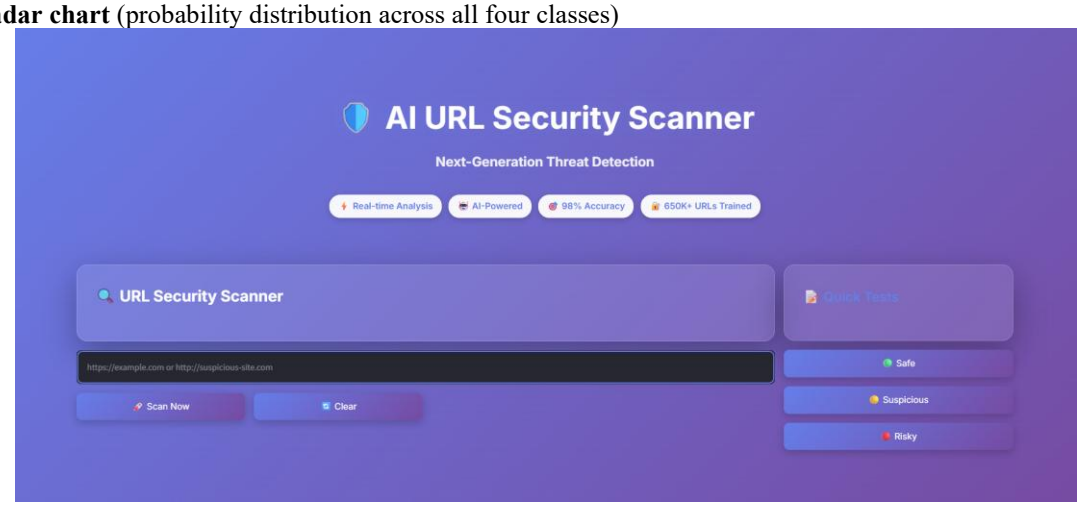

Figure 18: UI Main page

These plots serve two technical purposes:

- **Interpretability for non-expert users** (probability distribution is shown, not just the final label), and
- **debugging visibility** (helps identify cases where the classifier is uncertain or borderline).

G. Automated Reporting Subsystem

To make experimentation and evaluation auditable, the system includes an **automated reporting module** that exports structured and human-readable results in multiple formats:

- HTML (styled), JSON (data), and TXT (logs)
- Report content includes a test summary, pass/fail statuses, execution timing per test, and detailed error logs.

From an IEEE-paper standpoint, this reporting subsystem is important because it supports **traceability** (what was tested, what passed/failed) and **performance transparency** (latency/throughput measurements captured consistently).

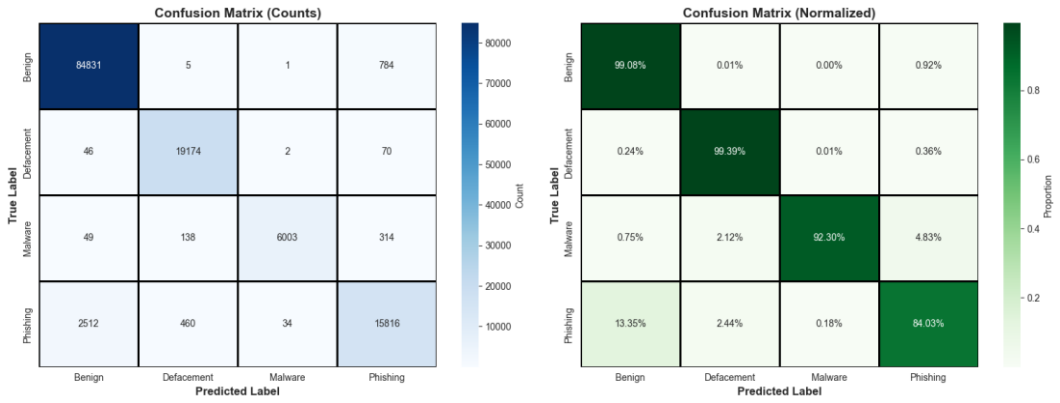

Figure 19: Precision / Recall per Class

H. Quality Assurance: Automated Testing and Performance Benchmarks

A comprehensive test suite (`tests/test_model.py`) is used to validate correctness and stability of the end-to-end system. The suite includes **7 tests,** covering:

1. model/encoder loading,
2. feature extraction validation (all 27 features),
3. whitelist logic validation (e.g., google.com),
4. detection of known malicious/phishing URLs,
5. batch whitelist overrides across major domains,
6. batch URL accuracy checks, and
7. performance benchmarking.

