Phishing URL Detection

Francisco Ferreira
DETI, UA
124467
ftferreira@ua.pt

Sara Almeida
DETI, UA
108796
sarafalmeida@ua.pt

***Abstract*—This project was developed in the context of the UC *Aprendizagem Aplicada à Segurança* and addresses the problem of detecting phishing attempts by classifying URLs as benign or phishing using only the URL string, without fetching page content. We evaluate lightweight lexical approaches based on engineered URL features (length statistics, character composition, hostname structure, keyword indicators, and entropy-like complexity signals) and compare them against each other and against a character-level CNN baseline and a hybrid ensemble. Experiments were conducted on a Kaggle dataset with 450,176 labeled URLs and a test set of 78,214 samples. Among classical models, XGBoost provided the best trade-off between discrimination and deployability. We further applied isotonic probability calibration to enable reliable threshold-based decisions in a browser setting. The selected deployed model, `XGB_calibrated`, achieves very low false-positive rates at the standard 0.5 evaluation threshold (FP=1 on the test set) with phishing-class precision of 0.9999, recall of 0.9512, and F1-score of 0.9750. Finally, we deliver a working prototype composed of a local Flask API and a Manifest V3 Chrome Extension implementing a three-tier policy (safe/warning/unsafe) to provide real-time user feedback during browsing.**

***Index Terms*—phishing detection, malicious URLs, lexical features, XGBoost, probability calibration, browser extension, security analytics**

I. Introduction

Phishing remains one of the most prevalent and operationally impactful attack vectors because it scales cheaply, targets both individuals and organizations, and is frequently the first step for personal data or credential theft and follow-on compromise. The Anti-Phishing Working Group (APWG) reported 1,003,924 phishing attacks in the first quarter of 2025 and 1,130,393 in the second, and they also stated that the number of phishing attacks has risen steadily over the last year, indicating that phishing activity continues at very high levels. [1]

Moreover, attackers increasingly leverage automation and AI-enabled lures to improve realism and conversion.

Against this background, fast and lightweight URL-based detection is a crucial area to explore as it can be deployed early in security workflows without the cost and privacy implications of fetching page content.

A. Problem

In this context, this project addresses the problem of **automatically classifying URLs as benign or phishing** using **only lexical properties of the URL string**, without downloading the web page or analyzing server-side content.

This report details the development of our solution for this problem, including different tested models and respective results. The code repository can be found here: https://github.com/ftferreira02/AAS-Project.

B. Objectives

For the development of this work, the following objectives were formulated:

1) Find a good-quality training dataset of a sufficient size that is well labeled with benign VS phishing URLs;
2) Engineer a set of lightweight lexical features that capture URL structure and commons phishing artifacts (length, token patterns, character composition, entropy-like signals, and suspicious indicators);
3) Train and compare multiple machine learning models suitable for this feature space, selecting a final approach based on metrics performance and error analysis;
4) Deliver an executable prototype of a usable browser extension that exposes the detector via a simple API, enabling real-time URL scoring during browsing.

C. Document Structure

The remainder of this report is organized as follows: Section II reviews prior approaches to phishing URL detection. Section III describes the dataset, preprocessing, feature extraction, details the models evaluated and the architecture. Section IV details the implementation going through the folder and code structure, key libraries used, API and extension details and challenges encountered during development. Section V present the results and related discussion, comparisons between models, security analysis and final model selection. Finally, section VI concludes this report and outlines future work directions.

II. Related Work

Phishing detection has traditionally relied on *blacklists* and rule-based heuristics. While effective for known campaigns, these approaches struggle with *zero-hour* attacks (newly created domains/URLs) and require continuous manual updates. As a result, many systems moved toward machine learning models that infer malicious intent directly from URL patterns without waiting for reporting or human-crafted signatures [2], [3].

A widely adopted research direction is *lexical URL classification*, where models use features computed from the URL string (e.g., length statistics, token composition, suspicious

substrings, number of delimiters, presence of IP addresses). Blum *et al.* demonstrated that lexical features combined with online learning can detect phishing URLs in a fast, pre-click setting, making them suitable for real-time decision-making [2]. More recent work continues to show that purely static lexical approaches can be effective and practical, especially when low latency and safe analysis (no page crawling) are required [3].

Alongside feature engineering, deep learning methods have been proposed to reduce manual design effort by learning representations directly from raw URL strings. Character-level and hybrid character/word models can capture obfuscation tricks (e.g., homoglyphs, unusual character sequences, adversarial path/query patterns) that are hard to enumerate as handcrafted features. Texception is an example of a character/word-level deep learning architecture designed for phishing URL detection, motivated by internet-scale constraints and minimal-latency pipelines [4]. Other architectures explore temporal or convolutional sequence modeling over URL tokens/characters, such as temporal convolutional networks for URL classification [6].

A complementary trend is *hybrid/ensemble* systems that combine the strengths of both families: engineered lexical features (interpretable, stable, cheap) and neural sequence models (expressive, robust to obfuscation). Recent ensemble designs explicitly fuse a feature-based learner (e.g., gradient boosting) with a character-level CNN to improve robustness and reduce false positives compared to either model alone [9]. This motivates architectures like the one we experimented and will detail further in this report, where separate lexical and character-level models are combined in a weighted ensemble.

Finally, deployment-oriented phishing detectors often require calibrated risk scores rather than raw classifier margins, particularly when multiple actions (allow, warn, block) are triggered by thresholds. Probability calibration methods such as Platt scaling and isotonic regression have been shown to significantly improve the reliability of predicted probabilities, which is important when decisions depend on score cutoffs [5]. Dataset quality is also a known bottleneck: public feeds such as PhishTank provide large collections of verified phishing URLs [7], but recent work highlights issues like label noise, staleness, leakage, and unrealistic base rates, motivating newer curated benchmarks such as PhreshPhish [8].

III. Methodology

A. Data Acquisition

The dataset used for training and evaluation was the one indicated by the professor, specifically the "Malicious and Benign URLs" dataset, which was obtained from Kaggle on January 2, 2026 and can be accessed here [10] .

The dataset is provided as a CSV file containing 4 columns:

- *Unnamed: 0*: index;
- *url*: URL strings;
- *label*: benign or malicious;
- *result*: integer - 0 for benign and 1 for malicious

After the header with the column names, this dataset contains 450,176 labelled URLs, with 345,738 benign (76.8%) and 104,438 malicious (23.2%) samples. This dataset is compliant with the lexical-only approach of this project, as we rely exclusively on the URL text.

1) Alternative Datasets Considered: Before selecting the final dataset used in this project, we considered and experimentally evaluated two other datasets from Kaggle that revealed poor results.

The first was "Phishing Site URLs" [11] , which is distributed as a two-column CSV with approximately 549,346 entries and, besides the URL string column, has the *label* column, described as a prediction of the URLs, which contains a binary label - *good* vs *bad*. This dataset had limited transparency about how the "bad" URLs were sourced and validated. Moreover, a manual inspection suggested many benign examples did not resemble commonly observed real-world benign domains, indicating potential sampling bias. Also, the label provenance was unclear and the results we obtained with this data were inconsistent.

The second dataset considered was "Malicious URLs dataset" [12], which contains 651,191 URLs and is explicitly multi-class, including benign/safe, defacement, phishing, and malware categories (e.g., 428,103 benign; 96,457 defacement; 94,111 phishing; 32,520 malware). This dataset was discarded because while valuable for general "malicious URL" detection, it is not tightly aligned with the project's focus on phishing URL detection. When collapsing multiple heterogeneous classes into a single "malicious" label, we observed reduced discrimination and noisier decision boundaries, likely because "defacement" and "malware" URLs can exhibit different lexical patterns than phishing URLs (and can bias the model toward artifacts that are not representative of phishing).

