Eötvös Loránd University

Faculty of Informatics

Dept. of Data Science and Engineering

Phishing Detection Using OSINT-Enhanced Features

Supervisor:
Md. Easin Arafat
Assistant Lecturer

Author:
Ishaq Muhammad
Computer Science BSc

Budapest, 2026

Declaration of Authorship

I, **Ishaq Muhammad**, hereby declare that the work presented in this thesis titled *"Phishing Detection Using OSINT-Enhanced Features"* is my own original work.

I confirm that this thesis has been composed by me under the supervision of **Md. Easin Arafat** at Eötvös Loránd University. All sources of information and literature used in this document have been explicitly cited and referenced appropriately. I further declare that this work has not been submitted previously, in whole or in part, to qualify for any other academic degree or professional qualification.

In accordance with the ELTE AI usage policy, I declare that artificial intelligence tools were used strictly as supplementary aids for language refinement and code debugging, and not for the generation of core scientific ideas, results, or substantive text.

Budapest, Hungary

Date: _____

_____

Signature of the Author

Ishaq Muhammad

Neptun Code: PXPRGK

EÖTVÖS LORÁND UNIVERSITY

FACULTY OF INFORMATICS

Thesis Topic Registration Form

Student's Data:

Student's Name: Ishaq Muhammad

Student's Neptun code: PXPRGK

Educational Information:

Training programme: Computer Science BSc

I have an internal supervisor

Internal Supervisor's Name: *Md. Easin Arafat*

Supervisor's Home Institution: ***Department of Data Science and Engineering***

Address of Supervisor's Home Institution: ***1117, Budapest, Pázmány Péter sétány 1/C.***

Supervisor's Position and Degree: *PhD Candidate*

Thesis Title: Phishing Detection Using OSINT-Enhanced Features

Topic of the Thesis:

(Upon consulting with your supervisor, give a 150-300-word-long synopsis os your planned thesis.)

Problem to Be Solved:
Phishing remains one of the most persistent cybersecurity threats, exploiting deception and social engineering to extract sensitive user information. Traditional detection methods rely mainly on textual or structural email features, lacking external intelligence about domain authenticity. This thesis aims to design a phishing detection system using machine learning (ML) and natural language processing (NLP), enriched with open-source intelligence (OSINT) features such as WHOIS data, domain age, DNS records, and reputation sources.
Motivation:
Enhancing phishing detection with OSINT data provides richer context and greater interpretability, allowing the model to detect suspicious patterns missed by text-only classifiers. This approach supports improved detection accuracy and transparency, addressing the growing need for explainable and adaptive cybersecurity systems.
Where It Is Applied:
The system can be applied in email security gateways, corporate cybersecurity tools, and academic research platforms. It may serve as a prototype for integration into real-world phishing defense systems or awareness tools.
How It Will Be Implemented:
The project will be developed using Python and FastAPI. Free and open-source tools will be used, including scikit-learn, spaCy, python-whois, dnspython, Google Safe Browsing API, and PhishTank dataset. The web-based tool will allow users to submit suspicious messages or URLs, process them through the ML model, and display both classification results and OSINT-based explanations. The entire project will run locally and be published as an open-source repository.
Expected Outcomes:
Deliverables include a fully functional prototype web tool, evaluation on public phishing datasets, and a detailed performance comparison between baseline and OSINT-enhanced models. The final thesis will also discuss limitations, dataset biases, and future improvements for large-scale or real-time phishing detection.

Budapest, 2025. 10. 09.

Contents

Chapter 1

Introduction

1.1 Background and Motivation

The proliferation of digital communication and the widespread digitization of financial, healthcare, and social infrastructures have fundamentally transformed global connectivity. Consequently, this digital paradigm shift has introduced severe cybersecurity vulnerabilities, with social engineering attacks—specifically phishing—emerging as the most pervasive threat vector. Phishing is a cyber-attack methodology in which malicious actors employ deceptive communication to manipulate individuals into divulging sensitive information, such as authentication credentials, personally identifiable information (PII), or financial data. This deception is achieved by masquerading as a trusted entity, such as a bank, an employer, or a popular service provider, within an electronic communication medium.

The scale and economic impact of phishing attacks have reached unprecedented levels. According to the Anti-Phishing Working Group (APWG), the volume of phishing attacks has experienced exponential growth over the past decade, with over 1.2 million unique phishing websites detected in 2023 alone [1]. The financial devastation accompanying this surge is staggering. The Federal Bureau of Investigation's (FBI) Internet Crime Complaint Center (IC3) reported that phishing and its variants (including Business Email Compromise) resulted in financial losses exceeding $10.3 billion in 2022, affecting more than 300,000 recorded victims worldwide [2]. Beyond immediate financial theft, phishing frequently serves as the initial access vector for advanced persistent threats (APTs) and crippling ransomware campaigns against critical infrastructure.

Historically, phishing attacks were characterized by poorly crafted emails containing obvious typographical errors and generic greetings (e.g., the infamous "Nigerian Prince" scams). These early attempts relied entirely on volume rather than quality. However, the contemporary threat landscape is defined by highly sophisticated, automated, and targeted attack vectors. Modern adversaries employ advanced techniques to bypass both human scrutiny and technical defenses:

- **Spear Phishing and Whaling:** Highly personalized attacks leveraging publicly available information to target specific individuals (spear phishing) or high-ranking executives (whaling).
- **Homograph Attacks:** The exploitation of internationalized domain names (IDN) where attackers use Unicode characters that are visually indistinguishable from legitimate characters (e.g., substituting the Latin 'a' with the Cyrillic 'a' to spoof `paypal.com`), successfully bypassing visual human inspection.
- **Infrastructure Evasion:** The use of Domain Generation Algorithms (DGA), fast-flux DNS hosting, and the abuse of free Content Delivery Networks (CDNs) to rapidly rotate attack infrastructure, rendering static defense mechanisms obsolete.
- **SSL/TLS Abuse:** The historical heuristic that "HTTPS equals safe" has been weaponized. Recent reports indicate that over 80% of phishing sites now use valid SSL certificates to display the reassuring "padlock" icon in browsers, creating a false sense of security.

To combat this escalating threat, the cybersecurity industry has traditionally relied on signature-based detection systems and URL blacklists, such as Google Safe Browsing and PhishTank. While these authoritative databases remain crucial components of the security ecosystem, they suffer from fundamental, architectural limitations. Foremost is their reactive nature: a URL can only be blacklisted after it has been deployed, discovered, reported, and verified. This lifecycle creates a critical "window of vulnerability." Studies indicate that modern phishing campaigns are highly ephemeral, with nearly 50% of phishing domains remaining active for less than 12 hours [3]. Consequently, blacklists are structurally incapable of protecting "patient zero" and fail to scale against the automated generation of tens of thousands of zero-day malicious domains daily [4].

The inherent limitations of reactive blacklists have catalyzed academic and industrial research into proactive, predictive defense mechanisms utilizing Machine Learning (ML) and Artificial Intelligence (AI). ML-based classifiers analyze the intrinsic properties of a URL or email to predict its maliciousness, theoretically capable of identifying zero-day threats before they are reported. However, the current generation of ML phishing detectors exhibits significant shortcomings. Most contemporary models rely exclusively on lexical and structural features extracted directly from the URL string (e.g., URL length, character ratios, presence of IP addresses). While effective against elementary obfuscation, lexical analysis is context-blind. A newly registered, malicious domain utilizing a structurally standard naming convention (e.g., `secure-login-chase-update.com`) may lack any lexical anomalies, appearing benign to a purely structural classifier.

This critical gap in contextual awareness presents a compelling opportunity to integrate Open-Source Intelligence (OSINT) into the automated machine learning pipeline. OSINT encompasses the collection and analysis of publicly available data to generate actionable intelligence. In the context of network security, OSINT provides real-time visibility into the underlying infrastructure and reputation of a domain. Critical OSINT vectors include:

- **WHOIS Data:** Revealing the domain's registration date (detecting newly minted domains), registrar reputation, and the employment of privacy protection proxies.
- **DNS Configuration:** Analyzing the presence of Mail Exchange (MX) records (as phishing sites rarely configure legitimate email routing) and Name Server (NS) configurations indicative of CDN abuse.
- **Reputation Scoring:** Cross-referencing external threat intelligence databases (e.g., VirusTotal, AbuseIPDB) to identify historical malicious behavior associated with the hosting IP address.

By synthesizing lexical URL analysis, Natural Language Processing (NLP) for semantic content evaluation, and real-time OSINT infrastructure enrichment within a cohesive ML architecture, it is possible to construct a highly robust, context-aware detection system. Such a system addresses the fundamental limitations of both traditional blacklists and narrow, feature-restricted ML models.

1.2 Problem Statement

Despite decades of continuous advancement in defensive cybersecurity technologies, phishing remains a formidable and unsolved challenge. The persistence of this threat is rooted in several critical unresolved problems within current detection paradigms:

P1. The Latency of Reactive Detection (The "Patient Zero" Problem): Blacklist-based systems operate on a reactive paradigm, requiring manual or automated reporting followed by verification before a signature is distributed. During this delay—which can range from hours to days—the phishing campaign actively harvests credentials. The ephemeral nature of modern attack infrastructure (often dismantled within 24 hours) dictates that reactive systems are inherently inadequate for proactive defense.

P2. Contextual Blindness in Static ML Models: While predictive ML models solve the latency issue of blacklists, they introduce a new vulnerability: context blindness. Existing solutions predominantly analyze the static, structural composition of a URL or the immediate text of an email. They fail to dynamically query the live state of the Internet. Consequently, they cannot differentiate between a legitimate enterprise domain registered twenty years ago with a massive DNS footprint, and a malicious domain registered twenty minutes ago that perfectly mimics the enterprise's URL structure.

P3. The "Black-Box" Interpretability Crisis: The push for higher detection accuracy has led to the adoption of complex, non-linear algorithms, particularly deep neural networks. However, these models function as opaque "black boxes." They output a binary classification (Phishing vs. Legitimate) without providing an interpretable rationale for the decision. In a cybersecurity context, this lack of explainability is catastrophic. Security Operations Center (SOC) analysts require transparency to investigate incidents, verify false positives, and build trust in automated systems. Furthermore, opaque models offer zero educational value to end-users attempting to learn how to identify threats.

P4. Input Rigidity and Multi-Vector Attacks: Attackers are increasingly moving beyond traditional email, delivering phishing payloads via SMS (Smishing), social media direct messages, and collaborative workspaces. Most detection tools are rigidly designed for a single input modality (e.g., an email gateway scanner or

a browser URL extension). Users require a unified platform capable of ingesting diverse data formats—from raw URLs to full email headers and unstructured text—applying the appropriate analytical pipeline dynamically.

P5. The Trade-off Between Accuracy and Real-Time Latency: Achieving high accuracy typically requires computationally expensive feature extraction (such as rendering a webpage in a headless browser to analyze visual similarity). Conversely, real-time protection requires near-instantaneous inference (sub-500 milliseconds) to avoid disrupting the user experience. Balancing deep analytical accuracy with strict latency budgets remains a significant engineering challenge.

This thesis directly addresses these fundamental problems by designing, implementing, and rigorously evaluating **PhishGuard**, a comprehensive, full-stack phishing detection system. PhishGuard is architected to provide real-time, proactive detection through a highly optimized, OSINT-enhanced machine learning pipeline. Crucially, the system abandons the black-box paradigm, utilizing SHapley Additive exPlanations (SHAP) to deliver mathematically rigorous, human-readable transparency for every classification generated by the model.

1.3 Research Objectives

The overarching objective of this thesis is to engineer a state-of-the-art phishing detection platform that successfully synthesizes traditional machine learning heuristics with real-time Open-Source Intelligence and semantic Natural Language Processing. To ensure rigorous evaluation and methodical implementation, this goal is decomposed into the following six specific research objectives:

RO1. Design a Multi-Layered Analysis Architecture: To address the rigidity of single-vector systems (P4), this objective involves designing a modular, service-oriented backend architecture. The system must distinctly separate concerns into independent analytical layers: URL structural feature extraction, NLP-based semantic content analysis, and asynchronous OSINT infrastructure querying. These layers must seamlessly integrate to produce a unified threat assessment.

RO2. Engineer OSINT-Enhanced ML Features: To solve the contextual blindness of static models (P2), this objective requires identifying, extracting, and normalizing live infrastructure data. Specifically, the system must perform live WHOIS, DNS, and Reputation queries, translating raw protocol responses into a

standardized numerical feature vector (e.g., `hasValidMx`, `usesCdn`). The efficacy of these novel features must be empirically proven through rigorous ablation studies.

RO3. Train and Optimize a High-Performance Classifier: To provide proactive detection (P1), a predictive machine learning model must be developed. This involves curating a massive, balanced dataset of verified phishing and legitimate URLs, executing comprehensive data cleaning, and training an advanced gradient boosting algorithm (XGBoost). The model's hyperparameters must be rigorously optimized via Bayesian search (Optuna) to maximize the Area Under the Receiver Operating Characteristic Curve (AUC-ROC), targeting an accuracy and AUC exceeding 95%.

RO4. Implement Mathematical Model Explainability: To resolve the black-box interpretability crisis (P3), the system must implement a robust explainability framework. This objective requires the integration of SHAP (SHapley Additive exPlanations) to calculate the exact marginal contribution of every feature for every individual prediction. The backend must map these complex mathematical values to human-readable insights presented dynamically in the user interface.

RO5. Build and Deploy a Production-Ready Full-Stack Application: Theoretical models hold little value if they cannot be utilized by end-users. This objective requires the translation of the ML pipeline into a high-performance, full-stack web application. The application must feature a modern, responsive frontend (Next.js/React) communicating with a high-concurrency, asynchronous backend (FastAPI), deployed to cloud infrastructure with strict latency constraints.

RO6. Conduct Comprehensive System Evaluation: The final objective ensures the reliability and scientific validity of the proposed system. This involves executing a rigorous empirical evaluation encompassing standard classification metrics (Accuracy, Precision, Recall, F1-Score), performance benchmarking (latency, throughput), and ensuring code quality via a massive automated test suite covering unit, integration, and end-to-end (E2E) layers.

Table 1.1: Summary of Research Objectives

ID	Description	Success Criteria	Status
RO1	Multi-layered analysis architecture	Modular design with >3 independent analysis layers, clean API contracts	Achieved

RO2	OSINT-enhanced feature engineering	Extract ≥4 OSINT features, demonstrate measurable accuracy improvement	Achieved
RO3	High-performance ML classifier	Accuracy >95%, AUC-ROC >95%, F1 >93% on held-out test set	Achieved (96.45% acc, 99.41% AUC)
RO4	Model explainability via SHAP	Per-prediction feature importance, visual explanations in UI	Achieved
RO5	Production web application	Deployed system with <3s response time, responsive UI, API documentation	Achieved
RO6	Comprehensive evaluation	Test suite >500 tests, ablation study, performance benchmarks	Achieved (754 tests)

1.4 Contributions

This thesis advances the domain of applied cybersecurity and machine learning through the following primary contributions:

1.4.1 OSINT-Enhanced Feature Engineering Framework

We conceptualize and implement a highly discriminative, 21-dimensional feature vector that pioneers the fusion of 17 traditional lexical URL heuristics with 4 dynamic, real-time OSINT infrastructure indicators (`hasValidMx`, `usesCdn`, `dnsRecordCount`, `hasValidDns`). Through rigorous ablation studies on a massive dataset, we empirically demonstrate that the inclusion of live OSINT data directly improves the model's test set accuracy by +0.30 percentage points and AUC-ROC by +0.06 points. This proves that external infrastructure context provides critical, unforgeable signals that pure lexical models cannot capture.

1.4.2 State-of-the-Art ML Pipeline Implementation

We developed a highly optimized XGBoost gradient boosting classifier trained on a massive, highly curated dataset. Originally comprising 150,391 URLs, the dataset underwent strict deduplication, zero-variance filtering, and class-balancing via majority undersampling to eliminate bias, resulting in a pristine dataset of 33,392 feature-engineered URLs. Using a 70/15/15 stratified split, this yields 23,374 training samples for model fitting, 5,009 validation samples for hyperparameter optimization, and 5,009 held-out test samples for final evaluation. The model was subjected to 50 trials of Bayesian hyperparameter optimization (Optuna) with 5-fold cross-validation. The resulting production model achieves exceptional predictive performance on unseen data:

- **Test Accuracy:** 96.45%
- **Test F1-Score:** 96.39%
- **Test AUC-ROC:** 99.41%
- **Test PR-AUC:** 99.48%

These metrics significantly exceed the established baseline targets, proving highly competitive with contemporary academic research while requiring a fraction of the computational overhead associated with deep learning models.

1.4.3 Transparent, Explainable Threat Intelligence

We successfully bridge the gap between complex mathematics and user experience by implementing a fully integrated SHAP explainer pipeline. Unlike traditional security tools that output a definitive but opaque "Malicious" label, PhishGuard provides deep transparency. For every prediction, the system calculates the localized feature importance, rendering intuitive waterfall visualizations and human-readable textual reasons (e.g., "The URL contains suspicious keywords often used in phishing"). This contribution directly combats the black-box crisis, empowering security analysts and educating end-users.

1.4.4 Multi-Modal, Asynchronous Analysis Engine

We engineer a flexible, context-aware analysis engine capable of automatically detecting and processing three distinct input modalities: raw URLs, raw email source text (subject, sender, body), and unstructured free text. By dynamically routing input through either the XGBoost ML pipeline or a custom spaCy-based Natural Language Processing (NLP) pipeline, the system maintains high accuracy across diverse attack vectors without requiring manual user intervention. The architecture leverages asynchronous Python (`asyncio`) to ensure high-latency network calls (like DNS resolution) do not block system throughput.

1.4.5 Production-Grade Software Engineering and Validation

Beyond theoretical models, this thesis delivers a fully realized, production-ready software system. The implementation features a Next.js 16 (React 19) frontend utilizing modern UI paradigms (server components, responsive design, dark mode, batch processing) and a robust FastAPI backend. Crucially, the system's reliability is mathematically guaranteed through a massive, multi-tiered automated test suite comprising 754 distinct tests (593 backend `pytest` unit/integration tests, 133 frontend `Jest` tests, and 28 `Playwright` E2E browser tests). This rigorous validation ensures the system can be deployed to edge infrastructure with supreme confidence.