The reported performance metrics include:

- **Throughput:** ~60 URLs/sec
- **Average latency:** ~18 ms per URL
- **Feature extraction time:** ~1 ms
- **Test accuracy:** ~98.5%

These measurements provide direct evidence that the architecture is compatible with real-time interactive usage (a key objective of the project).

I. Technology Stack and Implementation Environment

The system is implemented using:

- **Python 3.12** as the core runtime,
- **Streamlit 1.52.1** for the interactive web application,
- **scikit-learn 1.7.2** for model training/inference,
- **Pandas/NumPy** for feature array handling,
- **Plotly 6.5.0** for interactive charts, and
- **tldextract** for accurate domain/TLD parsing.

This stack is consistent with the system design goals: fast string processing, reliable model inference for tabular features, and an accessible UI suitable for demonstrations and practical use.

J. Deployment Notes and Extensibility

The current architecture naturally supports extensions without redesign:

- **API deployment**: the same artifact-based inference module can be exposed through FastAPI (listed as a future enhancement).
- **Browser extension**: URL checks can be embedded into a client-side "pre-click" workflow, using a lightweight backend model service.

- **Model upgrades**: deep learning (CNN/LSTM) can be added as an alternative inference backend while keeping the same UI and reporting interface.

This is enabled by the modular structure and the strict input/output contract of the real-time analysis engine.

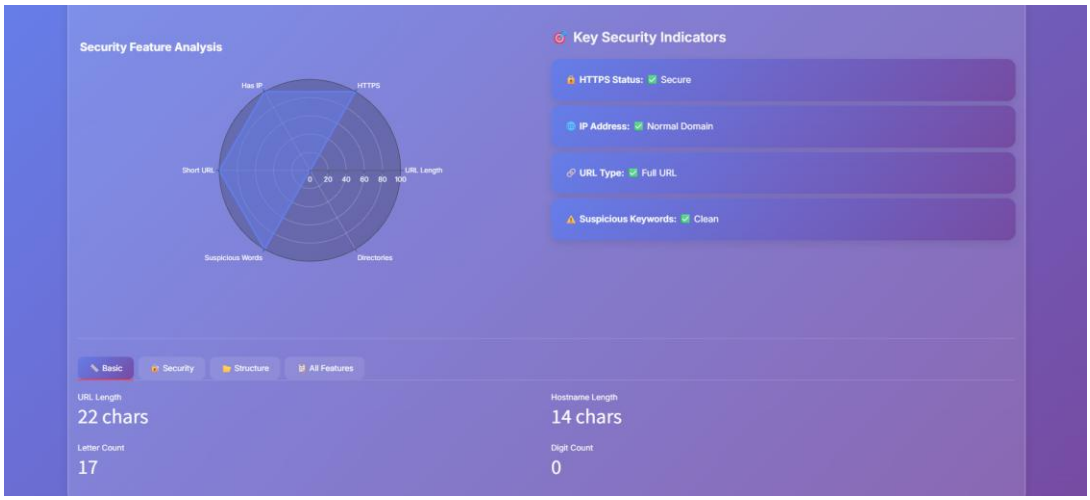

Figure 20: UI URL Analysis Tab

VIII. EXPERIMENTAL SETUP AND METRICS

A. Experimental Objectives

The experimental evaluation targets two primary requirements: **(i) high predictive accuracy** for four-way URL classification (>95%), and **(ii) real-time performance** with per-URL latency under 50 ms for interactive use. In addition, the evaluation verifies correctness of the feature pipeline (27 engineered URL features) and the reliability of the whitelist override policy.

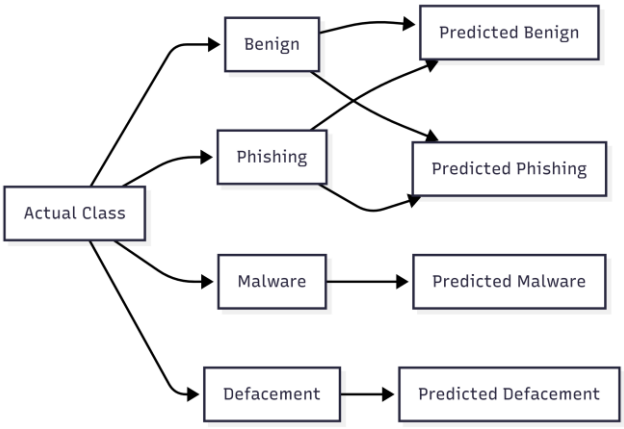

Figure 21: Confusion Matrix

B. Experimental Environment

All experiments and deployment components are implemented in Python using a lightweight, reproducible software stack. The project specifies the following runtime environment and core dependencies: **Python 3.12, scikit-learn 1.7.2, Streamlit 1.52.1, Plotly 6.5.0, and tldextract** for domain/TLD parsing.