Finally, we decided on the dataset presented at first in this subsection, suggested by the professor, which presents clearer binary semantics and closer alignment with our project main goal.

B. Preprocessing

1) Dataset-level Preprocessing: The first step was to manually analyze the dataset. During this process we noticed a *typo* in the data: 42 benign URL entries started with the invalid scheme *httpss* instead of the existent safe protocol *https*. To avoid introducing unnecessary noise, we proceeded to manually correct these cases by replacing *httpss* with *https* prior to training.

After loading the CSV, we remove the index column (`Unnamed: 0`) when present, because it is irrelevant as it is not a predictor. Labels are standardized to a binary format where `0` denotes benign URLs and `1` denotes malicious/phishing URLs; for the selected dataset, this is obtained directly from the `result` column (integer 0/1).

To support a leakage-aware split, each URL is parsed to extract its hostname. For robustness, URLs missing a scheme

are temporarily prefixed with `http://` for hostname parsing. Rows for which a hostname cannot be extracted (empty hostname after parsing) are removed, since they cannot be grouped reliably and may correspond to malformed inputs.

Before splitting, the dataset is sorted by the URL string to ensure deterministic row ordering across runs and environments, independent of CSV read order.

To reduce train/test leakage due to repeated domains, we perform the split using `GroupShuffleSplit` with the hostname as the grouping variable, ensuring that the same hostname does not appear in both training and test sets. We use a single split (`n_splits=1`) with `test_size=0.2` and `random_state=42`. This evaluation setting better approximates real-world deployment, where the detector must generalize to previously unseen hostnames rather than memorizing domain-specific patterns.

2) URL-level Preprocessing: Each URL is normalized to reduce noise and improve consistency across samples. Again, a scheme is enforced (defaulting to `http://` when missing), common tracking parameters (e.g., `utm_*`, `gclid`, `fbclid`) are removed from the query string, and fragments are dropped since they are not sent to servers and often introduce irrelevant variability. The normalized URL is then parsed using standard URL parsing utilities to obtain *scheme*, *netloc* (hostname), *path*, and *query*. We additionally use `tldextract` to decompose the hostname into *subdomain*, *registered domain*, and *suffix/TLD*, enabling location-specific lexical features (e.g., keyword occurrence in subdomain versus registered domain).

C. Feature Extraction

From the normalized and parsed URL components described in the previous section, we compute the final lexical feature vector that combines structural, character-composition, indicator, and complexity signals that are commonly associated with phishing URLs. Table I summarizes the main feature groups.

1) Structural and composition features: Phishing URLs frequently exhibit abnormal length and tokenization patterns (e.g., long paths, many delimiters, and digit-heavy strings). To capture these effects we measure URL, hostname, path and query lengths, as well as counts of dots, hyphens, underscores, digits, the `@` symbol, and a broader "special character" count. To reduce sensitivity to absolute length, we also include normalized ratios such as the fraction of digits and special characters in the full URL, the fraction of digits in the registered domain, and the proportion of the URL devoted to query parameters.

2) Domain, protocol, and structural indicators: We include several boolean indicators that reflect common phishing tactics: use of plain HTTP (`is_https`), embedding an IP address in the hostname (`has_ip`), punycode usage (`is_punycode`), and a lightweight suspicious-TLD flag (`suspicious_tld`) for a small set of frequently abused suffixes (e.g., `tk`, `xyz`, `top`, `club`, `info`). Additionally, we quantify hostname complexity via the number of subdomain labels and the number of dots inside the subdomain component.

3) Suspicious keyword features: Phishing campaigns often incorporate lures such as `login`, `verify`, `update`, `secure`, `account`, `password`, `confirm`, `signin`, and `banking`. Instead of only checking whether a keyword appears anywhere in the URL, we add location-sensitive indicators that separately capture occurrences in the full hostname (`kw_in_netloc`), the registered domain (`kw_in_registered_domain`), the subdomain (`kw_in_subdomain`), the path (`kw_in_path`), and the query string (`kw_in_query`). We also count total keyword occurrences (`kw_count`) to capture repeated "pressure" terms.

4) Entropy-based complexity: To approximate the degree of randomness/obfuscation in URLs, we compute Shannon entropy over three different substrings: the full URL, the hostname, and the concatenation of path and query. For a string s, entropy is computed as:

$$H(s) = -\sum_{c \in \mathcal{A}} p(c) \log_2 p(c), \tag{1}$$

where $p(c)$ is the empirical frequency of character c in s. Higher entropy can indicate algorithmically generated tokens or heavy obfuscation, which are common in phishing and other malicious URLs.

D. Models Experimented

To assess the suitability of different learning paradigms for lightweight lexical URL-based phishing detection, we evaluated three classical supervised models on the handcrafted feature vector (Random Forest, Logistic Regression and XGBoost), as well as calibrated variants of the tree-based models. In addition, we trained a character-level CNN baseline directly on URL strings to compare feature engineering against representation learning.

1) Random Forest (RF): Random Forest combines the decisions of many different decision trees, each trained on a random sample of the data and a random subset of the features, which helps it learn complex "if–then" patterns in URL lexical signals. We train an `RandomForestClassifier` with `n_estimators=100`, `random_state=42`, and `class_weight='balanced'`. The model is also parallelized with `n_jobs=-1`. Random Forests are robust for heterogeneous tabular features and often perform well when interactions between multiple simple indicators (e.g., length, special characters, keyword presence) are relevant.

2) Logistic Regression (LogReg): Logistic Regression was used as a strong linear baseline. In our implementation, the model is trained within a pipeline that first standardizes the feature space using `StandardScaler` and then fits a `LogisticRegression` classifier with `max_iter=2000`. To account for class imbalance, we use `class_weight='balanced'`, which re-weights classes inversely proportional to their frequency. This model provides an interpretable reference point and helps quantify how much

TABLE I
SUMMARY OF LEXICAL FEATURES EXTRACTED FROM EACH URL.

Group	Features (examples)
Structure	`url_length`, `hostname_len`, `domain_length`, `path_length`, `query_length`
Character counts	`count_dots`, `count_hyphens`, `count_underscores`, `count_digits`, `count_at`, `count_special`
Ratios	`ratio_digits`, `ratio_special`, `domain_digit_ratio`, `param_ratio`
Domain/protocol indicators	`is_https`, `has_ip`, `is_punycode`, `suspicious_tld`
Hostname structure	`count_subdomains`, `subdomain_dot_count`
Query complexity	`count_params`
Keyword signals	`kw_in_netloc`, `kw_in_registered_domain`, `kw_in_subdomain`, `kw_in_path`, `kw_in_query`, `kw_count`
Entropy/complexity	`entropy_url`, `entropy_host`, `entropy_path_query`

of the detection performance is achievable with a linear decision boundary.

3) XGBoost (XGB): XGBoost is a gradient-boosted decision tree model that iteratively fits trees to correct previous errors, typically providing strong performance on tabular data. We train an `XGBClassifier` with `n_estimators=150`, `learning_rate=0.1`, `max_depth=10`, and `eval_metric='logloss'`. To mitigate class imbalance, we set `scale_pos_weight=3`, reflecting the approximate benign-to-malicious ratio in the dataset. The training is made reproducible with `random_state=42` and parallelized with `n_jobs=-1`.

4) Probability Calibration (RF_calibrated and XGB_calibrated): In security applications, well-calibrated probabilities are important because decisions are often threshold-based (e.g., issuing warnings versus blocking). Therefore, in addition to the base RF and XGB models, we evaluated calibrated variants using `CalibratedClassifierCV` with `method='isotonic'` and `cv=3`. Isotonic calibration is a non-parametric method that can improve probability reliability at the cost of additional training overhead. In our experiments, the calibrated XGBoost model (`xgb_calibrated`) was a particularly strong candidate because it drastically reduced false positives with more meaningful probability scores.