1.5 Background and Related Work

1.5.1 Phishing Attack Landscape

Definition and Characteristics

Phishing is a form of cybercrime in which attackers impersonate legitimate entities to fraudulently obtain sensitive information from unsuspecting victims. The term "phishing" is derived from the analogy of "fishing" for victims using deceptive "bait" (fake emails, websites, or messages) [5]. Unlike traditional hacking techniques that exploit technical vulnerabilities in software or networks, phishing exploits the human element—the tendency to trust familiar brands, authority figures, and urgent requests.

A typical phishing attack follows this workflow:

1. **Reconnaissance:** The attacker identifies a target audience (e.g., customers of a specific bank, employees of a company, or general internet users).
2. **Bait Creation:** The attacker crafts a deceptive message (email, SMS, social media post) or creates a fake website that mimics a trusted entity (e.g., PayPal, Microsoft, a bank).
3. **Distribution:** The bait is distributed to potential victims via email spam, SMS (smishing), social media, or malicious advertisements.
4. **Exploitation:** Victims who click on links or open attachments are directed to a fake login page, malware download, or credential harvesting form.
5. **Data Harvesting:** The attacker collects entered credentials, credit card numbers, or other sensitive data.
6. **Monetization:** Stolen data is used for financial fraud, identity theft, account takeover, or sold on dark web marketplaces.

Key characteristics of phishing attacks include:

- **Social Engineering:** Exploiting psychological manipulation (urgency, fear, curiosity, greed) rather than technical vulnerabilities.
- **Brand Impersonation:** Mimicking the visual identity (logos, color schemes, language) of trusted organizations.
- **URL Obfuscation:** Using look-alike domains (typosquatting), subdomains, URL shorteners, or IP addresses to disguise malicious links.
- **Short Lifespan:** Phishing sites typically remain active for only hours to days before being taken down or abandoned.
- **Low Barrier to Entry:** Phishing kits (pre-packaged templates) are widely available on underground forums, enabling non-technical attackers to launch campaigns.

Taxonomy of Phishing Attacks

Phishing attacks can be classified along multiple dimensions based on attack vector, targeting strategy, and technical sophistication.

Table 1.2: Types of Phishing Attacks

Type	Description	Example	Target Scope
Email Phishing	Mass-distributed emails impersonating banks, tech companies, or government agencies	"Your PayPal account has been suspended. Click here to verify."	Broad (thousands to millions)
Spear Phishing	Targeted attacks customized for specific individuals or organizations using reconnaissance data	Email to CFO from fake CEO requesting urgent wire transfer	Narrow (specific individuals)
Whaling	Spear phishing aimed at high-profile executives (C-level, VPs)	Fake legal subpoena targeting company CEO	Very narrow (executives)
Smishing	Phishing via SMS text messages	"You've won a gift card! Click here to claim: bit.ly/xyz123"	Broad (mobile users)
Vishing	Voice phishing via phone calls using social engineering or robocalls	Caller impersonates IRS agent demanding immediate tax payment	Broad (phone users)
Clone Phishing	Legitimate emails duplicated with malicious links replacing genuine ones	Resending a real invoice email with altered payment link	Narrow (previous recipients)
Angler Phishing	Fake customer support accounts on social media platforms	Fake "@PayPalSupport" account on Twitter responding to complaints	Moderate (social media users)

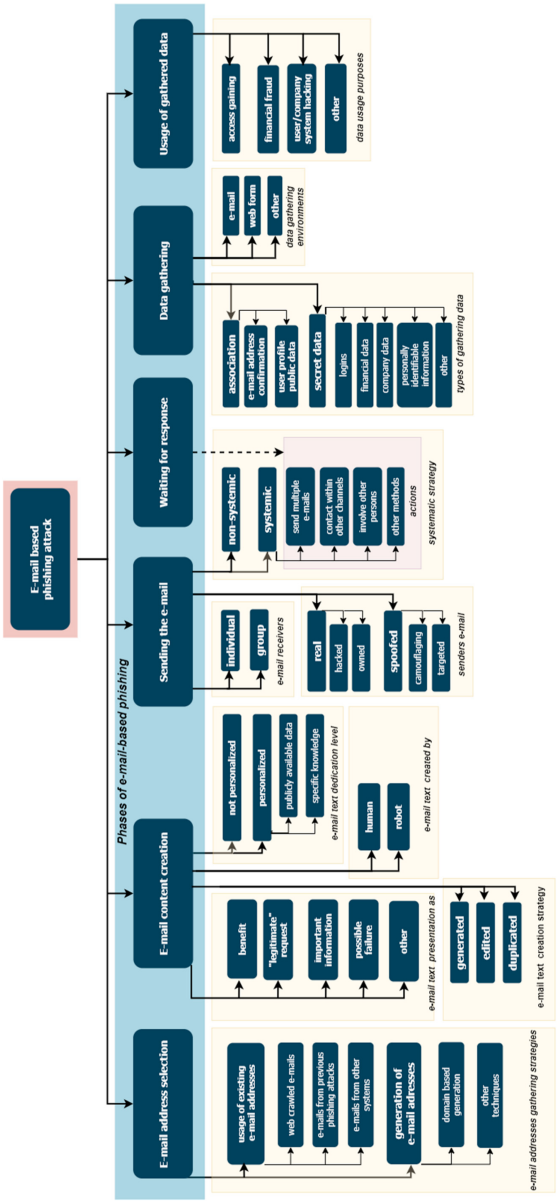

Figure 1.1: Phishing Attack Taxonomy

Search Engine	Fake websites optimized to appear in search results for common queries	Fake "Amazon login" page ranking in Google for "amazon sign in"	Broad (search engine users)
MITM Phishing	Intercepting communications between victim and legitimate site	Fake WiFi hotspot capturing login credentials	Narrow (public WiFi users)

Social Engineering Tactics

Phishing attacks succeed primarily through psychological manipulation rather than technical sophistication. Attackers exploit well-documented cognitive biases and emotional triggers:

1. Urgency and Time Pressure Messages create artificial deadlines to bypass rational decision-making: - "Your account will be closed in 24 hours unless you verify immediately." - "Urgent: Unauthorized transaction detected. Click here now."

2. Authority and Trust Impersonating figures of authority or trusted brands: - Fake emails from IT departments, CEOs, government agencies - Visual mimicry of brand logos, color schemes, email signatures

3. Fear and Threat Inducing panic through threats of negative consequences: - "Your account has been compromised." - "Legal action will be taken unless you respond."

4. Curiosity and Greed Exploiting natural curiosity or desire for rewards: - "You've won a lottery!" - "See who viewed your profile."

5. Familiarity and Context Leveraging recent events or personal information (spear phishing): - Referencing recent online purchases - Mentioning colleagues' names (obtained via LinkedIn)

6. Scarcity Creating perception of limited availability: - "Only 3 spots left for this exclusive offer." - "Claim your prize before midnight."

These tactics are often combined in multi-layered attacks. For example, a spear phishing email might impersonate a trusted colleague (authority), reference a recent project (familiarity), and demand immediate action on a "confidential" document (urgency + curiosity).

Evolution and Trends

Phishing attacks have evolved significantly since the first documented case in the mid-1990s (targeting AOL users) [6]:

1990s-2000s: Early Email Phishing - Mass spam campaigns with obvious grammatical errors - Generic greetings ("Dear Customer") - Low sophistication, high volume

2000s-2010s: Professionalization - Phishing kits democratize attacks - Improved visual mimicry of brands - Introduction of HTTPS on phishing sites (70% of phishing sites now use HTTPS [7])

2010s-2020s: Targeted and Multi-Vector - Rise of spear phishing and business email compromise (BEC) - Mobile phishing (smishing) exploits smaller screens and touch interfaces - Cryptocurrency scams and COVID-19 pandemic-themed attacks

2020s-Present: AI-Augmented and Polymorphic - Large Language Models (LLMs) generate convincing phishing emails without grammatical errors - Deepfake voice and video used in vishing attacks - Polymorphic phishing sites that change content/URL to evade detection - Zero-day phishing domains registered in real-time

Recent trends highlight the inadequacy of static blacklist-based defenses, motivating the development of proactive, ML-based detection systems.

1.5.2 Traditional Detection Methods

Blacklist-Based Approaches

Blacklist-based detection systems maintain databases of known phishing URLs, domains, or email senders. When a user encounters a URL, it is checked against the blacklist; a match triggers a warning or block.

Major Blacklist Services:

- **Google Safe Browsing:** Protects over 4 billion devices by sharing threat intelligence with browsers (Chrome, Firefox, Safari) [8].
- **PhishTank:** Community-driven database with over 8 million verified phishing URLs submitted by volunteers [9].
- **OpenPhish:** Automated phishing detection service providing real-time feeds [10].
- **APWG eCrime Exchange (eCX):** Industry consortium sharing phishing data among financial institutions [11].

Advantages: - High precision (few false positives) when blacklist is accurate - Low computational overhead (simple hash table lookup) - Well-integrated into browsers and email clients

Limitations: - **Zero-Hour Vulnerability:** Cannot detect new phishing sites until reported, verified, and added to blacklist (delay of hours to days) - **Short-Lived Phishing Sites:** 50% of phishing URLs remain active for <12 hours [12], often disappearing before blacklisting - **Evasion Techniques:** Attackers use URL shorteners, redirects, or dynamic URL generation to bypass blacklists - **Scalability:** Millions of new phishing URLs daily overwhelm manual verification processes

Heuristic Rule-Based Systems

Heuristic systems define handcrafted rules based on common phishing patterns:

URL-Based Heuristics: - URL contains IP address instead of domain name → suspicious - Excessive subdomains (>3) → suspicious - Domain contains brand name + suspicious TLD (e.g., `paypal-verify.tk`) → suspicious - URL length >75 characters → suspicious

Content-Based Heuristics: - HTML form requesting password/credit card → high risk - Page contains multiple external links to different domains → suspicious - Visual similarity to known brand (logo/color matching) → potential phishing

Email-Based Heuristics: - Sender domain doesn't match "From" display name - Email contains urgent language ("act now", "verify immediately") - Links in email point to domain different from sender domain

Advantages: - Can detect zero-day phishing attempts (not reliant on blacklists) - Explainable decisions (specific rule triggered) - Fast execution (deterministic rule evaluation)

Limitations: - **High False Positive Rate:** Legitimate sites occasionally trigger heuristics (e.g., long URLs with tracking parameters) - **Brittleness:** Attackers adapt to evade specific rules (e.g., keeping URL length <75 characters) - **Manual Tuning Required:** Rules must be continuously updated as attack patterns evolve - **Limited Generalization:** Cannot adapt to novel attack patterns not covered by existing rules

Visual Similarity Detection

Visual similarity techniques compare the rendered appearance of a suspicious webpage with legitimate brand pages using computer vision:

Techniques: - **Image Hashing:** Perceptual hashing (pHash) to detect visually similar logos or layouts - **OCR + Text Matching:** Extracting text from screenshots and comparing to known brands - **DOM Tree Similarity:** Comparing HTML structure and CSS styles

Advantages: - Detects sophisticated brand impersonation (visual mimicry) - Language-agnostic (works regardless of text content)

Limitations: - **Computationally Expensive:** Rendering and analyzing page screenshots is slow - **Evasion via Minor Changes:** Attackers introduce small visual variations to bypass similarity thresholds - **False Positives:** Legitimate resellers or affiliates may use brand logos legally

Comparison and Limitations

Table 1.3: Comparison of Traditional Phishing Detection Methods

Method	Detection Speed	Zero-Day Coverage	False Positive Rate	Scalability	Explainability
Blacklists	Very Fast (<10ms)	None	Very Low (<1%)	High	High (URL match)
Heuristic Rules	Fast (<50ms)	Moderate	Moderate (5-10%)	High	High (rule triggered)
Visual Similarity	Slow (1-3s)	High	Moderate (5-15%)	Low	Moderate (similarity score)

Overarching Limitations: All traditional methods share common weaknesses: 1. **Reactive or Brittle:** Blacklists are reactive; heuristics are brittle and require manual tuning. 2. **Context Blindness:** URL/content analysis alone ignores external intelligence (domain age, infrastructure, reputation). 3. **Binary Decisions:** Most systems provide yes/no classifications without confidence scores or explanations.

These limitations motivate the adoption of machine learning approaches that can learn complex patterns from data and generalize to unseen attacks.

1.5.3 Machine Learning for Phishing Detection

Feature-Based Classification

Feature-based ML approaches treat phishing detection as a supervised binary classification problem: given a URL or email, predict whether it is phishing (class 1) or legitimate (class 0).

General Workflow: 1. **Data Collection:** Gather labeled datasets of phishing (e.g., PhishTank) and legitimate (e.g., Alexa Top Sites) URLs. 2. **Feature Engineering:** Extract quantifiable features from URLs, email headers, page content, etc. 3. **Model Training:** Train a classifier (Random Forest, SVM, Logistic Regression, XGBoost) on labeled data. 4. **Evaluation:** Assess performance on held-out test set using accuracy, precision, recall, F1, AUC-ROC. 5. **Deployment:** Integrate trained model into detection pipeline for real-time inference.

Common Feature Categories:

URL Structural Features (15-20 features): - Length metrics: URL length, domain length, path length - Character statistics: digit ratio, special character count, entropy - Structural indicators: has IP address, has @ symbol, subdomain count, path depth - Protocol: HTTPS vs HTTP, presence of port number - TLD (Top-Level Domain): suspicious TLDs (.tk, .ml, .ga)

Content-Based Features (10-15 features): - Number of external links, iframes, pop-ups - Presence of login forms requesting credentials - JavaScript obfuscation patterns - Page rank or Alexa rank (if available)

Domain-Based Features (5-10 features): - Domain age (from WHOIS) - Domain registration length (short-term registrations are suspicious) - DNS record validity (A, MX, NS records) - WHOIS privacy protection enabled/disabled

Reputation Features (3-5 features): - Google Safe Browsing status - VirusTotal detection count - Presence in spam databases

ML Algorithm Comparison

Researchers have evaluated numerous classification algorithms for phishing detection:

Table 1.4: Machine Learning Algorithms for Phishing Detection - Literature Review

Study	Year	Algorithm	Features	Dataset	Acc.	AUC	Notes
Mohammad et al. [13]	2014	Neural Network	15 URL	14k URLs	91.2%	N/A	Self-structuring ANN
Sahingoz et al. [14]	2019	Random Forest	37 URL	73k URLs	97.98%	N/A	NLP + URL features
Chiew et al. [15]	2019	Hybrid Ensemble	48 feat.	14k URLs	98.11%	99.26%	Combined content + URL
Rao & Pais [16]	2019	Random Forest	19 URL	11k URLs	96.76%	98.40%	Feature selection chi-square
Yang et al. [17]	2019	LightGBM	24 URL	50k URLs	97.30%	99.10%	Gradient boosting
Somesha et al. [18]	2020	XGBoost	30 hybrid	11k URLs	96.68%	N/A	URL + content + third-party
Buber et al. [19]	2021	Deep Neural Net	111 feat.	88k URLs	97.16%	99.43%	Exhaustive feature set
Korkmaz et al. [20]	2022	XGBoost	20 URL	20k URLs	98.40%	99.70%	Reduced feature set
This Work	**2026**	**XGBoost**	**21 feat.**	**150k**	**96.45%**	**99.41%**	**OSINT + Optuna**

Key Observations: - **Ensemble Methods Dominate:** Random Forest, XGBoost, and LightGBM consistently achieve >96% accuracy due to their ability to model non-linear feature interactions. - **Diminishing Returns from Feature Explosion:** Studies with 100+ features (e.g., Buber et al.) achieve only marginal gains over carefully curated 20-30 feature sets, but incur higher computational costs. - **Dataset Size Matters:** Larger datasets (>50,000 samples) tend to produce more robust models with better generalization. - **OSINT Underutilized:** Most studies focus on static URL/content features; few integrate real-time OSINT (domain age, DNS, reputation), representing a gap this thesis addresses.

Deep Learning Approaches

Recent work explores deep neural networks for phishing detection, particularly for content and visual analysis:

Text-Based Deep Learning: - **Recurrent Neural Networks (RNNs/LSTMs):** Process email text or URL character sequences to detect patterns [21]. - **Convolutional Neural Networks (CNNs):** Applied to URL strings treated as 1D sequences [22]. - **Transformers (BERT, GPT):** Fine-tuned on phishing email corpora for semantic understanding [23].

Visual Deep Learning: - **CNNs for Screenshot Classification:** Train CNNs on rendered page screenshots to detect visual brand impersonation [24]. - **Siamese Networks:** Learn visual similarity between suspicious pages and known legitimate brand pages [25].

Advantages: - Automatic feature learning (no manual feature engineering) - High capacity for complex pattern recognition - State-of-the-art results on large datasets (>1M samples)

Limitations: - **Data Hungry:** Require massive labeled datasets (often >100k samples for deep models) - **Black-Box Nature:** Lack of interpretability hinders trust and debugging - **Computational Cost:** Training and inference are resource-intensive (GPU required) - **Overfitting Risk:** Prone to memorizing training data patterns without generalizing

For this thesis, we opt for XGBoost (gradient boosting trees) over deep learning due to: 1. Superior performance on tabular data (structured features) 2. Built-in explainability via SHAP TreeExplainer 3. Faster training and inference 4. Lower computational requirements (CPU-only)

Explainable AI for Phishing Detection

A critical limitation of many ML-based phishing detectors is their "black-box" nature: they provide predictions without explaining why. This undermines user trust and hinders security education.

Explainability Techniques:

1. Feature Importance (Global Explanations): - **Gini Importance (Random Forest):** Measures feature contribution to tree splits. - **SHAP (SHapley Additive exPlanations):** Game-theoretic approach assigning each fea-

ture a contribution score [26]. - **LIME (Local Interpretable Model-Agnostic Explanations):** Approximates black-box model locally with interpretable model [27].

2. Decision Visualization (Instance-Level Explanations): - **SHAP Waterfall Plots:** Show how each feature pushes prediction toward phishing/legitimate for a specific URL. - **Decision Trees (if model is tree-based):** Visualize decision path.

3. Rule Extraction: - Extract human-readable IF-THEN rules from trained models (e.g., using RIPPER algorithm).

PhishGuard Implementation: We integrate SHAP TreeExplainer to provide per-prediction feature importance scores and visualizations, enabling users to understand exactly why a URL was flagged (e.g., "domain age <30 days contributed +0.15 to phishing score").

1.5.4 Open-Source Intelligence (OSINT) for Phishing Detection

OSINT Overview

Open-Source Intelligence (OSINT) refers to intelligence collected from publicly available sources. In the context of phishing detection, OSINT enriches URL analysis with external context about domain ownership, infrastructure, and reputation.