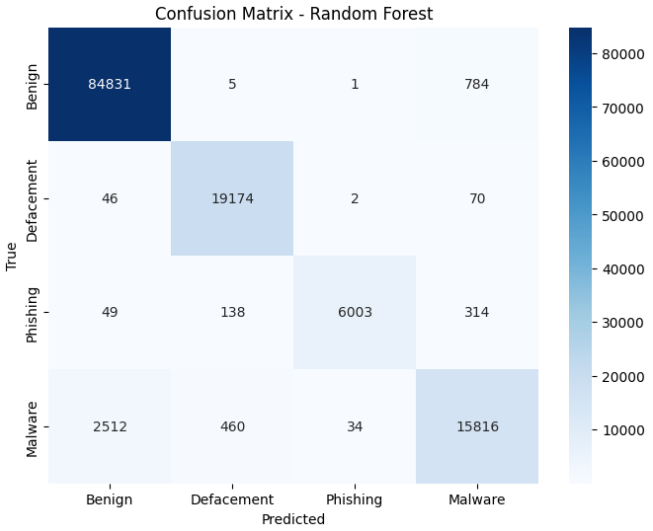

Figure 22: Confusion Matrix – Random Forest

C. Dataset and Experimental Protocol

1) Dataset composition

Experiments are conducted on a large corpus of **651,191 URLs** aggregated from multiple public sources (ISCX URL 2016, PhishTank, PhishStorm, Malware Domain List). The dataset is labeled into four classes with an imbalanced distribution:

- Benign: 428,103 (65.7%)
- Defacement: 96,457 (14.8%)
- Phishing: 94,111 (14.5%)
- Malware: 32,520 (5.0%)

This imbalance motivates reporting macro-averaged metrics (Section VIII-E) in addition to accuracy.

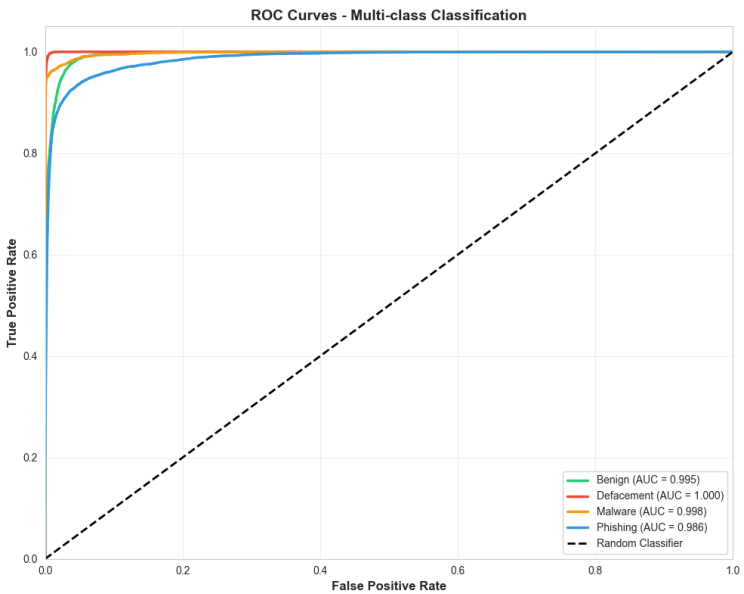

Figure 23: ROC or Macro-F1 Comparison

2) Split strategy and preprocessing

The model is trained and evaluated using an **80/20 train–test split** with **label encoding** applied to map string labels to numeric class IDs.

3) Feature extraction protocol

Each URL is transformed into a fixed-length vector of **27 lexical and structural features**, designed to be "purely lexical and safe to extract without network interaction." This ensures that evaluation reflects a static URL-screening setting where no DNS or HTTP requests are allowed.