5) Character-level CNN baseline (Char-CNN): To contrast handcrafted lexical features with representation learning, we trained a character-level neural baseline that operates directly on the URL string. Each URL is lowercased and encoded with a character-level tokenizer, producing an integer sequence that is padded or truncated to a fixed maximum length of 200 characters. The network maps this sequence to a phishing probability through a compact 1D convolutional architecture.

Concretely, the model starts with an `Embedding` layer that learns a dense representation for each character token. This is followed by a single `Conv1D` layer (128 filters, kernel size 5) that extracts local character n-gram patterns commonly associated with obfuscation and phishing lures. The resulting feature maps are aggregated with `GlobalMaxPooling1D`, producing a fixed-size representation independent of URL length. Finally, a small fully-connected head (`Dense(64)` with ReLU activation, `Dropout(0.5)`) and a sigmoid output neuron produce the probability of the malicious class.

The model is trained with binary cross-entropy loss for 3 epochs using batch size 64, and the positive class corresponds to phishing/malicious URLs.

6) Hybrid ensemble (lexical model + Char-CNN): In addition to evaluating the lexical models and the Char-CNN independently, we also experimented with a hybrid ensemble approach that combines their outputs. The motivation is that the two models capture complementary information: the lexical-feature model focuses on explicit structural signals (e.g., URL length, suspicious keywords, hostname structure), while the Char-CNN can learn character-level patterns that are harder to express with handcrafted rules (e.g., unusual n-grams, obfuscation, and deliberate misspellings). In the ensemble, each model produces a probability of the URL being malicious, and the final score is computed as a weighted average of the two probabilities. We included this approach because, as mentions in Section II, it has been suggested in prior work as a practical way to improve detection performance by leveraging complementary representations, and we wanted to measure whether a similar gain would be observed in our setting compared to single-model baselines.

7) Rationale for model choice: The selected model set reflects the requirements of our project guidelines and the nature of the problem. Logistic Regression provides a fast and interpretable baseline for the engineered feature space. Random Forest and XGBoost are well-suited for heterogeneous tabular features and can capture non-linear interactions that commonly occur in URL-based phishing signals (e.g., the combination of suspicious keywords with abnormal hostname structure). Calibration was included because the deployed system uses probability thresholds to support different warning levels, and uncalibrated scores may be poorly aligned with operational decision-making. Finally, the Char-CNN baseline was included to assess whether learning directly from character sequences offers additional discriminative power beyond handcrafted lexical features, while still respecting the constraint of using only the URL string as input. In addition, the classical models were prioritised because they match the project requirement of a **lightweight** detector: once trained, they can score URLs

quickly on CPU with a small dependency footprint. The Char-CNN is included as a reference baseline and a tool for the hybrid approach, but it is heavier to deploy due to its deep-learning runtime requirements, which makes it less aligned with the lightweight objective for the final system.

E. Candidate Final System and Selection Criteria

Although multiple models are evaluated in this work (Section III-D), the final deliverable is a **lightweight, real-time** phishing URL detector integrated into a browser extension and operating **only on the URL string**. This imposes practical constraints beyond raw classification performance: the deployed approach should (i) run efficiently on CPU with low latency, (ii) have a small dependency footprint suitable for local deployment, and (iii) output probabilities that are meaningful for a threshold-based risk policy.

Given the user-facing nature of the extension, we explicitly prioritize minimizing false positives for high-severity actions. In practice, repeated false alarms can reduce user trust and lead to warning fatigue, which undermines the usefulness of the system.

Based on these constraints, **lexical-only classical models** (particularly tree-based models such as Random Forest and XGBoost) are treated as the primary deployment candidates. Probability calibration is included because calibrated scores are more suitable for thresholded decision rules than raw margins. Deep-learning (Char-CNN) and hybrid ensemble variants are evaluated as comparative baselines to quantify the potential benefits of representation learning, but they are not assumed to be deployable candidates due to their heavier runtime and operational complexity.

The final model choice and operating point are therefore determined after the empirical comparison in Section V, combining quantitative performance with the above deployment criteria.

F. System Architecture

Figure 1 (see Appendix) illustrates the end-to-end architecture of the proposed phishing URL detection system. The design follows a lightweight client–server pattern: a Chrome Extension provides the user-facing interface and triggers analysis, while a local API performs feature extraction, model inference, and risk-level decision logic.

Client layer (Chrome Extension): When the user navigates to a URL, the extension captures the target URL and (optionally) checks a local cache and allowlist to avoid repeated analysis for known safe domains. If no cached decision exists, the extension sends a `POST` request to the local `/predict` endpoint and awaits a JSON response containing both the risk level and the associated score. Based on the returned level, the extension updates the badge/popup indicator and can render an overlay for warning or unsafe cases.

Analysis layer (Local API): The API is responsible for the complete inference pipeline. First, the input URL is normalized (e.g., enforcing a scheme and removing common tracking parameters) to reduce noise and ensure consistent parsing. The normalized URL then follows the primary (lexical) inference path: feature extractor produces a structured feature vector that is evaluated by the lexical model, which is designed to be lightweight and therefore suitable for real-time use.

Optional Char-CNN branch and fallback behavior: To improve robustness on difficult cases, the architecture also supports the optional character-level model (Char-CNN) developed. When enabled (and when the required model weights/dependencies are available), the normalized URL is additionally passed through a character preprocessing stage (tokenization and fixed-length padding/truncation) and evaluated by the Char-CNN. The API can then fuse the lexical and character predictions via a weighted ensemble. Importantly, this branch is *toggleable*: if the Char-CNN is disabled or unavailable, the system operates in a lexical-only mode without any changes required on the client side.

Decision logic: The API converts the resulting phishing likelihood score into a three-level decision policy suitable for a browser context: *SAFE* for scores below 0.60, *WARNING* for scores in $[0.60, 0.85)$, and *UNSAFE* for scores above 0.85. The API returns the discrete level together with the underlying score, enabling transparent UI feedback and consistent downstream handling by the extension.

Placing feature extraction, model inference, and thresholding inside the API provides a clear separation of concerns: the extension remains thin and responsive, while the analysis logic is centralized and easier to maintain and update.

IV. Implementation

A. Code structure

1) Folder structure: The project is organized into three main components: (i) the local inference API, (ii) the machine learning training/benchmarking pipeline, and (iii) the Chrome Extension. A simplified directory layout is shown below:

```
AAS-Project/
  api/
    app.py # Flask API (/predict, /health)
    requirements.txt # Python deps
  ml/
    train.py # Train lexical models
    features.py # URL FeatureExtractor
    char_cnn.py # Char-CNN (optional)
    benchmark_ensemble.py# Benchmark models
    model.pkl # Default model (if present)
    runs/ # Saved artifacts + metrics
    data/ # dataset.csv
  extension/
    manifest.json # MV3 manifest
    background.js # Service worker
    content.js # Overlay UI
    popup.html, popup.js # Popup UI
    icons/ # Icons
  scripts/
    generate_icons.py # Utilities
  Makefile # Setup/train/run
```

2) Files: Below we briefly describe the purpose of the main files.

- `api/app.py`: Flask application code.
- `ml/features.py`: Implementation of the `FeatureExtractor` used by both training and

inference. This module centralizes URL parsing and lexical feature computation (length and count features, ratios, hostname decomposition with `tldextract`, keyword indicators, and entropy-based signals), ensuring consistency across experiments and deployment.