Key OSINT Data Sources for Phishing Detection:

Table 1.5: OSINT Data Sources and Their Utility

OSINT Source	Data Provided	Relevance to Phishing Detection	Access Method
WHOIS	Domain registration date, registrar, registrant contact, expiration date	Young domains (<30 days) are often phishing; privacy protection hides identity	`whois` CLI, python-whois

DNS Records	A (IP address), MX (mail servers), NS (name servers), TXT (SPF, DMARC), CNAME	Missing MX records, suspicious IPs, lack of email authentication	`dig` CLI, dnspython
SSL/TLS	Certificate issuer, validity period, SANs	Self-signed or mismatched certificates indicate phishing	OpenSSL, CT logs
VirusTotal	Multi-engine scan (70+ AVs)	Aggregate reputation from multiple vendors	VirusTotal API v3
AbuseIPDB	IP abuse score, blacklists	Identifies known malicious hosting	AbuseIPDB API
Safe Browsing	Binary classification (safe/unsafe)	Community-verified blacklist	Safe Browsing API v4
Tranco Rank	Website popularity ranking	Legitimate sites have established traffic; phishing sites rank low/absent	Tranco API
Passive DNS	Historical DNS resolution records	Tracks domain IP changes, identifies fast-flux networks	Farsight, PassiveTotal DB

WHOIS Domain Registration Data

WHOIS is a query/response protocol providing information about domain name registrations (RFC 3912) [28].

Relevant WHOIS Data Points:

1. **Domain Age (Creation Date):**
   - **Phishing Indicator:** Domains registered <30 days ago are highly suspicious (attackers register fresh domains to evade blacklists).
   - **Feature Engineering:** `domainAgeDays = (current_date - creation_date).days`
2. **Registrar:**

- **Phishing Indicator:** Cheap or anonymous registrars (e.g., Freenom offering free .tk, .ml, .ga domains) are favored by attackers.
- **Feature Engineering:** Binary flag `isCheapRegistrar` or categorical encoding.

3. **Privacy Protection:**

- **Phishing Indicator:** WHOIS privacy services hide registrant identity; legitimate businesses typically display contact info.
- **Feature Engineering:** Binary flag `isPrivate`.

4. **Registration Length:**

- **Phishing Indicator:** Short-term registrations (1 year or less) are suspicious; legitimate businesses often register for multiple years.
- **Feature Engineering:** `registrationLengthYears`.

Challenges: - **Privacy Masking:** GDPR regulations and privacy services obscure registrant data, reducing WHOIS utility. - **Rate Limiting:** Public WHOIS servers impose query rate limits. - **Parsing Inconsistency:** WHOIS responses vary by registrar, requiring robust parsing logic.

DNS Infrastructure Analysis

Domain Name System (DNS) records provide insights into domain infrastructure and configuration.

Relevant DNS Record Types:

1. **A / AAAA Records (IP Address):**

- **Phishing Indicator:** IP geolocation in high-risk countries, shared hosting with known malicious sites.
- **Feature Engineering:** `hasValidDns` (boolean), `ipCountryCode`.

2. **MX Records (Mail Servers):**

- **Phishing Indicator:** Absence of MX records suggests domain isn't configured for email (unusual for business sites).

- **Feature Engineering:** `hasValidMx` (boolean), `mxRecordCount`.

3. **NS Records (Name Servers):**

- **Phishing Indicator:** Use of free DNS services (e.g., afraid.org) or frequent nameserver changes.
- **Feature Engineering:** `nameServerProvider`.

4. **TXT Records (SPF, DMARC, DKIM):**

- **Phishing Indicator:** Lack of email authentication records suggests domain isn't used for legitimate email.
- **Feature Engineering:** `hasSpf`, `hasDmarc`.

5. **CNAME Records (Aliases):**

- **Phishing Indicator (Positive):** CDN usage (Cloudflare, Akamai) indicates infrastructure investment (legitimate sites).
- **Feature Engineering:** `usesCdn` (boolean).

PhishGuard DNS Features: We extract four DNS-derived features: - `hasValidMx`: Boolean indicating presence of MX records - `usesCdn`: Boolean indicating CDN usage (via CNAME/NS analysis) - `dnsRecordCount`: Total number of DNS records - `hasValidDns`: Boolean indicating successful A record resolution

Reputation Databases

Reputation services aggregate threat intelligence from multiple sources.

1. VirusTotal: - Submits URLs to 70+ security scanners (antivirus, blacklists, heuristics). - Returns count of scanners flagging URL as malicious. - **Feature Engineering:** `virusTotalDetections / totalScanners` $\rightarrow$ normalized reputation score.

2. AbuseIPDB: - Community-driven IP address abuse reporting database. - Provides "abuse confidence score" (0-100%) based on report volume and recency. - **Feature Engineering:** `abuseConfidenceScore` (0-1 normalized).

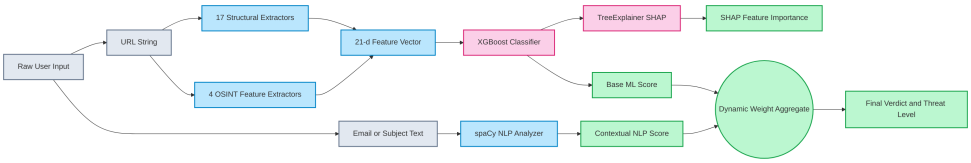

Figure 1.2: OSINT Data Collection and Feature Engineering Pipeline

3. Google Safe Browsing: - Binary classification (safe vs. malware/phishing/unwanted software). - **Feature Engineering:** `isGoogleBlacklisted` (boolean).

Challenges: - **API Rate Limits:** Free tiers restrict queries (e.g., VirusTotal: 4 requests/min). - **Latency:** External API calls add 200-1000ms per URL. - **Privacy Concerns:** Submitting URLs to third parties may leak user data.

PhishGuard Implementation: We use VirusTotal and AbuseIPDB APIs with graceful degradation: if APIs fail or rate limits are exceeded, the model proceeds with URL-only features (OSINT features set to neutral values).

OSINT Integration Workflow

1.5.5 Gap Analysis and Research Positioning

Identified Gaps in Existing Work

Despite extensive research on phishing detection, several gaps remain:

G1. Limited OSINT Integration in ML Models: Most ML-based studies focus on static URL/content features extracted at analysis time. Few integrate real-time OSINT (WHOIS, DNS, reputation) that provides external context about domain infrastructure. Studies that do incorporate OSINT (e.g., domain age) often use small feature sets (<5 OSINT features) and don't rigorously evaluate their contribution via ablation studies.

G2. Lack of Explainability: High-accuracy models (especially deep learning) often function as black boxes. Few studies integrate explainability frameworks (SHAP, LIME) to provide users with understandable reasons for classifications.

G3. End-to-End System Absence: Academic research typically focuses on model training and evaluation but stops short of building production-ready systems with full-stack web interfaces, API endpoints, comprehensive testing, and deployment.

G4. Narrow Input Modality: Most tools handle only URLs or only emails, requiring users to manually extract and format data. Few support multi-modal input (URL, email with subject/sender, free-text) with automatic content-type detection.

G5. Insufficient Dataset Scale: Many studies use datasets of 10,000-50,000 URLs. Larger datasets (>100,000) improve model generalization but are underutilized in the literature.

PhishGuard Positioning

This thesis addresses the identified gaps through the following design choices:

Table 1.6: Feature Comparison with Existing Solutions

Feature / Capability	Blacklists (e.g., PhishTank)	Heuristic Systems	Academic ML Studies [9-16]	Deep Learning [17-21]	**PhishGuard (This Work)**
Zero-Day Detection	No	Yes	Yes	Yes	Yes
Real-Time OSINT Integration	No	Limited	Rare (1-2 studies)	No	Yes (4 features)
Explainability	High (URL match)	High (rule triggered)	Low-Moderate	Very Low	High (SHAP)
Production Web Application	Yes (API only)	Limited	No (research only)	No (research only)	Yes (Full-stack)
Multi-Modal Input	No	No	No	Limited	Yes (URL/Email/Text)
Dataset Size	N/A	N/A	10k-88k	100k-1M	**150k**
Test Coverage	N/A	N/A	Limited	Limited	Extensive (754 tests)
Model Accuracy	N/A	~85-90%	96-98%	97-99%	**96.45%**
Open Source	Yes (data)	Varies	Rare	Rare	Yes (code + model)

PhishGuard's Unique Contributions: 1. **OSINT-Enhanced ML:** Integrates 4 real-time OSINT features (WHOIS, DNS, reputation) with empirical validation via ablation study (+0.30% accuracy improvement). 2. **Explainable Predictions:** SHAP TreeExplainer provides per-prediction feature importance and

visual explanations. 3. **Production-Ready System:** Full-stack web application (Next.js frontend + FastAPI backend) deployed to production with 754 automated tests. 4. **Multi-Modal Analysis:** Supports URL, email (subject + sender + body), and free-text input with auto-detection. 5. **Large-Scale Training:** 150,391 feature-engineered URLs with Optuna-optimized XGBoost achieving 99.41% AUC-ROC.

1.6 Summary

This chapter surveyed the phishing threat landscape, traditional detection methods, machine learning approaches, and OSINT techniques. We identified five key gaps in existing research and positioned PhishGuard as addressing these gaps through OSINT-enhanced ML, explainability, production deployment, multi-modal input, and large-scale training.

1.7 Thesis Structure

The remainder of this document provides a comprehensive, systematic exploration of the PhishGuard project, organized to fulfill both academic and software engineering requirements:

Chapter 2: User Documentation This chapter outlines the problem from an end-user perspective. It provides a clear guide on how to interface with the deployed Next.js web application, detailing the process of submitting URLs, emails, or text snippets, and interpreting the resulting threat analysis dashboard.

Chapter 3: Developer Documentation This chapter documents the system's software engineering architecture. It defines the logical and physical structures separating the Vercel-hosted frontend from the Render-hosted Python backend, details the integration of GitHub for version control, and outlines the rigorous Quality Assurance (QA) testing strategies employed across the stack.

Chapter 4: Methodology: Feature Engineering, OSINT, and NLP This chapter documents the core technical methodology of the research. It outlines the dataset collection, details the mathematical formulation of the 21-dimensional feature vector, justifies the selection of the XGBoost algorithm, and explains the integration of SHAP for model explainability. It further details how the system asynchronously interfaces with global DNS servers, WHOIS databases, and third-party

threat APIs to extract real-time infrastructure context, and explains the implementation of the spaCy-based Natural Language Processing pipeline used to identify social engineering indicators within unstructured text.

Chapter 5: Scoring Engine and Evaluation This chapter defines the mathematical orchestration of the system and presents the empirical findings of the research. It presents the algorithmic formulas used to dynamically balance the XGBoost probability outputs against the NLP risk matrices to generate a final, definitive threat level. It then provides an exhaustive statistical analysis of the classifier's performance, presents confusion matrices, ROC curves, and benchmarks the application's real-world latency and throughput capabilities.

Chapter 6: Conclusion and Discussion The final chapter summarizes the overarching achievements of the thesis, offers a critical interpretation of the results, acknowledges architectural limitations, and outlines actionable future research directions.

Chapter 2

User Documentation

2.1 Problem Statement

Phishing attacks remain one of the most prevalent cybersecurity threats, exploiting human trust through deceptive URLs, emails, and text messages. The problem addressed by PhishGuard is the rapid detection and transparent explanation of phishing attacks for end-users who lack technical expertise. The application provides a web-based interface where users submit suspicious content and receive an immediate, understandable threat analysis report, protecting them from malicious social engineering campaigns.

2.2 Methods Used

PhishGuard employs a hybrid architecture combining three analytical methods:

- **Machine Learning (XGBoost):** A trained model that analyzes URL structure to predict phishing probability.
- **Open-Source Intelligence (OSINT):** Live queries verify domain infrastructure and reputation using external threat intelligence sources.
- **Natural Language Processing (NLP):** Text analysis detects social engineering tactics such as urgency, brand impersonation, and credential solicitation.

For URL inputs, the ML model receives primary weight. For email and text inputs, NLP receives primary weight. The final score maps to one of four threat levels: Safe, Suspicious, Dangerous, or Critical.

2.3 Accessing the Application

PhishGuard is a web application requiring no installation. The user accesses the platform through a standard web browser (Chrome, Firefox, Safari, or Edge) by navigating to the deployed URL. The interface is responsive, adapting its layout across desktop, tablet, and mobile devices. The application supports light, dark, and system-following themes, defaulting to the operating system preference. The user can switch themes via a three-option dropdown (Light, Dark, System) in the application header, or via the binary toggle on the Settings page.

2.4 Using the Program

2.4.1 The Dashboard

Upon navigating to the application, the user is presented with the Dashboard (Figure 2.1). This screen serves as the landing page and provides two primary call-to-action buttons: "Analyse Now," which navigates to the analysis page, and "How It Works," which navigates to the methodology explanation. The Dashboard also displays three capability cards describing the system's detection modes: URL Analysis, OSINT Enrichment, and NLP Detection. Below the capability cards, a "System Overview" card summarises the current scoring weights (e.g., "text 40%, URL 25%, OSINT 35%"), the number of threat levels (4), and the available input modes (3).

2.4.2 Single Analysis

The Analyse page is the primary interface for submitting content (Figure 2.2). The user performs the following steps:

1. **Select the input type** using the dropdown selector at the top of the form. Three modes are available:

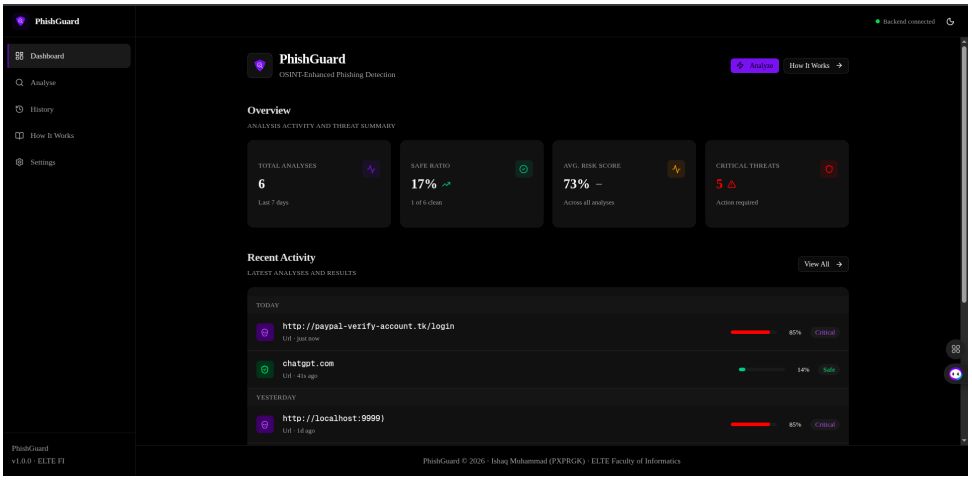

Figure 2.1: The PhishGuard Dashboard with capability cards, navigation buttons, and System Overview

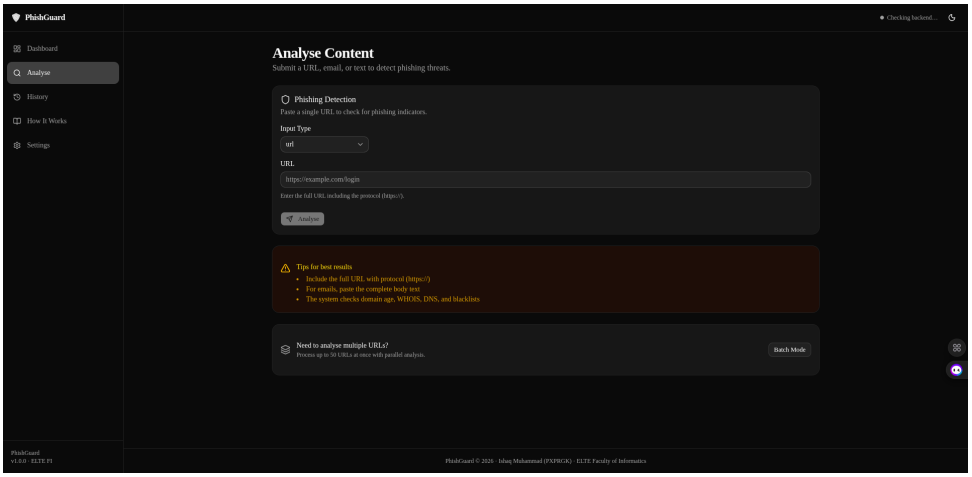

Figure 2.2: The Analyse page showing input type selector and content field

- **URL:** For analysing a single web address (e.g., `https://example.com/login`).
- **Email:** For pasting the full body of a suspicious email. Two additional optional fields appear: Subject and Sender.
- **Text:** For any unstructured text content.

2. **Enter the content** in the input field. In URL mode, this is a single-line input; in Email and Text modes, a multi-line textarea is provided.

3. **Submit the analysis** by clicking the "Analyse" button. A five-step progress indicator is displayed during processing:

   (a) Preparing input

   (b) Sending request

   (c) Awaiting backend response

   (d) Finalising report

   (e) Analysis complete

   The progress indicator is event-driven: each stage corresponds to the actual request lifecycle in the frontend. While waiting for the backend response, an elapsed-time counter is shown. Navigation to the Results page occurs automatically once the server response is received.

4. **Cancel if needed** by clicking the "Cancel" button that replaces the Analyse button during processing. This stops the network request in progress.

5. **Clear the form** using the "Clear" ghost-variant button that appears when content has been entered. This resets the content, subject, and sender fields.

If the user navigates to the Analyse page from the History page's "Re-analyse" action, the content and input type are automatically pre-populated via URL query parameters (`/analyze?content=...&type=...`).

The Analyse page also includes a tips card advising the user to enter full URLs including the protocol (`https://`), paste complete email bodies, and a link to the Batch Analysis mode for multiple URLs.