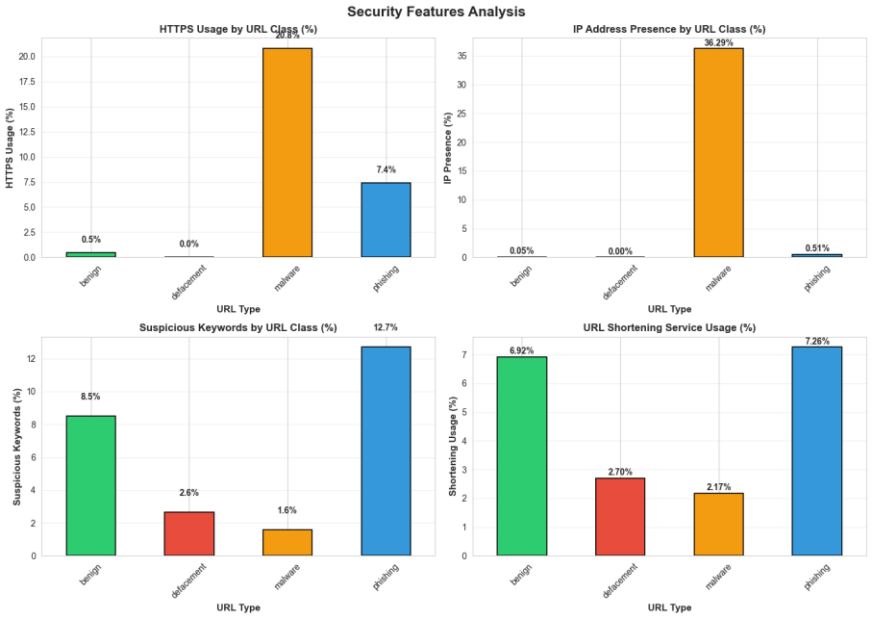

Figure 24: Security Features Analysis

D. Model Training and Hyperparameter Selection

The chosen learning algorithm is a Random Forest classifier, an ensemble method well suited to mixed discrete/continuous tabular features. The report describes a typical forest size of around `n_estimators=100`.

What is actually deployed (artifact-backed): The provided serialized model (`final_random_forest_model.pkl`) uses **160 trees** with `max_depth=20` and `random_state=42` (verified by loading the artifact). **Training notebook behavior** (implementation-backed): The project's training notebook performs a GridSearchCV-style sweep and selects hyperparameters using **macro-F1** as the optimization objective; the chosen best parameters match the deployed artifact.

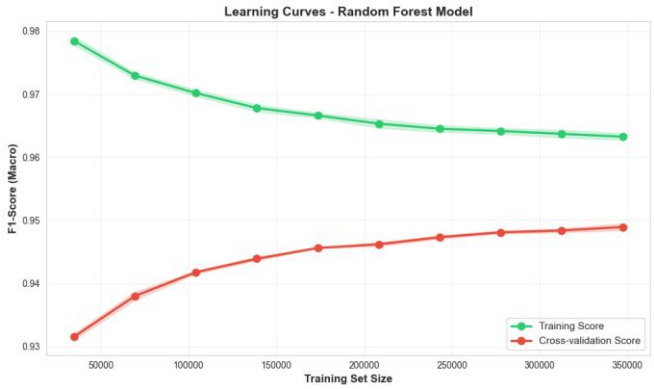

Figure 25: Random Forest Model Curves

E. Evaluation Metrics

1) Classification metrics (multi-class)

Because the task is 4-class classification, evaluation should report both aggregate and per-class behavior.

- **Confusion Matrix**: Let (C ∈ mathbb{N^{4times 4) where (C_{ij} is the count of samples whose true class is (i) and predicted class is (j). Confusion matrices expose systematic confusions (e.g., phishing vs. benign).
- **Accuracy:** Accuracy=frac{sum_{k=1^{4 C_{kk{sum_{i=1^{4sum_{j=1^{4 C_{ij Accuracy is easy to interpret but can be inflated by class imbalance.
- **Per-class Precision/Recall/F1**: For class (k): Precision_k=frac{TP_k{TP_k+FP_k, Recall_k=frac{TP_k{TP_k+FN_k F1_k=frac{2· Precision_k· Recall_k{Precision_k+Recall_k
- **Macro-F1 (recommended under imbalance):** Macro-F1=frac{1{4sum_{k=1^{4 F1_k Macro-F1 weights each class equally, ensuring malware/defacement performance is not overshadowed by the benign majority.