- `ml/train.py`: Training script for the classical lexical models (e.g., Logistic Regression, Random Forest, XGBoost) and calibrated variants. It loads the dataset, applies preprocessing and hostname-based splitting, fits the selected model(s), and saves the final artifact(s) for later inference.
- `ml/char_cnn.py`: Character-level CNN implementation used as an experimental baseline. It defines the tokenizer/padding strategy and the CNN architecture, and can also save/load weights for reproducible experiments. This component is optional in the deployed system.
- `ml/benchmark_ensemble.py`: Benchmarking script that evaluates multiple trained models on the same test split, computes metrics and confusion matrices, and (when enabled) tests the hybrid ensemble by combining lexical and Char-CNN probabilities using a weighted average.
- `ml/data/`: Storage location for the dataset CSV used in training and evaluation. After training it also stores the feature caches.
- `ml/runs/`: Output directory for experiment artifacts (trained models and metric report json). This supports reproducibility and simplifies comparing multiple runs.
- `extension/manifest.json`: Chrome Extension Manifest V3 file declaring permissions, service worker, content scripts, and UI assets (icons and popup). It is the entry point for installing the extension.
- `extension/background.js`: Service worker that monitors navigation events, applies allowlist/cache logic, calls the local API `/predict`, and updates extension state (badge text/indicator and popup data) based on the returned risk tier.
- `Makefile`: Convenience entry points to set up the environment, run training/benchmarking, and launch the API. This improves reproducibility by standardizing common commands across machines.

B. Key libraries

Backend API. The API is implemented using `Flask` to expose a simple `/predict` endpoint that accepts a URL in JSON format and returns a structured response containing the predicted class, calibrated phishing probability, and a three-level risk label (`safe`, `warning`, `unsafe`). `flask_cors` is used to restrict cross-origin access to extension and localhost origins.

Feature engineering and classical ML. Lexical feature extraction is implemented in Python (standard library URL parsing plus `tldextract` for robust domain decomposition). Feature vectors are managed with `pandas`. Classical models and calibration are implemented with `scikit-learn` (e.g., `GroupShuffleSplit` for hostname-based splitting and `CalibratedClassifierCV` with isotonic calibration), while gradient boosting uses `xgboost` for strong performance on heterogeneous tabular features.

Deep learning (Char-CNN). The character-level CNN is implemented using `TensorFlow/Keras`. To keep the system robust across environments, the CNN component is optional: if `TensorFlow` is not installed or the CNN weights are absent, the API automatically falls back to the lexical model only.

Browser extension. The Chrome Extension uses Manifest V3 with a service worker (`background.js`) for event-driven URL analysis, `chrome.storage` for persisting the allowlist, and a content script (`content.js`) to render an overlay warning for suspicious/unsafe pages. The extension also uses badge text/color updates to provide immediate feedback.

C. API and Extension

The deployed prototype consists of a local inference API and a Chrome Extension that acts as the user-facing client. The extension is responsible for capturing navigation events and rendering feedback, while the API centralizes URL normalization, feature extraction, model inference, and the conversion of a probability score into an operational risk label.

1) Inference API (Flask): The API exposes two endpoints: (i) `/health` for basic availability checks and (ii) `/predict` for URL scoring. The `/predict` endpoint receives a JSON payload containing the target URL and returns a structured response with: the calibrated phishing probability, a predicted class, and a risk tier used by the extension UI. To support safe cross-origin calls from the extension, `flask_cors` restricts requests to `chrome-extension://*` and local development origins.

Upon receiving a URL, the API executes the same preprocessing steps used during training (Section III-B), including enforcing a scheme when absent, removing common tracking parameters, and dropping fragments. The normalized URL is then parsed and mapped into the engineered lexical feature vector (Section III-C). Finally, the trained model produces a probability score $p \in [0, 1]$ representing the estimated likelihood that the URL is phishing.

2) Operational thresholds and risk policy: Rather than using a single binary cutoff, the system applies a three-level policy designed for browser interaction and low false-positive disruption. The API maps the probability score p to a discrete tier as follows:

- **SAFE**: $p < 0.60$
- **WARNING**: $0.60 \leq p < 0.85$
- **UNSAFE**: $p \geq 0.85$

This policy reflects the intended operational behavior of the tool: the `UNSAFE` tier is reserved for high-confidence detections that may justify strong interruption (e.g., blocking or a prominent interstitial), while `WARNING` captures borderline cases where the system signals risk but avoids hard blocking. This design explicitly prioritizes minimizing false positives for

high-severity actions, while still providing useful risk feedback to the user.

For experimental comparability, the confusion matrices reported in Section V are computed at a fixed threshold of 0.5 across all models; this evaluation threshold is not the one used for browser actions.

3) Chrome Extension (MV3): The Chrome Extension is implemented using Manifest V3 and consists of: (i) a service worker (`background.js`) that monitors navigation events and requests predictions, (ii) a content script (`content.js`) that can render an overlay warning on the active page, and (iii) a popup interface (`popup.html`/`popup.js`) that displays the latest score and decision.

When the user navigates to a new page, the service worker obtains the current URL and performs lightweight client-side checks before contacting the API. To reduce latency and avoid repeated queries, the extension maintains (a) a short-lived cache of recent URL decisions and (b) a user-managed allowlist stored in `chrome.storage`. If the URL is not allowlisted and no cache entry exists, the extension sends a `POST` request to `/predict`. The response tier determines the UI behavior: the extension updates the badge and popup for all cases, and for `WARNING` or `UNSAFE` it can inject a visible overlay explaining the risk and the recommended user action.

Example images (Fig. 2, Fig. 3, Fig. 4) of the extension working on the browser can be found in the appendix.

4) Optional Char-CNN support (experimental): In our implementation, the Char-CNN is treated as an optional experimental component. If the deep-learning runtime and model weights are available, the API can execute the Char-CNN for comparative experiments; otherwise, it defaults to the lexical model. This allows reproducible benchmarking without increasing the baseline deployment requirements of the lightweight system.

D. Challenges

1) Dataset consistency issues: Even in the selected dataset, manual inspection uncovered data quality problems that had to be handled explicitly before training. The most concrete example was the presence of benign URLs with the invalid scheme *httpss*. In addition, we ensured that the pipeline handled the dataset schema consistently (e.g., removing index-like columns such as `Unnamed: 0` when present and using the `result` field to obtain a stable 0/1 label mapping).

2) Environment reproducibility and dependency drift: A recurring challenge was keeping a consistent Python environment across machines and collaborators. These differences affected package resolution and resulted in inconsistent behaviour and results.

3) Optional Char-CNN path and lightweight constraints: The Char-CNN was implemented as an experimental baseline, but integrating it cleanly introduced additional complexity because it depends on a deep-learning runtime (TensorFlow/Keras) and follows a different training/inference interface than the scikit-learn models. This created an engineering trade-off: although the hybrid approach (lexical model + Char-CNN) is attractive for experimentation, it increases the dependency footprint and the training time.

V. Results and Evaluation

This section reports the quantitative evaluation of the models described in Subsection III-D. We present standard classification metrics beyond accuracy (Precision, Recall, and F1-score), report confusion matrices, and compare the main baselines (LogReg, RF, XGB) against calibrated variants and the Char-CNN/hybrid experiments. Finally, we discuss the security implications and detail the final model selection decision in the context of a browser-based phishing warning system.

A. Dataset and Experimental Setup

All models were evaluated under the same data split and preprocessing pipeline to ensure a fair comparison. We used the hostname-based split described in Section III-B, applying `GroupShuffleSplit` with `test_size=0.2`, `n_splits=1`, and `random_state=42`, using the extracted hostname as the grouping variable. This resulted in a training set of 371,958 samples and a testing set of 78,214 samples. The class distribution in the test set is 57,898 legitimate URLs (74.0%) and 20,316 phishing URLs (26.0%).