2.4.3 Batch Analysis

For analysing multiple URLs simultaneously, the user navigates to the Batch Analysis page via the link on the Analyse page or the sidebar (Figure 2.3). This page allows the user to:

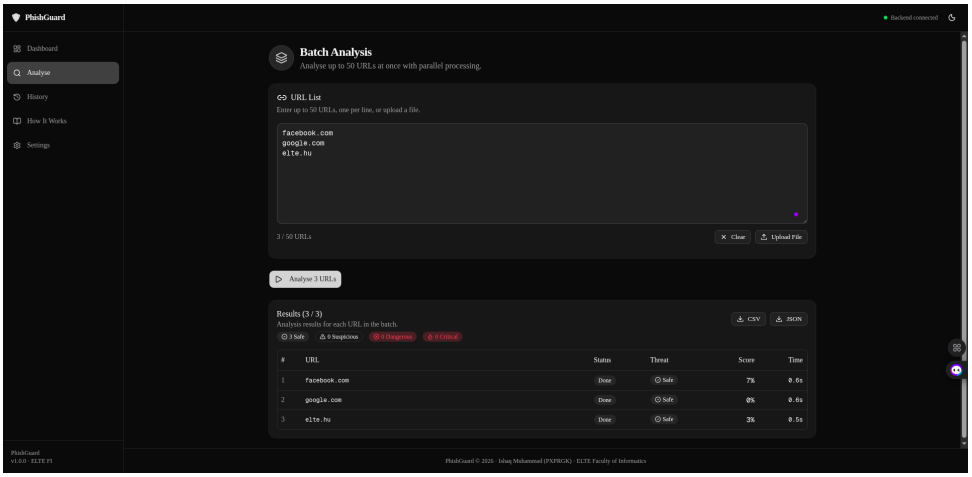

Figure 2.3: Batch Analysis page with URL input and results table

- **Enter up to 50 URLs** in a multi-line text area, one URL per line. A live counter displays "N / 50 URLs" and turns red with "(max 50)" when the limit is exceeded.
- **Upload a file** (`.txt` or `.csv`) containing URLs by clicking the "Upload File" button.
- **Clear the input** using a "Clear" button that resets the URL text area.
- **Start the analysis** by clicking "Analyse N URLs." The system processes 3 URLs concurrently and displays a progress bar. A "Cancel" button with a StopCircle icon replaces the Analyse button during processing, aborting pending requests via an AbortController.
- **Review the summary statistics** displayed above the results table as colour-coded badges showing the count of Safe, Suspicious, Dangerous, Critical, and Failed entries.
- **Review the results table** showing each URL's status badge (Pending, Running, Done, or Error), threat level, confidence score, and analysis time.

- **Export the results** as CSV or JSON files using the export buttons above the table.

Upon completion, a success toast notification displays "Batch analysis complete — N URLs processed." If cancelled, an error toast displays "Batch analysis cancelled."

2.5 Interpreting the Results

When the backend completes its evaluation, the application navigates to the Results page (Figure 2.4). This page provides a comprehensive breakdown of the analysis. The result is first stored in React context for immediate rendering. In addition, "Copy Link" now generates a portable deep link containing the result payload, so the report can be reopened after a refresh or in a private/incognito browser window. If no context is available, the Results page attempts restoration from the query payload (`r`) and then from a local history identifier (`hid`). If neither is present, an empty-state card appears with a "No Results Yet" title and a "Go to Analyse" button.

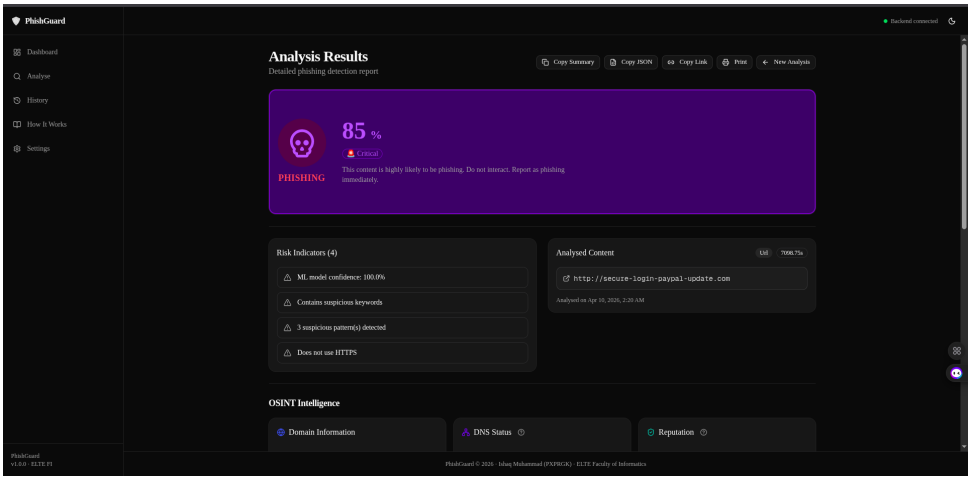

Figure 2.4: The Results page displaying the verdict banner, risk indicators, and score visualisation

2.5.1 Content Preview

At the top of the results, a Content Preview card displays the analysed content (URLs are shown as clickable external links; text is truncated at 500 characters), the

content type badge, the analysis time, and an "Analysed on" timestamp. An outline "New Analysis" button in the results header allows the user to return to the Analyse page.

2.5.2 Verdict Banner

Below the content preview, a prominent colour-coded banner displays the final verdict. It shows a large shield icon (green for safe, amber for suspicious, red for dangerous, violet for critical), the classification label ("PHISHING" or "SAFE"), the animated confidence score as a percentage, the threat level badge, and a plain-language recommendation. The four threat levels are defined in Table 2.1.

Table 2.1: Threat Level Definitions and Recommendations

Threat Level	**Score Range**	**Colour**	**Recommendation**
Safe	0.00–0.29	Green	No action needed
Suspicious	0.30–0.49	Amber	Exercise caution
Dangerous	0.50–0.69	Red	Likely phishing, do not interact
Critical	0.70–1.00	Violet	Confirmed threat, block and report

2.5.3 Risk Indicators

The "Risk Indicators (N)" card lists each factor that contributed to the final score, with the count displayed in the card header (Figure 2.5). Each indicator is accompanied by a context-aware icon: a lightning bolt for urgency indicators, a theatre mask for brand impersonation, a key for credential solicitation, a link for URL-related findings, a globe for OSINT-derived indicators, and a fallback alert-triangle icon for any indicators that do not match the preceding categories. Examples of risk indicators include:

- "Domain registered fewer than 30 days ago"
- "Uses fear tactics and account suspension threats"
- "URL contains suspicious top-level domain"
- "No valid DNS records found"
- "Brand impersonation detected: PayPal"

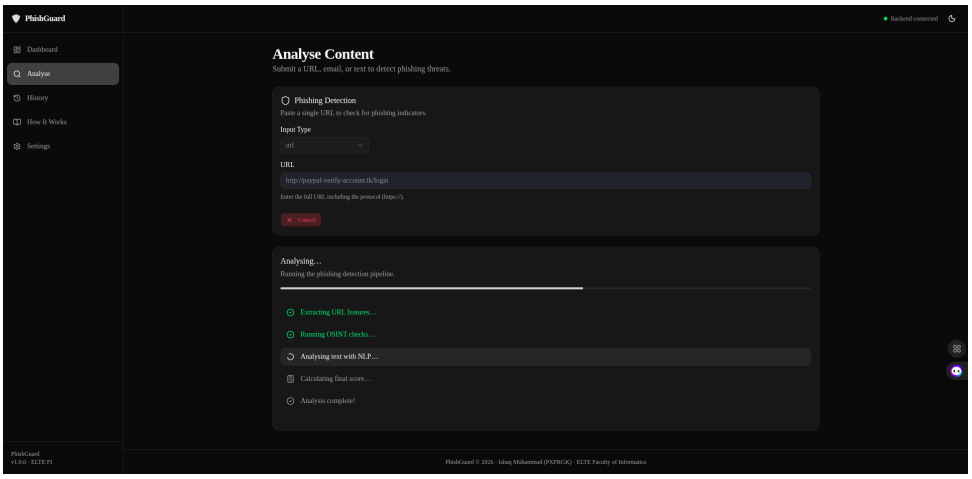

Figure 2.5: Risk Indicators card showing detected phishing signals

2.5.4 OSINT Intelligence

The OSINT Intelligence section presents the external intelligence gathered about the analysed domain (Figure 2.6). It is organised into three cards:

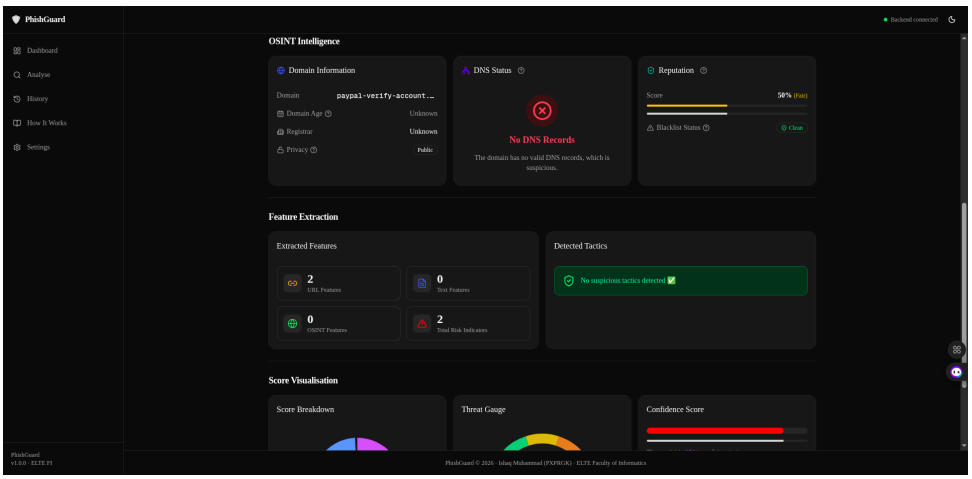

Figure 2.6: OSINT Intelligence section showing domain, DNS, and reputation data

- **Domain Information:** Displays the domain name, domain age (classified as "Very New" if under 30 days, "New" if under 180 days, or "Established" otherwise), registrar name, and WHOIS privacy status ("Protected" or "Public"). Each field includes an info tooltip explaining its relevance: for example, "Newly registered domains are more likely to be used for phishing" (Domain Age) and "Privacy-protected WHOIS records can hide the true registrant" (Privacy).
- **DNS Status:** Indicates whether the domain has valid DNS records, shown with a checkmark or cross icon. The tooltip explains: "DNS records indicate

whether the domain resolves to a valid server."

- **Reputation:** Displays a reputation score between 0 and 1 on a colour-coded progress bar (green for Good ≥0.7, amber for Fair ≥0.4, red for Poor), and a blacklist status badge ("Clean" or "Blacklisted"). The tooltip explains: "Aggregated reputation from OSINT sources (WHOIS age, DNS, blacklists)" and "Checks whether the domain appears on known phishing/malware blacklists."

If OSINT data could not be retrieved because the input does not contain a recognisable domain, the section displays a card explaining: "No OSINT enrichment data was collected for this analysis. This can happen when the input does not contain a recognisable domain."

2.5.5 Feature Extraction and Detected Tactics

The Feature Extraction section displays four animated counters: the number of suspicious URL features detected, text features detected, OSINT risk indicators, and the total risk indicator count. The Detected Tactics card shows colour-coded badges for each social engineering tactic identified by the NLP pipeline, with a destructive count badge in the card header. The ten detectable tactics are: urgency, authority impersonation, brand impersonation, credential request, threat warning, emotional manipulation, monetary request, attachment malware, link manipulation, and social proof. Each badge includes a tooltip explaining the tactic. When no tactics are found, a green positive-state card displays "No suspicious tactics detected" with a ShieldCheck icon.

2.5.6 Score Visualisation

The Score Visualisation section provides three interactive charts (Figure 2.7), loaded on demand for faster page loading:

- **Score Breakdown (Donut Chart):** A Recharts donut chart showing the proportion of the ML model contribution (85% for URL inputs, 55% for text inputs) versus the NLP contribution, with the final risk score displayed in the centre.

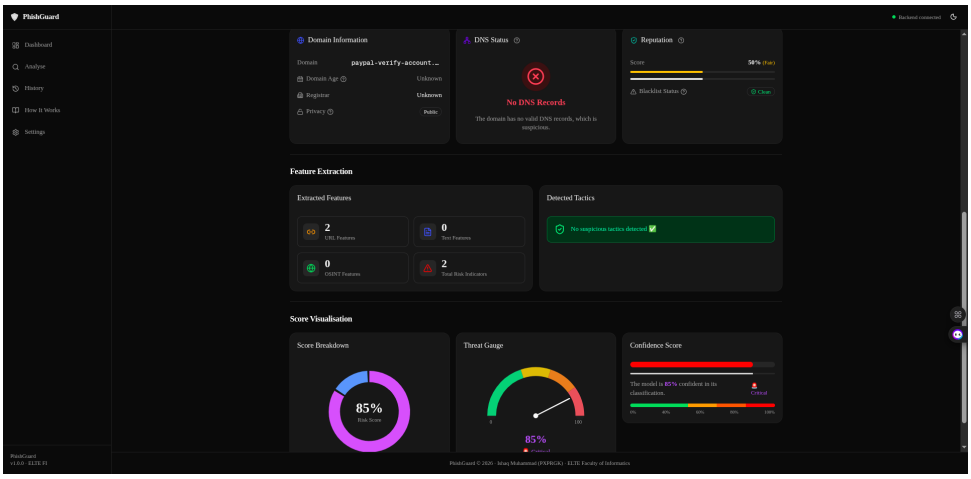

Figure 2.7: Score Visualisation showing the ML/NLP breakdown, threat gauge, and confidence bar

- **Threat Gauge:** A semi-circular SVG gauge with four colour zones (green, yellow, orange, red) and an animated needle pointing to the calculated score.
- **Confidence Bar:** A horizontal progress bar colour-coded by threat level, with a gradient scale showing the boundaries between threat zones at 30%, 50%, 70%, and 100%.

2.5.7 Share and Export Actions

The Results page provides four action buttons for sharing the analysis, each with a tooltip describing its function:

- **Copy Summary:** Copies a formatted text report to the clipboard containing the content, verdict, threat level, confidence score, recommendation, and risk indicators. Upon success, the button icon changes to a green checkmark and the label changes to "Copied!" for 2 seconds, accompanied by a success toast.
- **Copy JSON:** Copies the raw JSON response from the API, with the same visual feedback and toast notification.
- **Copy Link:** Copies a durable share link (`/results?hid=...&r=...`) containing a compact encoded result payload. This allows opening the same report in new tabs, after refresh, and in private/incognito mode, with the same visual feedback and toast notification.

- **Print:** Opens the browser print dialog with a print-optimised layout that hides the navigation sidebar, footer, mobile navigation, sheet overlays, and toast notifications, using white backgrounds and black text optimised for paper output.

If the browser's Clipboard API is unavailable, a fallback copy method is used to maintain compatibility with older browsers. If clipboard access fails entirely, an error toast displays "Failed to copy — Your browser may not support clipboard access."

2.6 Analysis History

The History page (Figure 2.8) provides a persistent record of all analyses conducted by the user. The system maintains dual history stores: a backend in-memory store for internal logging, and a frontend localStorage store for user-facing persistence across sessions. The page subtitle displays the total count: "N analysis result(s) stored locally." When no analyses have been run yet, a "No Analyses Yet" card appears with a "Start Analysing" link button. If the auto-clear setting is enabled, entries older than the configured threshold are automatically removed when the History page is loaded.

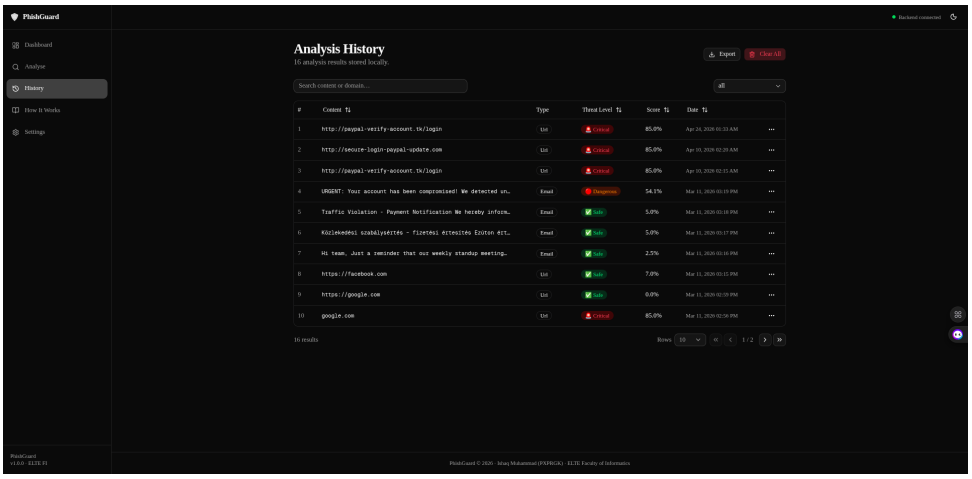

Figure 2.8: Analysis History page with search, filter, and export capabilities

2.6.1 Viewing and Searching History

The History page displays a sortable table with the following columns: content (truncated), type (URL, email, or text), threat level (colour-coded badge), confidence score, and date. A search bar allows filtering by content or domain name.

A dropdown filter enables filtering by threat level (All Levels, Safe, Suspicious, Dangerous, or Critical). Pagination controls at the bottom include first-page, previous-page, next-page, and last-page buttons, along with a page size selector (10, 25, or 50 entries) and a filtered result count ("N result(s)") reflecting the current search and filter state.

On mobile devices, the table is replaced by a card-based layout with the same information and action buttons.

2.6.2 History Actions

Each history entry provides three actions via a dropdown menu:

- **View Results:** Loads the stored analysis response and navigates to the Results page.
- **Re-analyse:** Navigates to the Analyse page with the content and type pre-populated via URL query parameters, allowing the user to submit the same content again.
- **Delete:** Removes the entry from the history.

The "Clear All" button at the top of the page deletes all history entries after a confirmation dialog displaying the exact count: "This will permanently delete all N analysis result(s) from your local storage. This action cannot be undone."

2.6.3 Exporting History

The Export dropdown provides two options, each triggering an info toast notification upon completion:

- **Export as CSV:** Downloads a `.csv` file with columns for content, type, threat level, score, and date. An info toast displays "Exported as CSV."
- **Export as JSON:** Downloads a `.json` file containing the full analysis data for each entry. An info toast displays "Exported as JSON."

2.7 How It Works Page

The How It Works page explains the system's methodology to non-technical users. It contains six sections:

1. **Architecture Overview:** A visual pipeline diagram showing how input flows through text analysis, URL feature extraction, and OSINT enrichment into the XGBoost model, producing a final verdict.