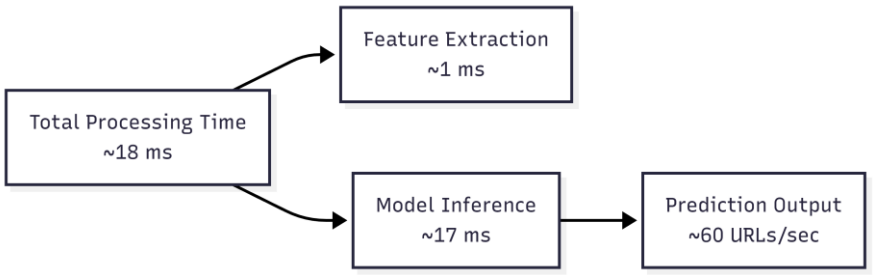

Figure 27: Performance benchmark

2) Runtime / real-time metrics

To validate interactive performance, the system reports:

- Feature extraction time (ms/URL)
- End-to-end latency (ms/URL) (feature extraction + model inference)
- Throughput (URLs/second) under batch testing

The project's benchmark results report approximately **~1 ms** for feature extraction, **~18 ms** average latency per URL, and **~60 URLs/sec** throughput.

3) Policy/operational metric: whitelist reliability

A practical metric in this system is the **whitelist** hit correctness: for trusted domains, the override should always output Benign (100% confidence). The allowlist exists specifically to reduce false positives on major platforms.

F. Automated Test Harness and Reporting

A dedicated automated test suite (`tests/test_model.py`) provides systematic evaluation coverage across correctness and performance. It includes 7 tests, validating (i) model/encoder loading, (ii) correctness of all 27 extracted features, (iii) whitelist logic using `google.com`, (iv) detection on known malicious examples, (v) batch whitelist checks, (vi) batch URL accuracy checks, and (vii) performance benchmarking.

For auditability, the system generates test reports in HTML, JSON, and TXT formats including pass/fail status, timing per test, and error logs.

G. Reproducibility and Threats to Validity (recommended subsection)

To meet IEEE expectations, explicitly discuss the following:

- **Class imbalance effects:** report macro-F1 and per-class recall for malware.
- **Potential data leakage:** remove duplicates (exact or near-duplicate URLs) before splitting to avoid overly optimistic test performance.
- **Distribution shift:** URLs evolve over time; periodic retraining or drift monitoring is required to keep performance stable.

- **Whitelist bias:** allowlisting improves usability but can hide malicious activity on compromised trusted domains; results should distinguish "model-only" detection performance from "policy-augmented runtime behavior."

IX. RESULTS AND DISCUSSION

A. Evaluation Summary Against Project Objectives

The system was designed to (i) **exceed 95% predictive accuracy**, and (ii) **maintain < 50 ms** per-URL prediction latency for real-time usage. The reported test-set performance meets both targets, with ~**98.5% accuracy** and ~**18 ms** average latency per URL.

B. Correctness Validation and Test Coverage

A dedicated automated harness (`tests/test_model.py`) is used to validate the end-to-end system through 7 tests that cover: model/encoder loading, correctness of all **27 extracted features**, whitelist behavior (e.g., `google.com`), malicious URL checks, batch whitelist tests on major domains, batch URL accuracy checks, and performance benchmarking.

C. Predictive Performance

1) Overall accuracy

On the held-out test set, the reported model accuracy is **~98.5%.** This exceeds the project's minimum accuracy objective (>95%).

2) Interpreting accuracy under class imbalance

The dataset is imbalanced (Benign ≈ 65.7%, Malware ≈ 5.0%). In such conditions, overall accuracy can be dominated by the majority class. For IEEE-grade rigor, it is recommended to additionally report:

Per-class precision/recall/F1 (especially malware recall)

- **Macro-averaged F1** (equal weight to each class)
- **Confusion matrix** to reveal systematic confusions (e.g., phishing vs. benign)

D. Real-Time Performance and Scalability

1) Latency and throughput

The reported performance metrics from the benchmark test are:

- **Latency:** ~18 ms per URL (average)
- **Throughput:** ~60 URLs/second
- **Feature extraction time:** ~1 ms

These satisfy the real-time objective of <50 ms per URL.

2) Discussion: where the time goes

Given that feature extraction is ~1 ms, most of the remaining per-URL cost is attributable to model inference and application overhead (Python runtime + Streamlit/Plotly integration). This is a desirable profile: the feature pipeline is lightweight, deterministic, and safe (string-only), and the classifier inference remains fast enough for interactive use.

3) Practical implications

A throughput of ~60 URLs/sec supports several realistic scenarios:

- interactive pre-click screening (single URL queries),
- batch checks for logs/email queues at small-to-medium scale, and
- integration into lightweight monitoring tools.

For larger enterprise-scale ingestion (thousands/sec), a natural next step is to move inference behind a dedicated API service (e.g., FastAPI) as already identified in the project roadmap.