For all classical models (Logistic Regression, Random Forest, XGBoost, and their calibrated variants), training was performed on the engineered lexical feature vectors computed from the normalized URL string, and inference was carried out on the test split using the same feature extraction procedure. For the Char-CNN baseline, the model was trained and evaluated directly on the raw URL strings (lowercased and padded/truncated to a fixed length), using the same train/test partition so that results are comparable.

Model hyperparameters are those listed in Section III-D. All experiments treat `1` as the positive class (phishing/malicious) and `0` as the negative class (benign). Random seeds were fixed whenever supported by the library to improve reproducibility across runs.

B. Evaluation Metrics

1) Precision, Recall and F1-score: Beyond accuracy, we compute precision, recall and F1-score for each class. Since the security-relevant class is the malicious class (`1`), we highlight the metrics for class `1`:

- **Precision (class 1)** measures how many URLs predicted as malicious are actually malicious:

$$\text{Precision}_1 = \frac{TP}{TP + FP}. \tag{2}$$

  High precision means the detector generates fewer false alarms (fewer benign URLs incorrectly flagged).
- **Recall (class 1)** measures how many truly malicious URLs are detected:

$$\text{Recall}_1 = \frac{TP}{TP + FN}. \tag{3}$$

High recall means fewer missed phishing URLs.

- **F1-score (class 1)** is the harmonic mean of precision and recall:

$$\mathrm{F1}_1 = 2 \cdot \frac{\mathrm{Precision}_1 \cdot \mathrm{Recall}_1}{\mathrm{Precision}_1 + \mathrm{Recall}_1}. \tag{4}$$

F1 is useful when we want a single value that balances false positives and false negatives.

2) Accuracy: Accuracy is the fraction of correctly classified URLs:

$$\mathrm{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}. \tag{5}$$

Although we report accuracy for completeness, it can be misleading in imbalanced datasets. Therefore, we emphasize precision, recall and F1-score together with the confusion matrix.

3) Confusion matrix: To make errors explicit, we report the confusion matrix, which counts: (i) true negatives (TN): benign URLs correctly classified as benign, (ii) false positives (FP): benign URLs incorrectly classified as phishing, (iii) false negatives (FN): phishing URLs missed as benign, and (iv) true positives (TP): phishing URLs correctly detected.

4) Macro and weighted averages: In addition to per-class metrics, we also report macro and weighted averages. Macro averaging gives equal weight to each class, while weighted averaging accounts for class support. Reporting both helps interpret performance under class imbalance.

C. Results

Below are presented the results obtained for each model for the evaluation metrics defined above. As detailed in the Subsection IV-A2 this values can be found in the project repo under the directory `ml/runs`.

TABLE II
DETAILED RESULTS FOR RF ON THE TEST SET.

Class / Avg	Precision	Recall	F1	Support
0 (benign)	0.9941	0.9985	0.9963	57898
1 (phishing)	0.9958	0.9831	0.9894	20316
Macro avg	0.9949	0.9908	0.9929	78214
Weighted avg	0.9945	0.9945	0.9945	78214

Accuracy: 0.9945
Confusion matrix: TN=57814, FP=84, FN=344, TP=19972

TABLE III
DETAILED RESULTS FOR LOGREG ON THE TEST SET.

Class / Avg	Precision	Recall	F1	Support
0 (benign)	0.9945	0.9943	0.9944	57898
1 (phishing)	0.9837	0.9843	0.9840	20316
Macro avg	0.9891	0.9893	0.9892	78214
Weighted avg	0.9917	0.9917	0.9917	78214

Accuracy: 0.9917
Confusion matrix: TN=57566, FP=332, FN=319, TP=19997

TABLE IV
DETAILED RESULTS FOR RF_CALIBRATED ON THE TEST SET.

Class / Avg	Precision	Recall	F1	Support
0 (benign)	0.9919	0.9993	0.9956	57898
1 (phishing)	0.9980	0.9766	0.9872	20316
Macro avg	0.9950	0.9879	0.9914	78214
Weighted avg	0.9935	0.9934	0.9934	78214

Accuracy: 0.9934
Confusion matrix: TN=57859, FP=39, FN=475, TP=19841

TABLE V
DETAILED RESULTS FOR XGB ON THE TEST SET.

Class / Avg	Precision	Recall	F1	Support
0 (benign)	0.9957	0.9981	0.9969	57898
1 (phishing)	0.9946	0.9876	0.9911	20316
Macro avg	0.9951	0.9929	0.9940	78214
Weighted avg	0.9954	0.9954	0.9954	78214

Accuracy: 0.9954
Confusion matrix: TN=57789, FP=109, FN=251, TP=20065

TABLE VI
DETAILED RESULTS FOR XGB_CALIBRATED ON THE TEST SET.

Class / Avg	Precision	Recall	F1	Support
0 (benign)	0.9832	1.0000	0.9930	57898
1 (phishing)	0.9999	0.9512	0.9750	20316
Macro avg	0.9916	0.9758	0.9832	78214
Weighted avg	0.9875	0.9873	0.9872	78214

Accuracy: 0.9873
Confusion matrix: TN=57897, FP=1, FN=992, TP=19324

TABLE VII
DETAILED RESULTS FOR THE CHAR-CNN ON THE TEST SET.

Class / Avg	Precision	Recall	F1	Support
0 (benign)	0.9975	0.9985	0.9980	57898
1 (phishing)	0.9959	0.9930	0.9944	20316
Macro avg	0.9967	0.9958	0.9962	78214
Weighted avg	0.9971	0.9971	0.9971	78214

Accuracy: 0.9971
Confusion matrix: TN=57814, FP=84, FN=143, TP=20173

TABLE VIII
DETAILED RESULTS FOR THE HYBRID ENSEMBLE ON THE TEST SET.

Class / Avg	Precision	Recall	F1	Support
0 (benign)	0.9947	0.9998	0.9973	57898
1 (phishing)	0.9994	0.9847	0.9920	20316
Macro avg	0.9970	0.9922	0.9947	78214
Weighted avg	0.9959	0.9959	0.9959	78214

Accuracy: 0.9959
Confusion matrix: TN=57885, FP=13, FN=311, TP=20005

D. Model Comparison and Discussion

Table II–Table VI summarize the performance of the classical lexical models and their calibrated variants, while Table VII and Table VIII report the deep-learning and hybrid experiments. For comparability across models, all confusion matrices are reported using a single probability threshold of 0.5 and the same hostname-grouped test split. Below we present the summary comparison table IX.

Among the **lexical-only** baselines, **XGBoost** achieves the strongest overall balance, obtaining an accuracy of 0.9954 with relatively low error counts (FP=109 and FN=251). **Random Forest** also performs strongly (accuracy 0.9945), with slightly fewer false positives (FP=84) but more false negatives (FN=344), indicating a more conservative tendency to avoid flagging benign URLs at the default threshold. **Logistic Regression** provides a competitive linear reference point (accuracy 0.9917), but yields the highest false-positive count among the three (FP=332), suggesting that a purely linear boundary is less effective at capturing the non-linear interactions present in URL lexical patterns.