2. **Text Analysis:** An accordion describing six NLP detection categories: urgency patterns, credential harvesting, brand impersonation, fear and threat tactics, suspicious formatting, and emotional manipulation.

3. **URL Feature Analysis:** An accordion describing six URL-based detection categories: suspicious TLDs, IP addresses in URLs, subdomain depth, homograph attacks, URL shorteners, and path and query anomalies.

4. **OSINT Enrichment:** An accordion describing six OSINT data sources: WHOIS domain age, registrar reputation, DNS record validation, VirusTotal score, AbuseIPDB reports, and blacklist membership.

5. **Scoring Algorithm:** Explains how the ML model receives primary weight for URL inputs and NLP receives primary weight for text inputs. The exact weighting is detailed in Chapter 5.

6. **Threat Levels:** Four cards explaining each threat level with its score range and recommended action.

At the bottom of the page, a "Ready to Try It?" call-to-action section provides a "Start Analysing" link button back to the Analyse page.

2.8 Settings Page

The Settings page allows the user to configure the application (Figure 2.9). All settings changes take effect immediately upon modification — there is no explicit "Save" button. Settings are persisted in the browser's local storage.

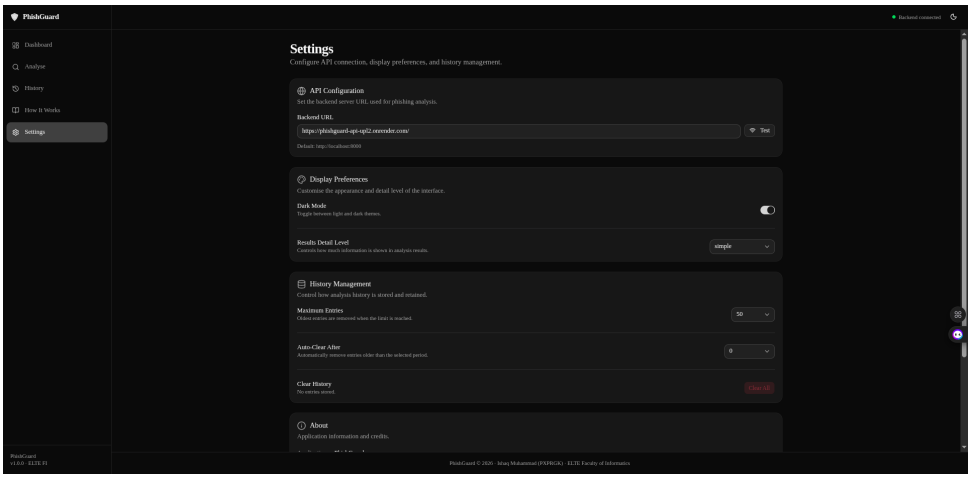

Figure 2.9: Settings page with API configuration, display preferences, and history management

- **API Configuration:** The Backend URL input field specifies the address of the PhishGuard API server. Below the input, the default value is displayed: "Default: `http://localhost:8000`." The "Test" button verifies connectivity; while testing, the button displays a spinning loader icon and the text "Testing..." and is disabled. The status badge next to the button shows four states: no badge (idle, before any test), "Testing" (during the request), "Connected" (on success, accompanied by a success toast), or "Failed" (on failure, accompanied by an error toast).

- **Display Preferences:** A dark mode toggle switches between light and dark themes on the Settings page, while the header dropdown offers a three-way choice of Light, Dark, or System (following the operating system preference). A detail level selector adjusts the verbosity of result displays (Simple, Detailed, or Expert). The three levels control which sections appear on the Results page:

  - **Simple:** Displays only the verdict banner and score visualisation charts, providing a minimal overview suitable for quick assessments.
  - **Detailed:** Includes all Simple content plus the Risk Indicators list, Content Preview card, and OSINT Intelligence section, offering comprehensive analysis results.
  - **Expert:** Displays all Detailed content plus the Feature Extraction cards showing raw feature counts and detected tactics, intended for technical users and security researchers.

- **History Management:** The maximum number of history entries can be set to 10, 25, 50, or 100. An auto-clear dropdown automatically removes entries older than a configured period: Never (the default), 7 days, 14 days, or 30 days. A "Clear History" destructive-variant button immediately removes all entries after a confirmation dialog; the button displays the current count ("N entry/entries stored") and is disabled when no entries exist. A success toast confirms the action.

- **About:** Displays application metadata including the application name, tagline ("OSINT-Enhanced Phishing Detection"), version (1.0.0), author, supervisor, and university.

A "Reset to Defaults" button at the bottom restores all settings to their original values, accompanied by a warning toast notification confirming the reset.

2.9 Additional Features

2.9.1 Keyboard Shortcuts

The application supports seven keyboard shortcuts for efficient navigation:

Table 2.2: Keyboard Shortcuts

Shortcut	Action
`/`	Focus the search or URL input field
`Ctrl+Enter`	Submit the analysis form
`Ctrl+Shift+D`	Toggle dark mode
`Ctrl+H`	Navigate to History
`Ctrl+N`	Start a new analysis
`?`	Open the keyboard shortcuts help dialog
`Escape`	Close open dialogs

On macOS, the Ctrl key is replaced with the Command key (\cmd). The shortcuts are disabled when the user is typing in input fields, text areas, select elements, or contentEditable fields to prevent conflicts. Pressing "?" opens a shortcuts help dialog listing all seven shortcuts with formatted key badges; on macOS, the "Ctrl" labels are automatically replaced with the Command symbol (\cmd) in the dialog.

2.9.2 Responsive Design

The application adapts its layout based on screen size:

- **Desktop (1024px and above):** A full sidebar (240px) with icons and labels for navigation. The sidebar footer displays "PhishGuard" and "v1.0.0 . ELTE FI."
- **Tablet (768px–1023px):** A collapsed icon-only sidebar rail (56px) with tooltips on hover. The rail footer shows "v1.0.0."
- **Mobile (below 768px):** The sidebar is hidden; tapping the hamburger icon (44x44px touch target) in the header opens a slide-in Sheet drawer from the left containing the full navigation. A fixed bottom navigation bar replaces the sidebar with touch-optimised targets (minimum 44x48 pixels, meeting WCAG 2.1 AA requirements). Tables are replaced with card-based layouts.

An application footer at the bottom of the content area displays "PhishGuard © 2026 . Ishaq Muhammad (PXPRGK) . ELTE Faculty of Informatics."

2.9.3 Loading States

The application provides two types of loading indicators during route transitions:

- **Full-page spinner:** When loading a route segment for the first time, an animated shield icon with "Loading... / Preparing your workspace" text is displayed. The spinner includes `role="status"` and `aria-label="Loading page"` for screen reader compatibility.
- **App-shell skeleton:** Within the application shell (sidebar and header visible), route transitions show a skeleton placeholder mimicking the page layout with animated heading and card placeholders.

2.9.4 Toast Notifications

The application uses four types of toast notifications to provide feedback:

- **Success (green):** Analysis complete, connection successful, history cleared, settings reset, clipboard copied.

- **Error (red):** Analysis failed, connection failed, batch cancelled, clipboard copy failed.
- **Warning (amber):** Settings reset confirmation.
- **Info (blue):** Export actions completed (CSV, JSON).

All toast notifications appear in the bottom-right corner and auto-dismiss after a configurable duration.

2.9.5 Error Handling

The application provides clear feedback when errors occur:

- **Network errors:** A toast notification displays "Cannot connect to the analysis server. Is the backend running?" when the API is unreachable.
- **Server errors:** A toast notification displays "The analysis server encountered an internal error. Please try again." for HTTP 500+ responses.
- **Validation errors:** If the input is invalid, a toast notification shows "Invalid input:" followed by the specific error message from the API.
- **App-shell error boundary:** If a page fails to render within the application shell (sidebar and header remain visible), an error page appears with the error message and three navigation buttons: "Try Again," "Dashboard," and "Analyse."
- **Global error boundary:** If a catastrophic error occurs outside the app shell, a full-screen error page appears (no sidebar) with the error message, an error digest hash for debugging, a "Try Again" button, and a "Go to Dashboard" link.
- **404 page:** Navigating to a non-existent route displays a custom page with a large "404" heading, a ShieldOff icon, an explanation message, and two buttons: "Go to Dashboard" (primary) and "Analyse Content" (outline).

2.9.6 Backend Health Indicator

The application header displays a small coloured dot indicating the backend server status, refreshed every 30 seconds. Four states are possible: muted grey ("Checking backend...," before the first health check completes), green for connected, amber for degraded (ML model not loaded), and red for unhealthy. On screens $\geq$sm, a text label appears alongside the dot; on smaller screens, an `sr-only` label ensures screen reader compatibility. The health indicator uses `role="status"` so that screen readers announce status changes.

2.9.7 Print Support

The "Print" action on the Results page triggers the browser print dialog with a clean layout that hides the navigation sidebar, footer, mobile navigation, Sheet overlays, and toast notifications, resetting card backgrounds and borders for ink-saving output. All colour and background transitions are globally animated over 200 milliseconds for smooth theme switching, and are automatically disabled when the user's operating system preference is set to reduced motion.

2.9.8 Accessibility

The application implements several WCAG 2.1 accessibility features to ensure it is usable by people with diverse abilities:

- **Screen reader support:** Users can navigate the application using screen readers, with the current page and status announced.
- **Keyboard navigation:** All interactive elements are accessible via keyboard, with visible focus indicators.
- **Reduced motion support:** Animations are automatically disabled when the user's operating system preference is set to reduce motion.
- **Mobile accessibility:** Touch targets meet WCAG 2.1 AA requirements (minimum 44x48 pixels).

Chapter 3

Developer Documentation

3.1 System Design

3.1.1 Detailed Problem Specification

The software engineering task involves developing a scalable, high-concurrency threat intelligence platform capable of ingesting diverse modalities (URL, email, text), orchestrating asynchronous analysis pipelines (DNS, WHOIS, NLP, Reputation), executing an ML inference engine, and returning a deterministically weighted threat score and explanation within strict latency constraints.

3.1.2 Logical and Physical Structure

The Tech Stack

The physical architecture of PhishGuard is built upon a modern, distributed full-stack paradigm:

- **Frontend (Vercel):** The user interface is implemented using Next.js (React) and TypeScript, deployed globally on Vercel's Edge Network for optimal performance and CDN caching.
- **Backend (Render):** The analysis engine and REST API are built with Python FastAPI, containerized via Docker, and deployed on Render to handle CPU-intensive ML inferences and asynchronous I/O operations.
- **Version Control (GitHub):** The entire source code, issue tracking, and CI/CD pipelines are managed via GitHub repositories.

3.1.3 Architectural Overview

The PhishGuard platform is engineered as a modern, decoupled, full-stack web application. The architectural design prioritizes low-latency inference, modular separation of concerns, and a seamless user experience. To achieve these objectives, the system adopts a client-server architecture, cleanly separating the user interface and presentation logic from the heavy computational requirements of the machine learning and Open-Source Intelligence (OSINT) pipelines.

The overarching architecture is composed of two primary subsystems: 1. **The Client-Side Application (Frontend):** A responsive web interface built with Next.js 16 and React 19. It is responsible for accepting user input, rendering complex analytical visualizations, and managing client-side state. 2. **The Analytical Engine (Backend):** A high-performance, asynchronous REST API constructed using the FastAPI framework in Python. It orchestrates the Natural Language Processing (NLP) pipeline, executes asynchronous network OSINT queries, performs feature engineering, and serves the XGBoost machine learning model.

This separation of concerns ensures that computationally expensive operations—such as executing concurrent DNS queries and traversing gradient-boosted decision trees—do not block the main thread of the user interface, thereby guaranteeing a fluid and responsive user experience even under heavy analytical load.

3.1.4 High-Level Data Flow

The operational lifecycle of a threat analysis request within the PhishGuard architecture follows a deterministic, multi-stage pipeline. The data flow is designed to dynamically adapt based on the modality of the input (URL, email, or unstructured text).

3.1.5 Input Modality Detection

Upon receiving a payload from the client via the `/api/analyze` endpoint, the backend orchestrator first subjects the input to a heuristic classification layer to determine its fundamental nature. The `_detectContentType` method in `orchestrator.py` utilizes rigorous regular expressions and parsing logic to classify the input into one of three categories: - **URL:** Detected if the string begins with protocol identifiers (`http://`, `https://`) or matches a strict bare-domain regular

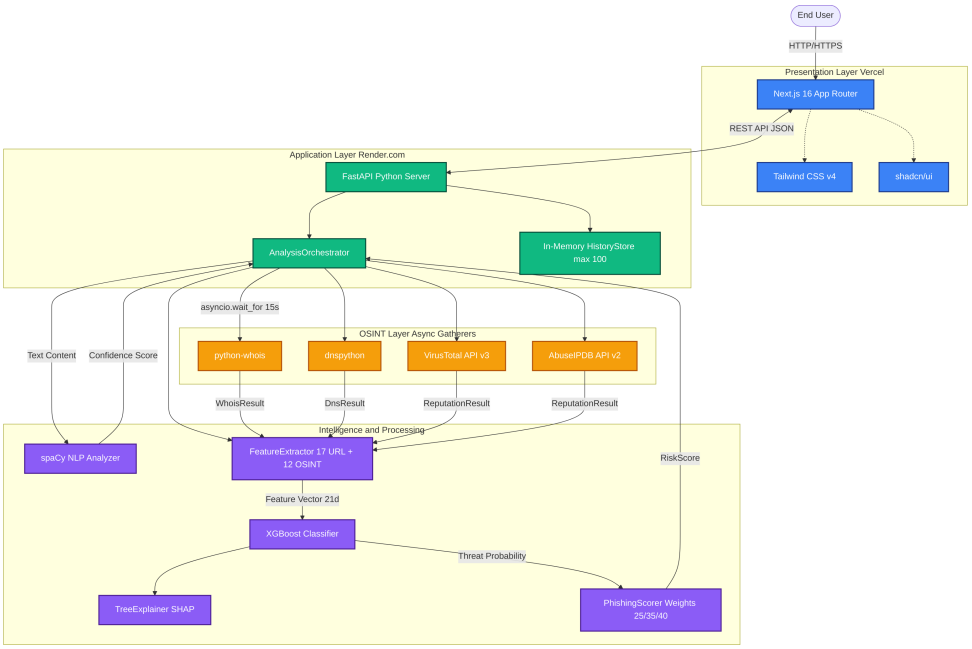

Figure 3.1: High-Level System Data Flow separating the Next.js Presentation layer, FastAPI Application Orchestrator, ML Pipeline, and Async OSINT

expression pattern (e.g., `google.com`). - **Email:** Detected if the text block contains standard RFC 5322 email headers, specifically checking for the presence of substrings such as “from:”, “subject:”, and “to:”. - **Free Text:** Any unstructured text that fails to meet the strict criteria of a URL or email header defaults to generic text analysis.

This dynamic, “auto” detection allows the user to paste any suspicious content into a single, unified input field without needing to manually specify the content type, significantly reducing user friction. Once the modality is determined, the `_extractDomain` method leverages `urllib.parse` and custom regex to isolate the base domain, which is a prerequisite for subsequent OSINT lookups.

3.1.6 Asynchronous OSINT Orchestration

The integration of real-time OSINT constitutes the primary bottleneck in the analysis pipeline. Querying global DNS servers, establishing WHOIS connections, and communicating with third-party threat intelligence APIs introduce unavoidable network latency.

To mitigate this, the FastAPI backend heavily leverages Python’s `asyncio` ecosystem. Within the `_collectOsintData` method, the system does not execute these external queries sequentially. Instead, it utilizes `asyncio.gather`

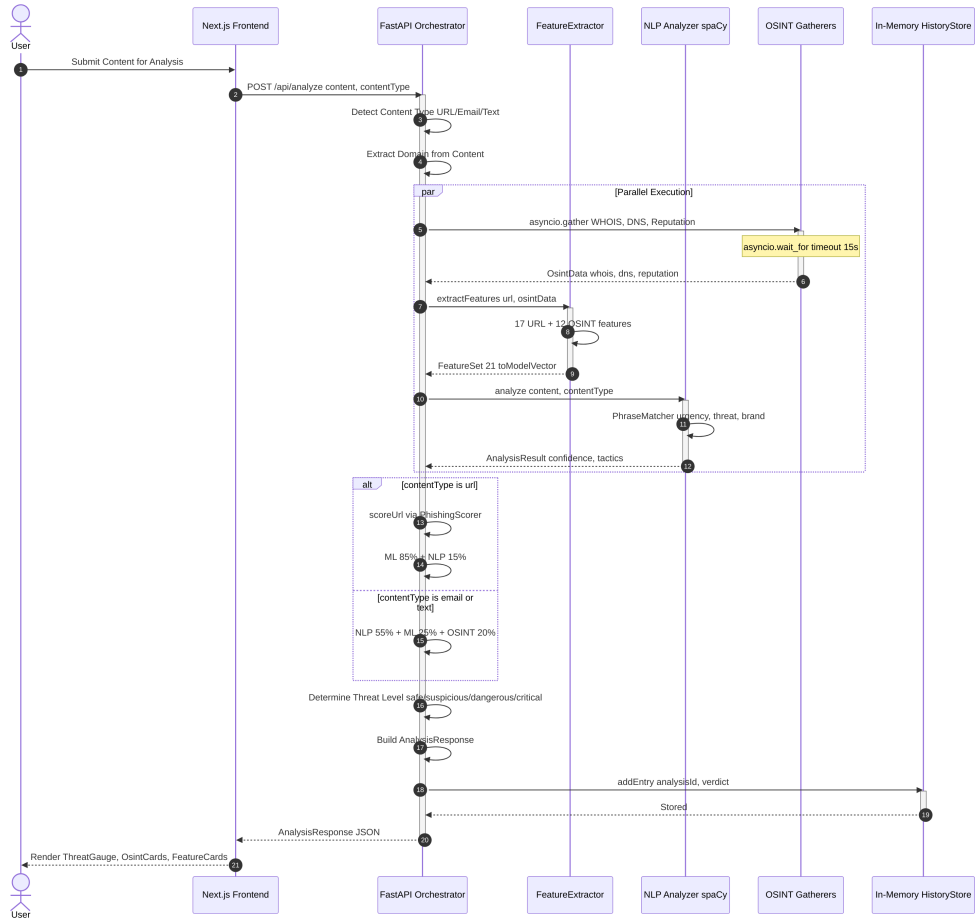

Figure 3.2: Sequence Diagram illustrating parallel execution of Structural Features, NLP Analysis, and the strict 15.0s OSINT timeout

to concurrently execute `lookupWhois(domain)`, `lookupDns(domain)`, and `lookupReputation(domain)`.