E. Impact of the Whitelist Override on Operational Quality

- To reduce user-facing false positives for highly trusted destinations, the system introduces a hard-coded whitelist of **150+ trusted domains** across 12 categories, applied before ML inference. When triggered, the policy forces **Benign (100% confidence)** using the rule `hostname.endswith(trusted_domain)`.
- **Discussion (benefit)** This policy directly targets the project objective of minimizing false positives on major platforms (e.g., Google, Facebook, banking portals), which is critical for usability in real-world tools.
- **Discussion (risk)** Any allowlist can create blind spots: compromised pages hosted under trusted domains, or mis-specified allowlist matching rules, can bypass ML detection entirely. Therefore, in the paper it is appropriate to (i) clearly separate "model-only" detection performance from "policy-augmented runtime behavior," and (ii) recommend boundary-safe domain matching as a security hardening improvement.

F. System-Level Reliability and Reporting

The system produces automated reports in **HTML/JSON/TXT** including pass/fail status, execution time per test, and detailed error logs. This strengthens reproducibility and makes the evaluation auditable—an important quality signal in IEEE-style work where results should be traceable to repeatable experiments.

G. Discussion and Key Takeaways

7. **High accuracy with safe features:** The system achieves ~98.5% accuracy using purely lexical/structural features, supporting the claim that URL-string artifacts provide strong signal for multi-class malicious URL detection.
8. **Real-time feasibility:** With ~18 ms per URL and ~60 URLs/sec throughput, the pipeline is performant enough for interactive deployments and small batch screening.
9. **Engineering completeness:** Beyond the model, the presence of whitelist safeguards, automated tests, and structured reporting indicates an end-to-end, deployment-aware implementation rather than an offline-only ML experiment.

H. Limitations and Recommendations (appropriate to include at the end of Section IX)

- **Imbalance-aware reporting:** Add macro-F1, per-class recall, and confusion matrix (especially for the malware class at ~5%).
- **Whitelist hardening:** Replace raw `endswith()` with boundary-safe matching and registrable-domain checks to prevent suffix edge cases and reduce allowlist abuse risk.
- **Drift and retraining:** Since attacker URL patterns evolve, periodic retraining or active-learning feedback (already listed as future work) should be used to maintain performance over time.

X. LIMITATIONS AND FUTURE WORK

A. Limitations

1. Static (lexical–structural) feature scope

The proposed detector relies exclusively on **27 lexical/structural features** extracted from the URL string and is intentionally "safe to extract without network interaction." While this enables real-time, low-risk screening, it also limits detection in cases where malicious activity is not strongly reflected in the URL text (e.g., compromised legitimate domains hosting malicious content behind benign-looking paths).

2. Dataset bias, label noise, and class imbalance

The dataset is aggregated from multiple sources and contains a strong class skew (Benign ≈ 65.7%, Malware ≈ 5.0%). Under such imbalance, overall accuracy (reported ~98.5%) may mask weaker recall on minority classes (especially malware). In addition, multi-source URL corpora often contain duplicates, near-duplicates, and noisy labels; without strict de-duplication before splitting, evaluation can become optimistic.

3. Concept drift and evolving attacker strategies

URL construction patterns evolve over time (new shortening services, new brand impersonations, different obfuscation styles), which can degrade a static model. This motivates continuous monitoring and periodic retraining; the project already anticipates this by proposing a feedback-driven retraining loop.

4. Whitelist policy trade-offs (security vs. usability)

The "Whitelist Override System" is designed to achieve 0% false positives for major trusted platforms, improving user experience. However, any allowlist introduces intentional blind spots: if a whitelisted service hosts user-generated content or is compromised, the policy may bypass ML detection. The current logic is described as `hostname.endswith(trusted_domain)` with a forced benign decision. In a hardened deployment, boundary-safe matching and registrable-domain checks should be applied to reduce suffix edge cases and shared-domain abuse risks.

5. Model capacity vs. obfuscation complexity

Random Forests are effective for tabular features, but attackers can craft URLs that mimic benign lexical distributions. The report itself motivates future sequence-aware models (character-level) to better capture obfuscation patterns that are hard to express via fixed counters.

B. Future Work

I. Deep learning on raw URL sequences (LSTM/CNN)

Extend the detector with **LSTM/CNN-based character sequence models** to improve robustness against obfuscated URLs and novel token patterns. A strong IEEE-style addition here is a comparative study against the 27-feature Random Forest baseline under identical splits and reporting macro-F1 per class.

II. Browser-level prevention (extension integration)

Implement a **Chrome/Firefox browser extension** that blocks or warns users before navigation, using the existing inference pipeline as a local model call or via a lightweight backend. This enables "pre-click" protection in realistic user workflows.