Introducing **probability calibration** substantially changes the operating behavior of the tree-based models. **RF_calibrated** reduces false positives (FP=39 versus 84 in RF) but increases false negatives (FN=475 versus 344), reflecting a shift in the score distribution that favors fewer alarms at the cost of missed phishing URLs. The same effect is more pronounced for **XGB_calibrated**, which achieves an extremely low FP count (FP=1) but at the cost of a larger FN count (FN=992). This is consistent with calibration producing probabilities that are more conservative around the decision boundary when a fixed 0.5 threshold is applied, effectively reducing alarms while increasing misses. In other words, calibration does not necessarily improve classification metrics at a fixed threshold; rather, it aims to make the probability scores more meaningful for threshold-based decision policies.

For the **Char-CNN** baseline, results are strong at the 0.5 threshold (accuracy 0.9971, FP=84, FN=143), indicating that character-level representations can capture obfuscation patterns and lexical artifacts that are difficult to encode explicitly. The **hybrid ensemble** (XGB Calibrated + Char-CNN) improves on some aspects of the lexical-only models by substantially reducing false positives compared to uncalibrated XGB (FP=13 versus 109) while maintaining a relatively low FN count (FN=311), which suggests complementary strengths between the handcrafted feature model and the Char-CNN representation. However, these gains must be interpreted in the context of deployment constraints: deep-learning components increase dependency footprint and operational complexity compared to a purely classical approach.

Overall, the comparison highlights a central trade-off: models that minimize FP at a fixed threshold (especially calibrated variants) may do so by accepting higher FN, while more sensitive models reduce FN but can produce more false alarms. Because the target application is a browser-facing warning system, the acceptable operating point is not determined solely by accuracy or F1-score, but by the balance between user disruption (FP) and missed phishing events (FN), which is discussed next.

E. Security Analysis

From a security perspective, false positives (FP) and false negatives (FN) carry different operational costs in a browser-integrated detector. A **false negative** corresponds to a phishing URL that is classified as benign, potentially exposing the user to credential theft or follow-on compromise. A **false positive** corresponds to a benign URL that is flagged as phishing, which can interrupt normal browsing and, more importantly, degrade user trust in the tool.

In user-facing security systems, repeated false alarms often lead to *alert fatigue*: users become desensitized to warnings and may ignore or disable protective mechanisms. Therefore, in this project we prioritize a low false-positive rate for high-severity actions (such as blocking).

In this report, confusion matrices are presented at a threshold of 0.5 to enable fair, standardized comparison across all evaluated models, unlike the deployed system which uses stricter thresholds as explained above in Subsection IV-C

This analysis emphasizes that the "best" model depends on the intended operating point. Models such as `XGB_calibrated` can be desirable in this setting because they yield very few false positives at the default threshold (FP=1), supporting conservative, high-confidence blocking decisions. Conversely, models with higher recall at 0.5 reduce missed phishing events but may generate more user-visible interruptions. The final deployment decision therefore reflects a usability–security trade-off rather than maximizing a single aggregate metric.

F. Final Model Selection and Operating Point

Considering the requirements of our project guidelines (lightweight, real-time analysis with lexical-only input) and the user-facing nature of a browser extension, we select `XGB_calibrated` as the final deployed classifier. Although the Char-CNN and the hybrid ensemble improve certain metrics in our experiments, they introduce additional operational complexity (deep-learning runtime dependencies and the maintenance of multiple inference pipelines), which is misaligned with the goal of a lightweight, easily deployable system.

This choice is also driven by the class imbalance observed in real browsing. Benign URLs are far more frequent than phishing URLs; therefore, even a small false positive rate can translate into a large number of incorrect warnings or blocks. In a browser extension context, false positives are particularly damaging because they interrupt normal navigation and quickly erode user trust, often leading users to disable the tool or overuse allowlisting. For this reason, our operating point prioritizes minimizing false positives, even if this implies accepting a higher number of false negatives.

The calibrated XGBoost model provides an effective compromise: it remains efficient on CPU and produces calibrated probabilities that can be used reliably by a decision policy. In

TABLE IX
MODEL COMPARISON ON THE HOSTNAME-GROUPED TEST SET (THRESHOLD = 0.5).

Model	Acc.	Prec. (1)	Rec. (1)	F1 (1)	FP	FN
LogReg	0.9917	0.9837	0.9843	0.9840	332	319
RF	0.9945	0.9958	0.9831	0.9894	84	344
RF_calibrated	0.9934	0.9980	0.9766	0.9872	39	475
XGB	0.9954	0.9946	0.9876	0.9911	109	**251**
XGB_calibrated	0.9873	0.9999	0.9512	0.9750	**1**	992
Char-CNN	0.9971	0.9959	0.9930	0.9944	84	143
Hybrid (0.4L + 0.6C)	0.9959	0.9994	0.9847	0.9920	13	311

particular, it achieves an extremely low false-positive count in our evaluation (FP=1), which supports conservative blocking behavior. In deployment, we further reduce disruption by applying a three-level policy with thresholds 0.60 and 0.85, reserving hard blocking for high-confidence cases while using softer warnings for borderline scores.

VI. CONCLUSION

This work presented a lightweight phishing URL detection system that relies exclusively on lexical properties of the URL string, enabling fast and privacy-preserving analysis without downloading web content. We constructed a robust preprocessing and feature engineering pipeline, and adopted a hostname-grouped train/test split to reduce leakage and evaluate generalization to unseen hostnames. Across multiple classical baselines (Logistic Regression, Random Forest, and XGBoost), as well as calibrated variants, a Char-CNN baseline, and a hybrid ensemble, the experiments highlighted the central operational trade-off between missed phishing events (FN) and user disruption through false alarms (FP).

Given the requirements for this project, we selected `XGB_calibrated` as the final deployed model because it combines CPU-efficient inference, a small dependency footprint, and the lowest false positives rate. While deep-learning and hybrid approaches achieved strong classification performance, their runtime and operational complexity make them less aligned with the lightweight requirement. The final prototype integrates the model into a local Flask API and a Chrome Extension that provides real-time risk feedback through a three-level policy, reserving stronger actions for high-confidence detections.

A. Future Work

With an additional six months, we would extend this work in several directions focused on robustness, realism, and operational validation. To start, we would validate generalization on fresher and more diverse datasets (including time-split evaluation) to quantify degradation under concept drift and newly emerging phishing campaigns. Secondly, we would test robustness against adversarial manipulations such as homoglyph attacks, subdomain padding, and token obfuscation, and evaluate defenses such as stricter normalization, punycode-aware features, and adversarial training or hard-negative mining.

From an engineering perspective, we would optimize the end-to-end latency and caching strategy in the extension, add richer telemetry for offline error analysis (while preserving privacy), and conduct a small user study to measure warning fatigue and determine whether the three-tier UI policy improves user compliance compared to binary blocking. Finally, we would explore incremental or online updating strategies to keep the model effective as attacker tactics evolve, while maintaining the lightweight constraints of the system.

REFERENCES

[1] Anti-Phishing Working Group (APWG), Phishing Activity Trends Report, 2nd Quarter 2025, APWG, Aug. 28, 2025, pp. 3–4.

[2] A. Blum, B. Wardman, T. Solorio, and G. Warner, "Lexical feature based phishing URL detection using online learning," in *Proc. 3rd ACM Workshop on Artificial Intelligence and Security (AISec)*, 2010, pp. 54–60, doi: 10.1145/1866423.1866434. [Online]. Available: https://cir.nii.ac.jp/crid/1360579817075651200

[3] 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, doi: 10.48550/arXiv.1910.06277. [Online]. Available: https://arxiv.org/abs/1910.06277

[4] F. Tajaddodianfar, J. W. Stokes, and A. Gururajan, "Texception: A character/word-level deep learning model for phishing URL detection," in *Proc. IEEE Int. Conf. Acoustics, Speech and Signal Processing (ICASSP)*, 2020, doi: 10.1109/ICASSP40776.2020.9053670.