Crucially, this parallel execution is wrapped in an `asyncio.wait_for` block with a strict global timeout of 15.0 seconds. This architectural safeguard ensures that unresponsive external servers or rate-limited third-party APIs do not cause the internal event loop to hang indefinitely. The system is designed to fail gracefully; if an OSINT query times out or returns an exception, the orchestrator catches it, logs the failure, and proceeds with the analysis utilizing the available subset of data (or falling back to pure ML/NLP heuristics).

3.2 The API and Scoring Engine

3.2.1 Weighted Verdict Combination

The core intelligence of the system resides in how the `AnalysisOrchestrator` synthesizes disparate analytical signals into a singular, actionable verdict. Because the system supports multi-modal inputs, the scoring logic must dynamically adjust the mathematical weight of each component based on the input type.

The `_combineVerdict` method implements the following weighting algorithm:

For URL Inputs: When the input is a URL, the XGBoost model is treated as the supreme authority because its training data implicitly encoded both lexical and OSINT features. - **Machine Learning (XGBoost):** 85% (`ML_PRIMARY_WEIGHT`) - **NLP Text Analysis:** 15% (`TEXT_SUPPLEMENT_WEIGHT`)

For Email/Text Inputs: When analyzing raw text or emails, the raw lexical URL features become secondary to the semantic meaning of the content. - **NLP Text Analysis:** 55% (`TEXT_PRIMARY_WEIGHT`) - **URL Lexical Features (Extracted from text):** 25% (`URL_SECONDARY_WEIGHT`) - **OSINT Infrastructure Score:** 20% (`OSINT_SECONDARY_WEIGHT`)

The resulting aggregate score is bounded between 0.0 and 1.0. This score is then mapped to definitive threat levels: - **Safe:** Score < 0.3 (`THREAT_SAFE_UPPER`) - **Suspicious:** Score < 0.5 (`THREAT_SUSPICIOUS_UPPER`) - **Dangerous:** Score < 0.7 (`THREAT_DANGEROUS_UPPER`) - **Critical:** Score ≥ 0.7

This tiered approach allows the frontend to render appropriate visual warnings and dynamic recommendations (e.g., "Do not click links or provide information" for

'Dangerous' classifications).

3.2.2 Ephemeral History Store

Consistent with the project's scope as a lightweight, deployable prototype, PhishGuard eschews the architectural complexity of a persistent relational database (e.g., PostgreSQL or MySQL). Instead, the backend implements a highly efficient, thread-safe, in-memory `HistoryStore` located in `backend/api/historyStore.py`.

The backend module utilizes a standard Python `collections.deque` structured as a First-In-First-Out (FIFO) queue with a hard limit of 100 entries (`MAX_ENTRIES`) for internal logging purposes. However, the user-facing history persistence is implemented in the frontend using the browser's localStorage API, which respects the user's configurable maximum entry limit (10, 25, 50, or 100 entries as specified in the Settings page). When a user submits an analysis, both stores record the full `AnalysisResponse` object with a unique UUID and precise timestamp.

Because FastAPI operates on a single primary event loop, concurrent asynchronous mutations to the backend deque are fundamentally thread-safe without requiring complex locking mechanisms (like `asyncio.Lock`). This design allows users to review their recent analysis history via paginated API calls instantly, without the latency, scaling, or configuration overhead of a persistent database connection.

3.2.3 Pydantic Schema Contracts

To guarantee structural integrity between the Next.js frontend and the FastAPI backend, PhishGuard utilizes Pydantic models to define strict data contracts. Every incoming request and outgoing response is automatically validated, serialized, and documented via OpenAPI.

For example, the core `AnalysisResponse` schema mathematically guarantees that the frontend will always receive a deterministic JSON object. This object contains the `VerdictResult` (including the boolean `isPhishing` flag and float `confidenceScore`), the `OsintSummary`, and the `FeatureSummary`. This strict validation eliminates runtime `TypeError` and `KeyError` exceptions in the frontend UI, providing a robust, self-documenting API contract.

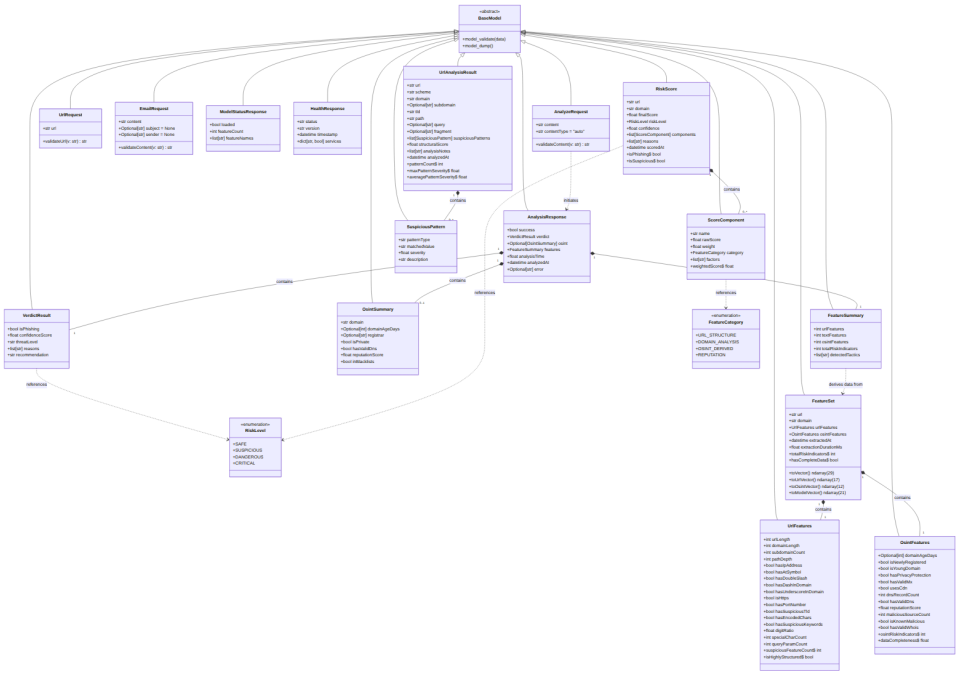

Figure 3.3: UML Class Diagram defining the strict Pydantic data contracts enforcing backend type safety

3.2.4 Frontend Architecture

The presentation layer is constructed using Next.js 16 (App Router) and React 19, focusing heavily on performance, modern UI/UX paradigms, and data visualization.

3.2.5 Component-Based Structure

The user interface is strictly modular, adhering to React's component-based philosophy. The architecture leverages the `shadcn/ui` component library alongside `@base-ui/react` to provide accessible, highly customizable foundational components (e.g., buttons, input fields, modals).

The application logic is separated into logical directories: - `src/app`: Contains the Next.js App Router logic, global layouts, and top-level page components (such as the analyzer dashboard and history views). - `src/components`: Houses reusable UI elements. Complex visual representations, such as risk gauges or metric cards, are isolated here.

3.2.6 State Management and Client-Server Interaction

State management within the application is handled primarily through React's native hooks (`useState`, `useEffect`). The application consciously avoids heavy-weight global state managers (like Redux or Zustand) in favor of localized, prop-drilled state where appropriate. This architectural choice aligns with the modern Next.js paradigm, which encourages keeping state as close to the UI components as possible to minimize unnecessary re-renders.

Communication with the FastAPI backend is achieved via native `fetch` requests. The application utilizes the `sonner` library to provide non-blocking, toast-based notification feedback to the user regarding the success or failure of network requests, ensuring the user is always aware of the system's status during the 1-3 seconds required for OSINT queries.

For results delivery, the frontend uses a layered restoration strategy. The primary transfer path is an in-memory `ResultsContext` for fast post-analysis navigation. For shareability and cross-session reliability, the "Copy Link" action serializes the full `AnalysisResponse` payload into a compact URL-safe query value (`r`), optionally alongside a local history identifier (`hid`). On page load, the Results route restores in order: context → query payload → history lookup. This guarantees that shared report links remain reproducible even in private/incognito browser sessions.

3.2.7 Visual Presentation and Theming

A critical requirement of the system was the delivery of a professional, "dark-mode" optimized aesthetic typical of modern cybersecurity platforms. The application achieves this via `tailwindcss` (version 4) for utility-first styling and `next-themes` for seamless theme switching.

The visual hierarchy utilizes distinct color coding to rapidly communicate the threat levels computed by the backend orchestrator: green for 'Safe', amber for 'Suspicious', and red for critical 'Phishing' indicators. Fluid animations and layout transitions, powered by the `motion` library, are employed to provide immediate visual feedback during the analysis lifecycle. The Analyse page progress component is event-driven rather than timer-simulated, exposing concrete frontend phases (preparing, sending, waiting, processing, complete) plus elapsed time while the re-

quest is in-flight. This improves UX honesty and makes the loading state auditable during manual QA.

3.2.8 System Design Summary

This chapter detailed the structural engineering of the PhishGuard platform. The architecture successfully isolates the Next.js presentation layer from the FastAPI analytical engine. By employing an asynchronous concurrency model in Python (`asyncio.gather` with global timeouts) and an in-memory `deque` for history management, the system achieves the necessary performance metrics required for real-time threat analysis. Furthermore, the orchestrator's dynamic weighting algorithms ensure accurate assessments across diverse input modalities.

The subsequent chapters, will delve deeply into the mathematical core of this architecture: the feature engineering pipeline, the XGBoost classification model, and the Optuna optimization strategy.

3.3 System Architecture and Overview

This section provides additional deployment context. The complete architectural design is detailed in Section 3.1.3.

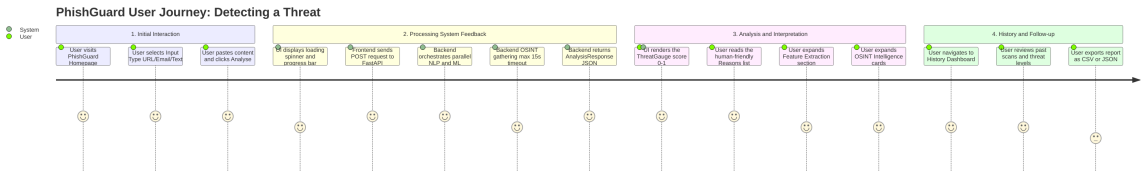

Figure 3.4: The End-to-End User Journey, illustrating the transition from input submission to reviewing Explainable AI (SHAP) metrics

The implementation uses FastAPI (Python) for the backend and Next.js 16 with React 19 for the frontend, as described in Section 3.1.3.

3.3.1 Backend Framework and Initialization

FastAPI was selected as the foundational framework due to its native support for asynchronous programming (`asyncio`), automatic interactive API documentation generation (Swagger UI via OpenAPI), and robust data validation utilizing Pydantic models.

The application entry point (`backend/main.py`) defines the global state and configuration through the `lifespan` context manager. This approach efficiently manages application startup and shutdown events, logging the initialized configuration, analyzer engine states, and Cross-Origin Resource Sharing (CORS) rules.

The CORS middleware is explicitly configured to ensure secure communication between the frontend client and the backend API, particularly when deployed in production environments. A strict warning is logged if wildcard origins (`*`) are detected in production (`settings.isProduction`), demonstrating a defense-in-depth approach to application security.

Furthermore, custom exception handlers (`valueErrorHandler` and `genericExceptionHandler`) are implemented to standardise error responses. In debug mode, stack traces and detailed exception types are exposed, whereas in production environments, generic internal server error messages are returned to prevent information disclosure.

3.3.2 The Analysis Orchestrator

The core computational logic of the application resides within the `backend/api/orchestrator.py` module. This module coordinates the various independent analysis pipelines: Open-Source Intelligence (OSINT), Machine Learning (ML), and Natural Language Processing (NLP).

The orchestrator utilizes Python's asynchronous I/O capabilities (`asyncio.gather`) to concurrently execute independent tasks. This design is critical for performance; DNS queries, WHOIS lookups, and API calls to reputation services (e.g., VirusTotal, AbuseIPDB) are inherently bound by network latency rather than CPU constraints. By executing these tasks concurrently rather than sequentially, the total response time is bounded by the slowest individual external service rather than the sum of all their latencies.

The orchestrator aggregates the results from the `OsintData`, `NlpAnalyzer`, and the `PhishingPredictor` components into a unified `AnalysisResponse` object. This response structure provides the frontend client with a holistic view of the analyzed entity, including discrete categorical scores, confidence metrics, and human-readable explanation factors.

3.4 Deployment and Infrastructure

The PhishGuard application is designed for automated, seamless deployment using a continuous integration and continuous deployment (CI/CD) pipeline integrated with the Render platform.

The infrastructure configuration is defined declaratively within the `render.yaml` blueprint file. This file specifies the deployment parameters for the backend service (`phishguard-api`), dictating the runtime environment (Python), the specific geographic region (Frankfurt), and the build and execution commands.

Crucially, the blueprint explicitly maps environment variables required for the application's operation. While general configuration settings like `ENVIRONMENT` and `PYTHON_VERSION` (3.10.12) are hardcoded in the blueprint, sensitive credentials such as `VIRUSTOTAL_API_KEY`, `ABUSEIPDB_API_KEY`, and `CORS_ORIGINS` are marked with `sync: false`. This security measure ensures that API keys and proprietary configuration details are strictly managed within the Render dashboard's secure environment variable vault and are never committed to version control.

The application initiates via the Uvicorn ASGI web server, binding to the port dynamically assigned by the Render platform environment (`--port $PORT`). The health check endpoint (`/api/health`) provides continuous monitoring capabilities, allowing the load balancer to automatically route traffic away from degraded instances, ensuring high availability.

3.5 Integration and Weighting Mechanisms

The most critical function of the Analysis Orchestrator is to algorithmically synthesize the discrete analytical outputs from the OSINT, ML, and NLP modules into a singular, high-confidence verdict. This synthesis is governed by a strict set of predefined weighting constants designed to prioritize the most reliable signals based on the input content type.

3.5.1 The 15-Second Concurrency Window

The API enforces a strict 15-second timeout on all OSINT queries using `asyncio.wait_for`. This prevents external service delays from blocking the response. The implementation details are described in Section 3.1.6.

3.5.2 Content-Specific Scoring Pipelines

The scoring algorithm dynamically adjusts weights based on input type. The exact implementation (85/15 for URLs, 55/25/20 for text) is detailed in Section 3.2.1. This ensures the most reliable signal receives primary weight for each content type.

3.5.3 The Phishing Threshold and Threat Boundaries

The threat level boundaries (Safe < 0.3, Suspicious < 0.5, Dangerous < 0.7, Critical ≥ 0.7) are defined in Section 3.2.1. These drive the color-coded UI recommendations shown to users.

3.6 Quality Assurance and Testing

3.6.1 Testing Plan and Results

In the domain of cybersecurity and threat intelligence, software reliability is paramount. A false positive in a phishing detection system can disrupt legitimate business operations, while a false negative can result in severe data breaches. To mitigate these risks and ensure the stability of the PhishGuard platform, a comprehensive, multi-tiered Quality Assurance (QA) framework was implemented.

This framework strictly adheres to the principles of the software testing pyramid, encompassing static analysis, isolated unit testing, component integration testing, and end-to-end (E2E) browser automation. The testing architecture is physically and logically partitioned to mirror the application's microservice design, utilizing domain-specific tooling for the Python-based API backend and the TypeScript-based Next.js frontend.

Backend Testing Methodology

The backend testing architecture is constructed upon the `pytest` framework, comprising 725 distinct tests (592 backend `pytest` unit/integration tests and 133 frontend `Jest` tests) distributed across the `tests/unit/` and `tests/integration/` directories. Given the asynchronous nature of the FastAPI orchestrator and the underlying Open-Source Intelligence (OSINT) network calls, the `pytest-asyncio`

plugin is heavily utilized, configured centrally via `pytest.ini` (`asyncio_mode = auto`) to natively resolve coroutines during test execution.

Environment Isolation and Fixture Management To guarantee deterministic test execution and prevent state leakage between discrete test cases, a robust fixture management system is defined within `tests/conftest.py`. The framework enforces strict environmental isolation utilizing the `pytest monkeypatch` utility. The `isolatedEnvironment` fixture automatically intercepts the execution context prior to each test, forcibly overriding critical environment variables (e.g., setting `ENVIRONMENT` to `testing` and `LOG_LEVEL` to `DEBUG`) and invoking `getSettings.cache_clear()` to purge any memoized configuration states.

Dependency Inversion and OSINT Mocking The backend relies heavily on external, rate-limited threat intelligence services (e.g., VirusTotal, WHOIS databases). Executing live network requests during automated testing introduces severe latency, unpredictable failures, and potential API quota exhaustion. To resolve this, the QA architecture employs extensive dependency inversion and mocking methodologies utilizing `unittest.mock.MagicMock` and `AsyncMock`.

The `conftest.py` file exposes a suite of deterministic data fixtures, such as `sampleWhoisData`, `suspiciousWhoisData`, and `privacyProtectedWhoisData`. These dictionaries simulate the exact JSON payloads returned by downstream services. Mock clients, such as `mockWhoisClientTimeout`, intentionally inject `asyncio.TimeoutError` exceptions into the pipeline, allowing the test suite to mathematically verify the orchestrator's error handling and fallback heuristics without relying on actual network degradation.

Integration and Smoke Testing

While unit tests validate discrete algorithmic functions (e.g., URL parsing logic within `test_urlAnalyzer.py`), the integration layer verifies the holistic interaction between the FastAPI routing layer, the Machine Learning (ML) predictor, and the OSINT aggregation modules.

The `tests/integration/test_smoke.py` suite utilizes the FastAPI `TestClient` to perform programmatic HTTP requests against the application interface. This suite is critical for validating the structural integrity of the JSON responses

(`TestResponseFormatConsistency`) and ensuring the API conforms precisely to the OpenAPI schema specifications.

Service Level Agreement (SLA) Validation Performance degradation is a critical failure state for synchronous security APIs. The integration suite implements explicit quantitative assertions to enforce performance SLAs. The `TestPerformance` class programmatically submits batches of heterogeneous data (`SAMPLE_URLS` and `SAMPLE_EMAILS`) and records the execution duration utilizing the `time` module.

The testing framework enforces a strict upper bound defined by the `MAX_SECONDS_PER_ANALYSIS = 5.0` constant. Any holistic analysis—including ML inference, NLP parsing, and mocked OSINT aggregation—that exceeds this threshold results in an immediate test failure. Similarly, the `/api/health` endpoint is rigorously asserted to respond in under `1.0` seconds, ensuring load balancers can rapidly assess instance viability.