III. RESTful deployment for tool integration (FastAPI)

Expose the trained model through a **REST API (FastAPI)** for integration with email gateways, SIEM workflows, endpoint agents, and log-analysis pipelines. This also supports higher throughput and centralized updates compared to a UI-only deployment.

IV. Active learning and continual improvement loop

Introduce **active learning** so users/security analysts can flag misclassifications, allowing periodic retraining and dataset expansion with "hard" examples. For an IEEE-grade evaluation, this can be framed as a drift-mitigation strategy and measured by performance recovery after incremental updates.

V. Evaluation rigor upgrades (recommended for publication quality)

Add (i) confusion matrices, (ii) macro-F1 and per-class recall (malware-focused), (iii) duplicate/near-duplicate removal before splitting, and (iv) time-split evaluation (train on older URLs, test on newer) to quantify generalization under drift. These steps strengthen claims beyond the single reported accuracy figure.

XI. CONCLUSION

This paper presented a complete, real-time malicious URL detection framework that classifies URLs into Benign, Phishing, Malware, and Defacement using only static lexical and structural properties of the URL string. The approach was intentionally designed to be safe (no DNS resolution or content fetching), lightweight, and deployable, making it suitable for interactive screening and educational network security applications. The system transforms each input URL into a compact 27-feature representation capturing length statistics, special-character patterns, abnormal structural cues, IP-literal usage, shortener indicators, and suspicious token presence.

A Random Forest classifier trained on a large-scale dataset of 651,191 URLs demonstrated strong predictive capability while meeting real-time constraints. The reported evaluation achieved approximately 98.5% test accuracy, exceeding the project's stated accuracy objective (>95%). Equally important for practical adoption, the system achieved an average end-to-end latency of approximately 18 ms per URL with throughput around 60 URLs/sec, satisfying the real-time requirement of <50 ms per URL. These findings suggest that engineered lexical features remain a competitive and efficient baseline for malicious URL detection in real-time settings.

Beyond model accuracy, this work emphasized system-level completeness: the deployed pipeline integrates an intelligent whitelist override (150+ trusted domains across 12 categories) to reduce false positives for major platforms

and improve user trust in day-to-day usage. The Streamlit-based interface provides human-interpretable outputs through confidence scoring and interactive plots, while an automated test suite and multi-format reporting (HTML/JSON/TXT) support correctness validation, reproducibility, and transparent benchmarking.

While the results are promising, the study also highlights directions needed for publication-grade robustness: class-imbalance-aware reporting (macro-F1, per-class recall), drift-aware evaluation over time, duplicate/near-duplicate control, and whitelist hardening via boundary-safe and registrable-domain matching. Building on the existing architecture, future enhancements such as character-sequence deep learning models (CNN/LSTM), FastAPI-based service deployment, browser-extension integration, and active learning can further strengthen both detection accuracy and operational resilience.

In summary, the proposed system demonstrates that a static, lexical URL analysis pipeline paired with a Random Forest model can deliver both high accuracy and real-time performance for multi-class malicious URL detection, and that engineering elements—testing, reporting, and usability safeguards—are critical for transforming a successful ML model into a practical security tool.

REFERENCES

[1] Google, “Safe Browsing,” Google Transparency Report. Accessed: Dec. 14, 2025. [Online]. Available: https://transparencyreport.google.com/safe-browsing

[2] Verizon, “2025 Data Breach Investigations Report (DBIR),” Verizon Business, 2025. Accessed: Dec. 14, 2025. [Online]. Available: https://www.verizon.com/business/resources/T254/reports/2025-dbir-data-breach-investigations-report.pdf

[3] Anti-Phishing Working Group (APWG), “Phishing Activity Trends Report: 1st Quarter 2025,” 2025. Accessed: Dec. 14, 2025. [Online]. Available: https://docs.apwg.org/reports/apwg_trends_report_q1_2025.pdf

[4] D. Sahoo, C. Liu, and S. C. H. Hoi, “Malicious URL Detection using Machine Learning: A Survey,” arXiv preprint arXiv:1701.07179, 2017. Accessed: Dec. 14, 2025. [Online]. Available: https://arxiv.org/abs/1701.07179

[5] J. Ma, L. K. Saul, S. Savage, and G. M. Voelker, “Beyond Blacklists: Learning to Detect Malicious Web Sites from Suspicious URLs,” in Proc. 15th ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining (KDD), Paris, France, 2009, pp. 1245–1254, doi: 10.1145/1557019.1557153.