[5] M. A. Remmide, F. Boumahdi, N. Boustia, C. L. Feknous, and R. Della, "Detection of phishing URLs using temporal convolutional network," *Procedia Computer Science*, 2022, pp. 74–82, doi: 10.1016/j.procs.2022.10.209.

[6] R. Dubey, A. M. Tripathi, A. Srivastava, and S. Singh, "Phishing detection system: An ensemble approach using character-level CNN and feature engineering," *arXiv preprint arXiv:2512.16717*, 2025, doi: 10.48550/arXiv.2512.16717. [Online]. Available: https://arxiv.org/abs/2512.16717

[7] A. Niculescu-Mizil and R. Caruana, "Predicting good probabilities with supervised learning," in *Proc. 22nd Int. Conf. Machine Learning (ICML)*, 2005, doi: 10.1145/1102351.1102430.

[8] A. Yasin, R. Fatima, J. A. Khan, and W. Afzal, "Behind the bait: Delving into PhishTank's hidden data," *Data in Brief*, vol. 52, Art. no. 109959, 2023, doi: 10.1016/j.dib.2023.109959.

[9] T. Dalton, H. Gowda, G. Rao, S. Pargi, A. H. Khodabakhshi, J. Rombs, S. Jou, and M. Marwah, "PhreshPhish: A real-world, high-quality, large-scale phishing website dataset and benchmark," *arXiv preprint arXiv:2507.10854*, 2025, doi: 10.48550/arXiv.2507.10854. [Online]. Available: https://arxiv.org/abs/2507.10854

[10] S. Kumar, "Malicious And Benign URLs," Kaggle, May 31, 2019. [Online]. Available: https://www.kaggle.com/datasets/siddharthkumar25/malicious-and-benign-urls . [Accessed: Jan. 2, 2026]

[11] T. Tiwari, "Phishing Site URLs," Kaggle (dataset). [Online]. Available: https://www.kaggle.com/datasets/taruntiwarihp/phishing-site-urls . [Accessed: Dec. 26, 2026]

[12] M. Siddhartha, “Malicious URLs dataset,” Kaggle (dataset). [Online]. Available: https://www.kaggle.com/datasets/sid321axn/malicious-urls-dataset . [Accessed: Dec. 28, 2026]

APPENDIX A

The images referenced throughout the report can be accessed in the following pages.

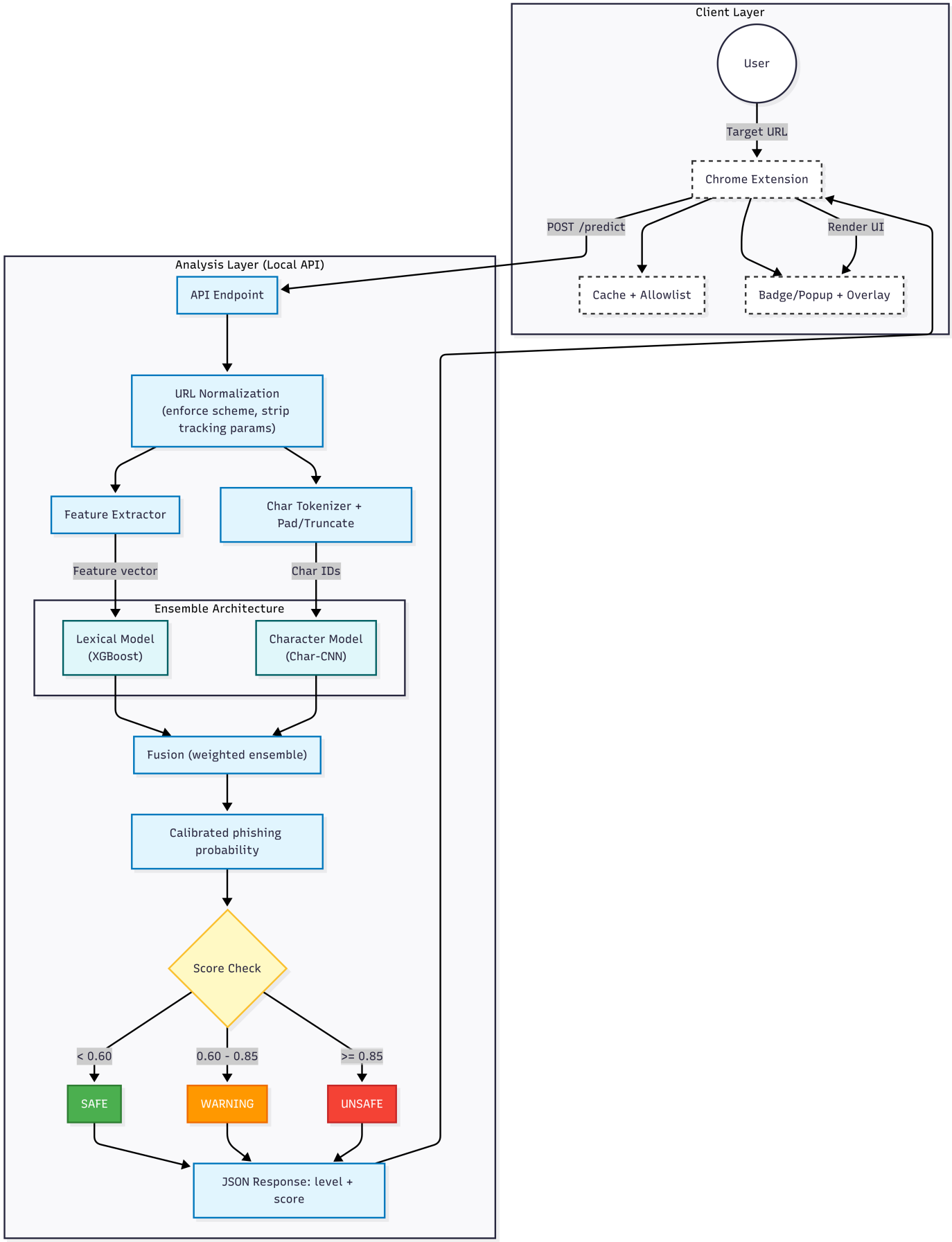

Fig. 1. End-to-end architecture of the phishing URL detection system (extension + local API).