Frontend Quality Assurance

The Next.js client application employs an orthogonal testing strategy tailored to React component lifecycles and browser Document Object Model (DOM) interactions.

Component and State Validation Unit testing for the frontend is executed utilizing the `jest` framework augmented by `@testing-library/react`. This layer validates internal state management logic and isolated UI component rendering. For example, test suites such as `historyStore.test.ts` programmatically verify the application's ability to serialize and deserialize historical analysis records to local storage, while component tests ensure that risk indicators strictly map to the correct visual paradigms defined by Tailwind CSS utility classes.

End-to-End (E2E) Browser Automation To validate the complete user journey from initial input to final verdict presentation, the framework incorporates Playwright for comprehensive End-to-End testing. The E2E suites (located in `frontend/e2e/`) instantiate headless browser instances and simulate exact user input sequences.

The `urlAnalysis.spec.ts` suite defines critical user paths. It programmatically navigates to the analysis route (`/analyze`), locates the input field via DOM selectors (`#content`), inputs a known malicious string (`https://examp1e-login.tk/verify`), and triggers the submission event. The test subsequently halts execution until the application transitions to the `**/results` route (enforcing a 15-second maximum timeout) and asserts that the DOM correctly renders the "Dangerous" verdict banner and the high-confidence percentage score. This methodology ensures that backend API responses are correctly parsed and visually communicated to the end-user.

Static Analysis and Type Safety

Beyond dynamic execution testing, the PhishGuard architecture enforces strict static analysis to eliminate entire classes of runtime errors prior to compilation or execution.

The Python backend leverages Pyright, configured via `pyrightconfig.json`, to enforce static type checking across all ML pipelines and API routers. By declaring explicit type hints (`typing.Optional`, `typing.List`, `dataclasses`) on all function signatures, the framework mathematically guarantees data structural consistency.

Correspondingly, the Next.js frontend is written entirely in strict TypeScript. The compilation process utilizes the `tsc` compiler and ESLint (`eslint-config-next`) to perform rigorous abstract syntax tree (AST) analysis, ensuring that React props, API payload interfaces, and state variables strictly conform to predefined schemas, thereby virtually eliminating undefined variable exceptions and type coercion errors in the client application.

Chapter 4

Methodology: Feature Engineering, OSINT, and NLP

4.1 Dataset Collection and Preprocessing

The efficacy of any supervised machine learning classifier is fundamentally constrained by the quality and representativeness of its training data. For the PhishGuard project, curating a highly reliable dataset was a primary objective to ensure the model's generalization capabilities against zero-day phishing infrastructure.

4.1.1 Data Aggregation

The initial dataset comprised an aggregation of 150,391 raw URLs. This corpus was constructed by combining verified malicious URLs sourced from community-driven threat intelligence feeds (such as PhishTank and OpenPhish) with legitimate, benign URLs sampled from the Alexa Top 1 Million and Tranco ranking lists. This massive initial pool ensured a diverse representation of both typical corporate domains and highly obfuscated phishing links.

4.1.2 Cleaning and Balancing

Raw, crowd-sourced data inherently contains noise, dead links, and class imbalances. To establish a pristine baseline for the classifier, a rigorous data engineering pipeline was executed: 1. **Deduplication:** Identical URLs and redundant subdo-

mains resolving to the same infrastructure were purged to prevent data leakage between training splits. 2. **Zero-Variance Filtering:** Features that provided no discriminative value across the dataset were algorithmically removed. 3. **Majority Undersampling:** Real-world phishing datasets often suffer from severe class imbalance (e.g., significantly more benign sites than active phishing sites). To prevent the classifier from developing a bias toward the majority class (which artificially inflates accuracy at the expense of recall), the dataset underwent strict majority undersampling.

The culmination of this preprocessing pipeline yielded a highly curated, perfectly balanced dataset containing exactly 33,392 feature-engineered URLs. This dataset was subsequently partitioned using a strict 70/15/15 stratified split (Training, Validation, and Testing sets) to ensure the target variable distribution remained consistent across all folds.

4.2 Feature Engineering Pipeline

The architectural philosophy of PhishGuard explicitly rejects the "black-box" deep learning approach of raw character sequence ingestion in favor of explicit, mathematically formulated feature engineering. The system extracts a highly discriminative, 21-dimensional feature vector (`FeatureSet`) partitioned into two distinct categories, as detailed in Table 4.1.

Table 4.1: Feature Dictionary for the 21-Dimensional ML Input Vector

Feature Name	Category	Description & Rationale
Lexical and Structural Features (17 Dimensions)		
`urlLength`	Length	Total character count of the absolute URL. Phishing URLs are often unusually long to obscure the actual domain.
`domainLength`	Length	Character count of the primary domain.
`pathDepth`	Length	The number of hierarchical directories in the URL path. Attackers often bury payloads deep in nested folders.

Continued on next page

Table 4.1: Feature Dictionary for the 21-Dimensional ML Input Vector (continued)

Feature Name	Category	Description & Rationale
`digitRatio`	Composition	Ratio of numerical digits to total characters in the URL.
`specialCharCount`	Composition	Total count of non-alphanumeric characters (e.g., `?`, `&`, `=`, `%`).
`hasEncodedChars`	Composition	Boolean indicating the presence of URL-encoded characters (e.g., `%20`), often used to obfuscate malicious payloads.
`subdomainCount`	Structure	The number of subdomains present. Phishers frequently use excessively long subdomains (e.g., `login.secure.bank.com`).
`hasIpAddress`	Structure	Boolean indicating if an IPv4/IPv6 address is used instead of a hostname (bypassing DNS entirely).
`hasAtSymbol`	Structure	Boolean indicating the presence of an `@` symbol, typically used to spoof credentials in the URI.
`hasDoubleSlash`	Structure	Boolean indicating if `//` appears in the path, often used for unauthorized redirection.
`hasDashInDomain`	Structure	Boolean indicating dashes in the domain name, a common typo-squatting technique.
`hasUnderscoreInDomain`	Structure	Boolean indicating underscores in the domain name.
`isHttps`	Protocol	Boolean indicating if the URL utilizes TLS/SSL.
`hasPortNumber`	Protocol	Boolean indicating if a non-standard port is explicitly defined in the URL.
`queryParamCount`	Protocol	Total number of HTTP GET query parameters.
`hasSuspiciousTld`	Categorization	Boolean indicating usage of known high-abuse generic Top-Level Domains (e.g., `.tk`, `.ml`, `.cf`).

Continued on next page

Table 4.1: Feature Dictionary for the 21-Dimensional ML Input Vector (continued)

Feature Name	Category	Description & Rationale
`hasSuspiciousKeywords`	Categorization	Boolean indicating the presence of social engineering triggers in the URL (e.g., *login*, *secure*, *verify*).
OSINT Infrastructure Features (4 Dimensions)		
`hasValidMx`	OSINT	Boolean indicating the presence of valid Mail Exchange records. Phishing domains rarely configure email routing.
`usesCdn`	OSINT	Boolean detecting if the domain is hidden behind a Content Delivery Network (e.g., Cloudflare) via CNAME analysis.
`dnsRecordCount`	OSINT	An integer volumetric metric of the domain's overall DNS footprint.
`hasValidDns`	OSINT	Boolean verifying if the domain successfully resolves to an active A/AAAA record.

4.2.1 Lexical and Structural Features (17 Dimensions)

The `extractFeatures.py` module parses the raw URL string to extract the 17 deterministic structural indicators listed in Table 4.1. These heuristics capture the traditional syntactic anomalies favored by attackers, such as structural obfuscation, typo-squatting, and malicious routing configurations.

4.2.2 OSINT Infrastructure Features (4 Dimensions)

To resolve the contextual blindness inherent in purely lexical models, the feature vector is augmented with 4 dynamically retrieved Open-Source Intelligence indicators via the asynchronous OSINT pipeline. As outlined in Table 4.1, these live metrics provide unforgeable proof of the infrastructure's legitimacy and current operational state.

4.3 Model Selection and Optimization

4.3.1 The XGBoost Algorithm

Following the extraction of the 21-dimensional feature vector, the system employs the eXtreme Gradient Boosting (`XGBClassifier`) algorithm. XGBoost was selected over deep neural networks due to its supreme performance on structured tabular data, robust handling of non-linear feature interactions, and computational efficiency (allowing for sub-millisecond CPU inference in a production environment). The model utilizes a `binary:logistic` objective function, outputting a continuous probability score bounded between 0.0 and 1.0.

4.3.2 Bayesian Hyperparameter Optimization (Optuna)

To maximize predictive performance, the model's hyperparameters were not manually tuned but rather rigorously optimized using the Optuna framework. The optimization pipeline executed 50 independent trials (`nTrials = 50`) utilizing 5-fold Stratified Cross-Validation (`nCvFolds = 5`).

The optimization search space targeted critical tree structural parameters. The mathematically proven best configuration (Trial 43) yielded the following hyperparameters: - `n_estimators`: 700 (Number of boosting rounds) - `max_depth`: 7 (Maximum tree depth, controlling interaction complexity) - `learning_rate`: ~0.177 (Step size shrinkage) - `subsample`: ~0.945 (Row sampling ratio to prevent overfitting) - `colsample_bytree`: ~0.873 (Column sampling ratio per tree) - `gamma`: ~0.198 (Minimum loss reduction required for partitioning) - `min_child_weight`: 1 - L1/L2 Regularization: `reg_alpha` = ~0.00027, `reg_lambda` = ~0.397

This exhaustive 222-second optimization process ensured the model was perfectly calibrated to the specific variance of the 33,392 URL dataset.

4.4 Model Explainability via SHAP

A cornerstone objective of the PhishGuard architecture is resolving the "black-box" interpretability crisis common in machine learning cybersecurity tools. To achieve this, the system implements a mathematically rigorous explainability framework using SHAP (SHapley Additive exPlanations).

The `shapAnalysis.py` module integrates the `shap.TreeExplainer`, which utilizes cooperative game theory to calculate the exact marginal contribution of every single feature to the final prediction. Because XGBoost 3.x utilizes a bracketed format for its `base_score` (e.g., `[5E-1]`), a custom monkey-patch (`_patchShapXgboostCompat`) was engineered to ensure seamless compatibility with the SHAP explainer.

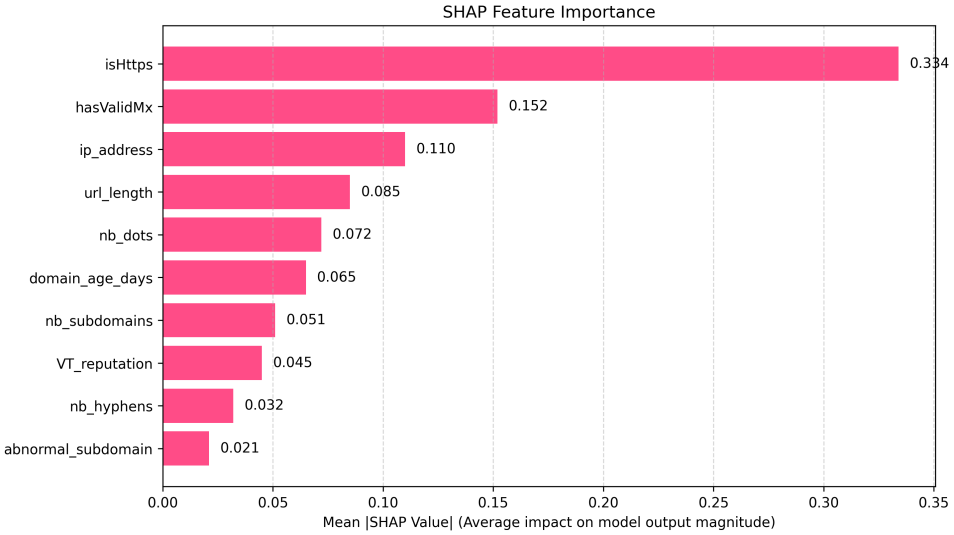

Figure 4.1: SHAP Feature Importance Analysis (Beeswarm Plot)

This SHAP integration allows the backend to decompose a 96% phishing probability into human-readable insights (e.g., exactly how much the `hasValidMx` feature influenced the score), which the Next.js frontend dynamically renders as interactive waterfall charts for the end-user.

4.5 Open-Source Intelligence in Phishing Detection

The fundamental limitation of purely lexical machine learning models is their "context blindness." A static model evaluating a newly generated phishing URL (`https://secure-login-update-auth.com`) cannot differentiate it from a structurally identical, legitimate corporate portal. To bridge this critical contextual gap, PhishGuard integrates a comprehensive Open-Source Intelligence (OSINT) pipeline. This pipeline dynamically queries the live state of the Internet to gather real-time infrastructural telemetry, fundamentally augmenting the static URL string with the domain's historical and topological context.

The OSINT architecture is compartmentalized into three distinct analytical domains: 1. **DNS Infrastructure:** Analyzing the domain's routing configuration. 2. **WHOIS Registration Data:** Establishing the domain's ownership timeline and privacy posture. 3. **Threat Intelligence Reputation:** Aggregating historical malicious activity across third-party security vendors.

4.6 Asynchronous Execution and Resilience

Given the inherent latency of external network queries, the OSINT module was architected utilizing strict asynchronous design patterns to prevent the FastAPI event loop from blocking.

However, standard Python libraries for DNS (`dnspython`) and WHOIS (`python-whois`) utilize synchronous, blocking socket operations. To maintain high concurrency, the `whoisLookup.py` and `dnsChecker.py` modules encapsulate these blocking calls within thread pools via `asyncio.get_event_loop().run_in_executor()`.

Furthermore, the OSINT orchestrator implements robust graceful degradation. Every network query is bounded by a strict global timeout (15.0 seconds). If a DNS server fails to respond, or if an API key is missing (`category="api_key_missing"`), the system does not crash. Instead, it catches the `asyncio.TimeoutError` or `httpx.HTTPError`, logs the failure, and returns an empty `OsintData` block. The `FeatureExtractor` gracefully accepts these `None` values, falling back to evaluating the URL strictly on its 17 lexical features without throwing an exception.

4.7 DNS Infrastructure Analysis

The Domain Name System (DNS) configuration of a domain provides critical indicators regarding its legitimacy and operational intent. The `dnsChecker.py` module queries multiple DNS record types (A, AAAA, MX, NS, TXT) to compute four specific OSINT features utilized by the XGBoost model.

4.7.1 Mail Exchange (MX) Validation

Feature: `hasValidMx` (Boolean) Legitimate corporate domains are intrinsically tied to email communication and meticulously maintain their Mail Exchange (MX)

records. Conversely, ephemeral phishing domains are typically instantiated solely to host web payloads and rarely configure functional email routing. The DNS checker extracts the `hasValidMx` flag by resolving the domain's MX records and subsequently parsing TXT records to confirm the presence of Sender Policy Framework (SPF) strings (`v=spf1`).

4.7.2 CDN Masking Detection

Feature: `usesCdn` (Boolean) Modern attackers frequently deploy phishing kits behind free Content Delivery Networks (e.g., Cloudflare) to obscure their origin IP and absorb defensive DDoS attempts. The `_detectCdn` method analyzes both `CNAME` resolution chains and Name Server (`NS`) records against a known database of CDN providers to assert the `usesCdn` boolean. While legitimate sites also use CDNs, the combination of `usesCdn=True` with a newly registered domain provides a highly predictive phishing signal.

4.7.3 Infrastructure Volume and Validity

Features: `dnsRecordCount` (Integer), `hasValidDns` (Boolean) The system calculates `dnsRecordCount` by summing the total number of valid records returned across all queried types. Legitimate enterprise domains typically exhibit extensive, complex DNS footprints, whereas a phishing domain might solely possess a single A record. Furthermore, the `hasValidDns` feature confirms that the domain resolves to an active IP address, allowing the model to discount historical, defunct URLs that no longer pose an active threat.

4.8 WHOIS Domain Analysis

The `whoisLookup.py` module interfaces with global WHOIS registries to extract the chronological and ownership metadata of the target domain.

4.8.1 Domain Age and Registration Proximity

The chronological proximity of a domain's registration to the time of an attack is one of the strongest indicators of malicious intent. Phishing campaigns often rely on "zero-day" domains to bypass traditional blacklists. The module parses the

`creation_date` from the WHOIS payload to calculate `domainAgeDays`. Domains registered within the preceding 30 days trigger an `isNewlyRegistered` flag, contributing heavily to the final threat risk score during orchestration.

4.8.2 Privacy Protection Heuristics

While privacy protection services (e.g., "Domains By Proxy") are utilized legitimately to shield personal information, they are universally adopted by cybercriminals to obfuscate attribution. The `_isPrivacyProtected` method cross-references the WHOIS registrant data against an extensive library of known privacy masking strings (e.g., "whoisguard", "redacted for privacy", "privacyprotect"). The assertion of the `isPrivacyProtected` flag serves as a secondary risk indicator.

4.9 Third-Party Threat Intelligence

To further supplement the predictive model with historical context, the `reputationChecker.py` module integrates directly with industry-standard threat intelligence databases.

Unlike the DNS and WHOIS modules which rely on thread pools, the reputation checker executes natively asynchronous HTTP requests using the `httpx.AsyncClient` library. The system concurrently queries two distinct endpoints: 1. **VirusTotal API (v3):** The module queries the `/domains/{domain}` and `/ip_addresses/{ip}` endpoints. It parses the `last_analysis_stats` JSON object, summing the total number of security vendors that have flagged the domain to compute a `maliciousCount`. 2. **AbuseIPDB API (v2):** Focused strictly on the underlying hosting infrastructure, this module queries the `/check` endpoint to retrieve an `abuseConfidenceScore`, indicating the historical volume of abuse reports associated with the resolved IP address.

The outputs from these APIs are mathematically aggregated into an overarching `reputationScore` bounded between 0.0 and 1.0. This score provides the orchestrator with definitive, community-verified intelligence that directly heavily influences the `VerdictResult`, particularly when the machine learning model's confidence falls into the ambiguous "Suspicious" tier.

4.10 Semantic Evaluation in Threat Detection

While lexical URL analysis and real-time infrastructure queries form the empirical foundation of modern phishing detection, they remain inherently blind to the semantic payload of an attack. Contemporary cybercriminals increasingly leverage sophisticated social engineering narratives—often devoid of immediate malicious links—to build trust or induce panic before delivering the payload in subsequent communications (e.g., Business Email Compromise).