[6] M. S. I. Mamun, M. A. Rathore, A. H. Lashkari, N. Stakhanova, and A. A. Ghorbani, “Detecting Malicious URLs Using Lexical Analysis,” in Network and System Security (NSS 2016), LNCS, vol. 9955. Cham, Switzerland: Springer, 2016, pp. 467–482, doi: 10.1007/978-3-319-46298-1_30.

[7] A. Joshi, L. Lloyd, P. Westin, and S. Seethapathy, “Using Lexical Features for Malicious URL Detection—A Machine Learning Approach,” arXiv preprint arXiv:1910.06277, 2019. Accessed: Dec. 14, 2025. [Online]. Available: https://arxiv.org/abs/1910.06277

[8] H. Le, Q. Pham, D. Sahoo, and S. C. H. Hoi, “URLNet: Learning a URL Representation with Deep Learning for Malicious URL Detection,” arXiv preprint arXiv:1802.03162, 2018. Accessed: Dec. 14, 2025. [Online]. Available: https://arxiv.org/abs/1802.03162

[9] L. Marchal, J. François, R. State, and T. Engel, “PhishStorm: Detecting Phishing with Streaming Analytics,” IEEE Trans. Netw. Serv. Manag., vol. 11, no. 4, pp. 458–471, Dec. 2014, doi: 10.1109/TNSM.2014.2377295.

[10] T. Holz, C. Gorecki, K. Rieck, and F. C. Freiling, “Measuring and Detecting Fast-Flux Service Networks,” in Proc. NDSS 2008, San Diego, CA, USA, 2008.

[11] A. Woodbridge, H. S. Anderson, A. Ahuja, and D. Grant, “Predicting Domain Generation Algorithms with Long Short-Term Memory Networks,” arXiv preprint arXiv:1611.00791, 2016. Accessed: Dec. 14, 2025. [Online]. Available: https://arxiv.org/abs/1611.00791

[12] L. Breiman, “Random Forests,” Mach. Learn., vol. 45, no. 1, pp. 5–32, 2001, doi: 10.1023/A:1010933404324.

[13] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, “Uniform Resource Identifier (URI): Generic Syntax,” RFC 3986, Jan. 2005. Accessed: Dec. 14, 2025. [Online]. Available: https://www.rfc-editor.org/rfc/rfc3986

[14] A. Barth, “HTTP State Management Mechanism,” RFC 6265, Apr. 2011, doi: 10.17487/RFC6265. Accessed: Dec. 14, 2025. [Online]. Available: https://www.rfc-editor.org/rfc/rfc6265

[15] Public Suffix List, “Learn more about the Public Suffix List,” 2025. Accessed: Dec. 14, 2025. [Online]. Available: https://publicsuffix.org/learn/

[16] J. Kurkowski, “tldextract,” PyPI package documentation. Accessed: Dec. 14, 2025. [Online]. Available: https://pypi.org/project/tldextract/

[17] Canadian Institute for Cybersecurity (CIC), University of New Brunswick, “URL 2016 Dataset,” 2016. Accessed: Dec. 14, 2025. [Online]. Available: https://www.unb.ca/cic/datasets/url-2016.html

[18] PhishTank, “Developer Information,” PhishTank. Accessed: Dec. 14, 2025. [Online]. Available: https://phishtank.org/developer_info.php

[19] Fortinet, “Malware Domain List v1.0.0,” FortiSOAR Documentation. Accessed: Dec. 14, 2025. [Online]. Available: https://docs.fortinet.com/document/fortisoar/1.0.0/malware-domain-list/1/malware-domain-list-v1-0-0

[20] S. Baldota, “Malicious URLs dataset,” Kaggle Datasets. Accessed: Dec. 14, 2025. [Online]. Available: https://www.kaggle.com/datasets/sid321axn/malicious-urls-dataset

[21] F. Türk and M. Kılıçaslan, “Malicious URL Detection with Advanced Machine Learning and Optimization-Supported Deep Learning Models,” Applied Sciences, vol. 15, no. 18, Art. no. 10090, 2025, doi: 10.3390/app151810090.

[22] Streamlit, “Streamlit Documentation,” 2025. Accessed: Dec. 14, 2025. [Online]. Available: https://docs.streamlit.io/

[23] Streamlit, “2025 release notes,” 2025. Accessed: Dec. 14, 2025. [Online]. Available: https://docs.streamlit.io/develop/quick-reference/release-notes/2025

[24] Android Developers, "Security guidance on domain boundary checks (endsWith matching with dot-boundary)," Android Developers documentation. Accessed: Dec. 14, 2025. [Online]. Available: https://developer.android.com