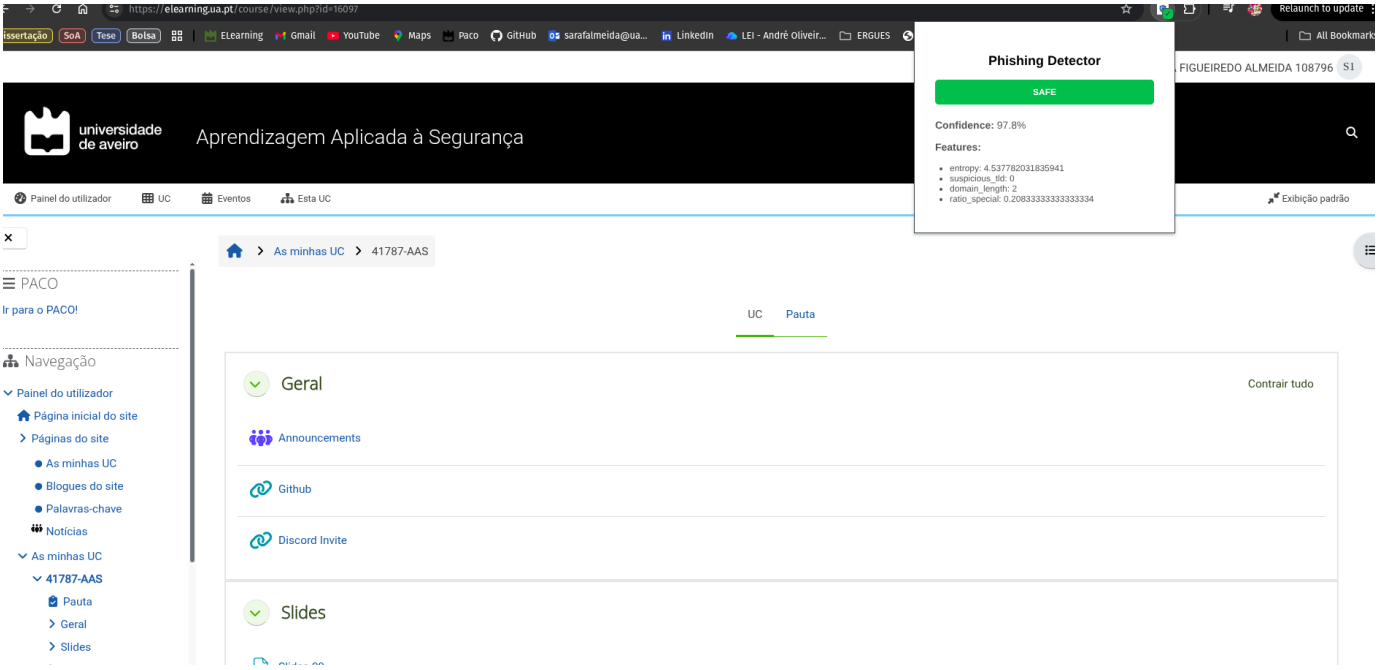

Fig. 2. Benign URL detected by the browser extension through the API

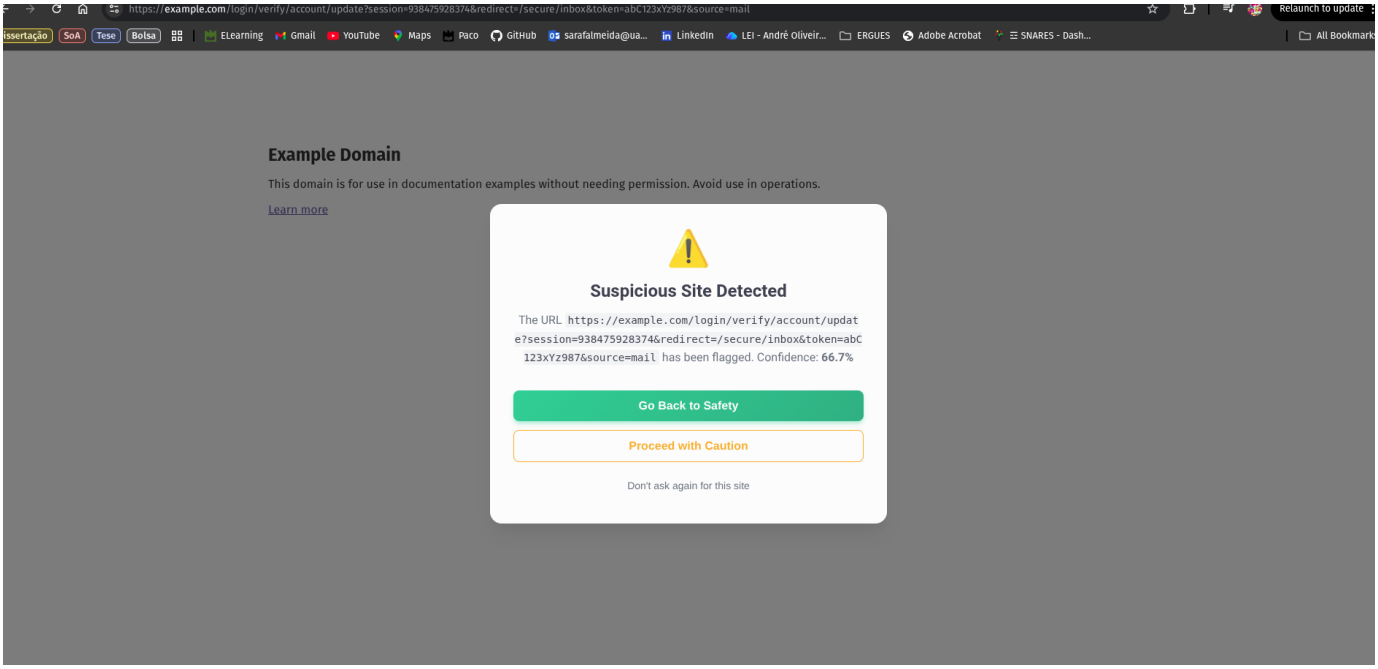

Fig. 3. Suspicious URL detected by the browser extension through the API

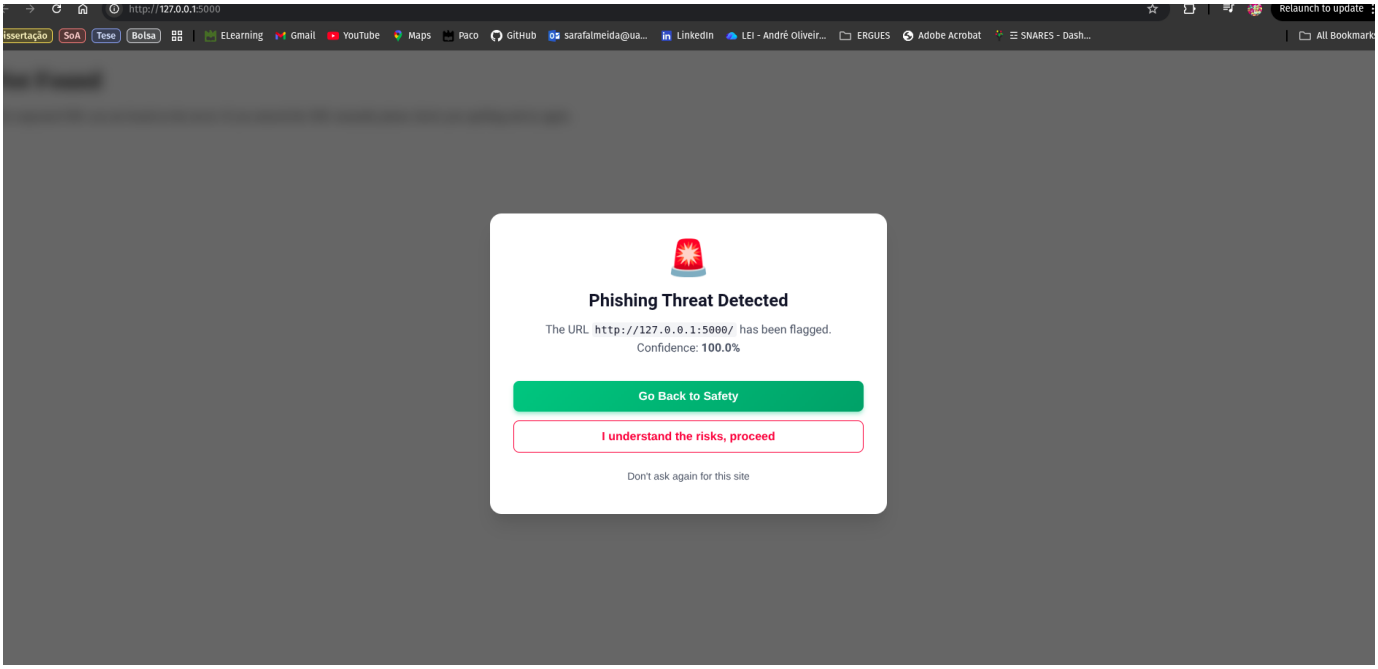

Fig. 4. Phishing URL detected by the browser extension through the API