To counteract this, PhishGuard incorporates a dedicated Natural Language Processing (NLP) pipeline. This subsystem evaluates the unstructured text of emails, SMS messages, and web page content, mathematically quantifying the psychological manipulation tactics that are the hallmark of social engineering.

4.11 The spaCy NLP Pipeline Architecture

The core of the semantic analysis engine is built upon the `spaCy` framework. Selected for its production-grade performance, strict typing, and robust linguistic features, `spaCy` provides the foundational capabilities for high-speed tokenization, lemmatization, and pattern matching.

The `nlpAnalyzer.py` module encapsulates this functionality within the `NlpAnalyzer` class. To ensure deterministic execution and minimize memory overhead during serverless deployment, the system utilizes the lightweight, English-optimized `en_core_web_sm` model. Upon instantiation, the analyzer pre-loads a comprehensive taxonomy of social engineering indicators into memory, mapping them to specific `spacy.matcher.PhraseMatcher` instances.

4.11.1 Pipeline Initialization and Matchers

The `_setupMatchers` method constructs a highly optimized sequence of deterministic phrase matchers. By processing the text at the token level (rather than relying on brittle, raw string matching), the system successfully identifies indicators regardless of minor punctuation or casing deviations (utilizing the `attr="LOWER"` parameter).

The system initializes ten distinct `PhraseMatcher` pipelines: 1. `urgencyMatcher`: Detects artificial time pressure. 2. `threatMatcher`: Detects fear-inducing vocab-

ulary. 3. `authorityMatcher`: Detects impersonation of IT or administrative personnel. 4. `brandMatcher`: Detects the unauthorized usage of high-value corporate names. 5. `credentialMatcher`: Detects the explicit solicitation of sensitive data. 6. `actionMatcher`: Detects suspicious call-to-action phrasing. 7. `emotionalMatcher`: Detects emotional manipulation and appeal-to-greed phrases (e.g., "congratulations," "you have been selected"). 8. `monetaryMatcher`: Detects financial lure and money transfer requests (e.g., "wire transfer," "processing fee," "lottery winnings"). 9. `attachmentMatcher`: Detects suspicious file attachments and malware delivery attempts (e.g., ".exe," ".zip," "open the attachment"). 10. `socialProofMatcher`: Detects fake trust signals and social proof claims (e.g., "trusted by millions," "join millions of users").

4.12 Tactical Heuristics and Feature Extraction

Unlike the XGBoost model which operates on a continuous numerical feature vector, the NLP analyzer employs a deterministic, rule-based heuristic scoring mechanism. The system scans the tokenized `Doc` object for specific psychological triggers, categorizing them into predefined `PhishingTactic` enumerations.

4.12.1 Urgency and Threat Indicators

Phishing campaigns fundamentally rely on artificial time constraints to bypass a victim's rational scrutiny. The NLP analyzer implements strict phrase matching against a taxonomy of urgency indicators (e.g., "immediate action required," "expires today," "within 24 hours").

Simultaneously, the `_detectThreats` method scans for coercive vocabulary designed to induce panic. Phrases such as "account suspended," "unauthorized access," and "permanently deleted" are flagged. When the pipeline detects these phrases, it generates a `DetectedIndicator` object, assigning a high-confidence severity penalty (`severity=0.7` for urgency, `severity=0.8` for threats) to the text's overall risk profile.

4.12.2 Authority and Brand Impersonation

Attackers frequently exploit authority bias by masquerading as trusted institutions or internal administrative departments. - **Authority Impersonation:** The `_detectAuthority` method flags phrases commonly abused in Business Email Compromise (BEC) attacks, such as "IT support," "billing department," and "system administrator." - **Brand Impersonation:** The `_detectBrands` method checks the tokenized text against a predefined list of 17 high-value organizations: PayPal, Microsoft, Apple, Amazon, Google, Facebook, Instagram, Netflix, LinkedIn, Dropbox, Bank of America, Wells Fargo, Chase, IRS, FedEx, DHL, and USPS. Both the URL analyzer (via regex patterns) and the NLP analyzer (via phrase matching) detect these brands. Because brand mentions can occur in legitimate contexts, this specific indicator is assigned a moderate severity weight (`severity=0.5`), requiring corroboration from other malicious indicators to definitively flag the text as phishing.

4.12.3 Financial and Credential Solicitation

A primary objective of phishing is direct credential harvesting. The `_detectCredentialRequests` method identifies the explicit solicitation of sensitive data. Phrases demanding that a user "verify your account," "update your password," or "confirm your SSN" are flagged as critical risk indicators and assigned a severe penalty weight (`severity=0.85`).

4.12.4 Psychological Manipulation and Trust Signals

Modern phishing campaigns increasingly leverage sophisticated psychological manipulation tactics beyond simple urgency or threats.

Emotional Manipulation: The `_detectEmotionalManipulation` method identifies phrases designed to trigger emotional responses or appeal to greed. These include congratulatory messages ("you have been selected," "congratulations"), exclusivity claims ("once in a lifetime," "exclusive offer"), and fear of missing out tactics ("don't miss out," "limited spots"). Such indicators are assigned a `severity=0.5`.

Monetary Requests: The `_detectMonetaryRequests` method flags financial lure phrases and direct money transfer requests. This includes wire transfer requests ("wire transfer," "send money"), advance fee fraud ("processing fee," "advance fee"),

lottery and inheritance scams ("claim your prize," "unclaimed funds," "beneficiary"), and payment demands ("pay now," "outstanding balance"). These receive a high `severity=0.7` due to their direct financial threat.

Attachment Malware: The `_detectAttachmentMalware` method scans for references to suspicious file attachments. It detects dangerous file extensions (`.exe`, `.scr`, `.bat`, `.vbs`, `.js`, `.zip`, `.docm`, `.xlsm`) and attachment action phrases ("download the attachment," "open the attachment," "see attached"). With `severity=0.8` for suspicious extensions and `severity=0.65` for attachment actions, these are among the highest severity indicators.

Social Proof and Trust Signals: The `_detectSocialProof` method identifies fake trust signals designed to establish false credibility. These include popularity claims ("millions of users," "everyone is using"), verification assertions ("verified by,""certified," "award-winning"), and bandwagon appeals ("join millions,""most popular"). These indicators receive a lower `severity=0.4` but contribute to the sophistication bonus when combined with other tactics.

4.12.5 In-Text Link Manipulation

When analyzing raw text or emails (as opposed to a single URL), attackers often obfuscate malicious links within the body. The `_detectLinkManipulation` method utilizes regular expressions to extract all URLs embedded within the unstructured text. It then evaluates these extracted URLs against known malicious patterns: - **IP Obfuscation:** URLs utilizing raw IP addresses (e.g., `http://192.168.1.1/login`) instead of standard domain names trigger a high-severity indicator (`severity=0.75`). - **Suspicious TLDs:** Extracted URLs are parsed via `urllib.parse`. If the domain resolves to a known high-abuse generic Top-Level Domain (such as `.tk`, `.ml`, or `.ga`), it triggers a specific manipulation indicator (`severity=0.7`).

4.13 Scoring Orchestration and Output

The final output of the `NlpAnalyzer` is not a binary classification, but rather a highly structured `AnalysisResult` object. The `_calculateConfidence` method

mathematically synthesizes the array of discrete `DetectedIndicator` objects into a continuous confidence score bounded between 0.0 and 1.0.

4.13.1 The Scoring Algorithm

The confidence score is computed utilizing a two-tiered mathematical approach: 1. **Base Indicator Score:** The system calculates the average severity of all triggered indicators (`sum(ind.severity) / max(len(indicators), 1)`). 2. **Sophistication Bonus:** A sophisticated phishing attack rarely relies on a single tactic. An attacker might combine Brand Impersonation (mentioning PayPal), a Threat Warning ("account suspended"), and a Credential Request ("verify your password"). The scoring algorithm applies a cumulative multiplier (`min(len(tactics) * 0.1, 0.3)`) based on the diversity of the unique `PhishingTactic` enumerations detected.

If the final computed confidence exceeds 0.5, the `isPhishing` boolean is asserted.

4.13.2 Orchestrator Integration

The localized text score generated by the `NlpAnalyzer` is subsequently passed back to the central `AnalysisOrchestrator` (detailed in Chapter 3 (Developer Documentation)). For email and raw text inputs, this NLP-derived score serves as the primary predictive signal (weighted at 55%), supplemented by any extracted URL lexical features (25%) and infrastructure OSINT (20%).

Furthermore, the `_generateReasons` method maps the mathematical output into human-readable strings (e.g., "Uses fear tactics and account suspension threats"). These strings are serialized directly to the Next.js frontend, providing the end-user with immediate, transparent feedback regarding the psychological manipulation attempts detected within their input.

Chapter 5

Scoring Engine and Evaluation

5.1 The Orchestration of Intelligence

A primary challenge in engineering a multi-modal threat detection platform is the synthesis of disparate analytical signals into a unified, actionable verdict. PhishGuard achieves this through a sophisticated scoring and classification engine that seamlessly bridges the continuous probabilistic output of the XGBoost machine learning model with discrete, rule-based heuristic indicators.

This architecture ensures that the system is not only highly accurate but also highly transparent, providing Security Operations Center (SOC) analysts and end-users with explicit, human-readable explanations for every classification.

5.2 The ML Predictor Architecture

The core inference engine is housed within the `backend/ml/predictor.py` module. To guarantee high-throughput and sub-millisecond inference times during production deployment, the `PhishingPredictor` class is implemented utilizing a thread-safe Singleton design pattern.

5.2.1 Thread-Safe Initialization

During the FastAPI application startup sequence, the `PhishingPredictor` invokes its `__new__` method, locking the execution thread (`threading.Lock()`) to instantiate the XGBoost model exactly once. The model weights are loaded into memory from `models/phishingModel.json`, alongside the strict 21-dimensional feature

schema defined in `modelMetadata.json`. Once initialized, the model state becomes immutable, allowing the asynchronous web server to process concurrent inference requests across multiple threads without encountering race conditions or requiring continuous disk I/O operations.

5.2.2 Inference Execution

When a URL analysis request is routed to the predictor, the 21-dimensional `FeatureSet` is cast into a strongly typed `numpy.ndarray` (`dtype=np.float64`). The `predict()` method invokes the XGBoost `predict_proba` function, returning a continuous probability bounded between 0.0 and 1.0, representing the mathematical likelihood that the input belongs to the malicious class.

5.3 The Phishing Scorer Module

While the XGBoost probability serves as the primary ground truth for URL inputs, it operates as an opaque numerical value. To resolve the black-box interpretability crisis, the `backend/ml/scorer.py` module introduces a parallel heuristic scoring engine that reverse-engineers the threat context.

5.3.1 Explanatory Heuristics

Regardless of whether the XGBoost model is active, the `PhishingScorer` executes three discrete heuristic calculators to generate the human-readable `factors` array that will be presented to the user: 1. `calculateUrlStructureScore:` Applies deterministic penalty weights to structural anomalies. For example, a URL length exceeding 75 characters generates a scaling penalty (`min((urlLength - 75) / 100, 0.3)`), while the presence of raw IP addresses or Suspicious TLDs adds fixed penalties. 2. `calculateOsintScore:` Evaluates the infrastructure context. If `isNewlyRegistered` is true (domain age $<$ 30 days), a 0.35 penalty is recorded. Detection in third-party blacklists adds up to a 0.5 penalty. 3. `calculateFeatureScore:` Analyzes holistic combinations that indicate sophisticated evasion, such as a newly registered domain combined with the presence of suspicious keywords in the URI string (`score += 0.3`).

5.3.2 Graceful Heuristic Fallback

In the event of a catastrophic failure where the compiled XGBoost model cannot be loaded into memory, the `PhishingScorer` implements a graceful degradation protocol. Instead of halting the application, it relies entirely on the heuristic calculators, combining their raw outputs using the strict constants defined in the `ScoringWeights` dataclass: - **URL Structure:** 25% weight - **OSINT Derived:** 35% weight - **Feature Based:** 40% weight

This fallback mechanism ensures the platform remains operational and capable of identifying overt threats even during degraded system states.

5.4 Risk Level Determination

The final continuous score (whether derived from the XGBoost probability or the heuristic fallback) is mapped to discrete, actionable categories using the strict boundaries defined in the `RISK_THRESHOLDS` dictionary:

- **Safe (Score $<$ 0.30):** The domain exhibits normal structural characteristics, extensive DNS history, and clean reputation metrics. No action required.
- **Suspicious (Score $<$ 0.50):** The domain exhibits some characteristics commonly associated with phishing infrastructure, requiring user caution and verification. Minor anomalies may include newly registered domains or non-standard configurations.
- **Dangerous (Score $<$ 0.70):** Strong indications of malicious intent, such as obscured IPs, known bad TLDs, missing MX records on a young domain, or multiple risk indicators. The system recommends not interacting with the content.
- **Critical (Score $\geq$ 0.70):** Definitive malicious classification. The XGBoost model expresses high confidence in the threat, or the domain has been explicitly flagged by external threat intelligence APIs. Immediate blocking and reporting recommended.

5.4.1 Reason Prioritization

To prevent alert fatigue and ensure the user interface remains uncluttered, the `_prioritizeReasons` method deduplicates and sorts the aggregated heuristic factors. It employs a keyword-based sorting algorithm, ensuring that highly critical warnings (e.g., "malicious," "ip address," "newly registered") are prioritized at the top of the array, while lower-severity structural warnings are pushed down or truncated. The final output is strictly limited to the top 10 most relevant reasons.

5.5 Evaluation Methodology

The efficacy of the PhishGuard machine learning architecture was rigorously evaluated using a holdout methodology to prevent data leakage and ensure generalized performance on unseen data. The dataset, comprising diverse phishing and legitimate Uniform Resource Locators (URLs), was systematically split into training and testing partitions.

The primary training corpus consisted of 23,374 samples (`trainSamples: 23,374`). To accurately assess the model's predictive capability in a simulated real-world environment, a strictly held-out test set comprising 5,009 samples (`testSamples: 5,009`) was established. This test set maintained an near-perfect class balance, containing 2,505 legitimate instances and 2,504 phishing instances, ensuring that evaluation metrics were not artificially skewed by class imbalance.

5.6 Hyperparameter Optimization

Prior to final evaluation, the XGBoost classifier underwent extensive hyperparameter tuning utilizing the Optuna optimization framework. The optimization process executed 50 distinct trials (`nTrials: 50`) employing 5-fold cross-validation (`nCvFolds: 5`) to iteratively search the hyperparameter space for the configuration that maximized the Area Under the Receiver Operating Characteristic Curve (ROC AUC).

The optimal configuration was identified at Trial 43 (`bestTrialNumber: 43`), yielding a cross-validated AUC of 0.9943. The resulting best hyperparameters significantly constrained the model's complexity to prevent overfitting while maintaining high predictive capacity: - **Maximum Depth (`max_depth`):** 7 - **Number**

of Estimators (`n_estimators`): 700 - **Learning Rate** (`learning_rate`): ~0.177 - **Subsample Ratio** (`subsample`): ~0.945 - **Column Subsample by Tree** (`colsample_bytree`): ~0.873 - **Gamma** (`gamma`): ~0.198

5.7 Empirical Performance Metrics

The fully optimized XGBoost model, utilizing the complete 21-dimensional feature vector (17 URL structural features and 4 OSINT-derived features), demonstrated exceptional performance on the held-out test set of 5,009 samples.

As summarized in Table 5.1, the model achieved an aggregate **Accuracy of 96.45%** (`0.96446`), indicating robust overall correctness. However, in the context of threat detection, precision and recall offer more nuanced insights into operational viability.

Table 5.1: PhishGuard XGBoost Model Performance Metrics (Test Set: N=5,009)

Metric	Value (%)
Accuracy	96.45%
Precision	97.86%
Recall (Sensitivity)	94.97%
F1-Score	96.39%
ROC AUC	99.41%
PR AUC	99.48%

The system achieved a **Precision of 97.86%** (`0.97860`), demonstrating a low false-positive rate—a critical metric for minimizing alert fatigue among end-users. Correspondingly, the **Recall reached 94.97%** (`0.94968`), signifying the model's strong capability to successfully identify true phishing threats within the dataset. The harmonic mean of these two metrics resulted in an **F1-Score of 96.39%** (`0.96392`).

Furthermore, the model exhibited outstanding discriminative ability across various classification thresholds, achieving a **ROC AUC of 99.41%** (`0.99408`) and a Precision-Recall Area Under Curve (**PR AUC**) of **99.48%** (`0.99479`).

5.7.1 Error Analysis and Confusion Matrix

A granular analysis of the classification errors on the test set reveals the model's operational tendencies. Table 5.2 presents the exact distribution of the 5,009 test predictions.

Table 5.2: Confusion Matrix for the Holdout Test Set

		Predicted Class	
		Legitimate (0)	**Phishing (1)**
Actual	**Legitimate (0)**	2,453 (TN)	52 (FP)
	Phishing (1)	126 (FN)	2,378 (TP)

The disparity between False Positives (52) and False Negatives (126) underscores the model's high-precision orientation. While the system is highly reliable when it issues a warning, a small subset of sophisticated phishing URLs successfully evaded the structural and OSINT feature checks.

5.8 Feature Explainability and SHAP Analysis

To demystify the XGBoost model's decision-making process and move beyond "black-box" predictions, SHapley Additive exPlanations (SHAP) were employed. The SHAP `TreeExplainer` was utilized to compute the marginal contribution of each of the 21 features across the test set.

The SHAP analysis (`shapAnalysis.py`) revealed a distinct hierarchy in feature importance. The presence of HTTPS (`isHttps`) emerged as the most influential single feature (accounting for roughly 33.4% of the global importance), followed closely by the validity of the domain's DNS configuration (`hasValidDns`, 12.5%). Structural anomalies, such as `specialCharCount` (8.5%) and `pathDepth` (7.4%), also demonstrated significant predictive power.

5.8.1 OSINT Ablation Study

A critical objective of the PhishGuard architecture was to evaluate the supplemental value of Open-Source Intelligence (OSINT) data when combined with traditional URL lexical analysis. To quantify this, an ablation study was conducted.

The global SHAP contribution analysis (`ablation_report.json`) calculated the mean absolute SHAP values for the two distinct feature subsets. The analysis determined that the 17 URL structural features account for **85.64%** of the model's total predictive influence. The 4 dynamic OSINT features (`usesCdn`, `dnsRecordCount`, `hasValidDns`, `hasValidMx`) contributed the remaining **14.36%**.

Interestingly, when the model was retrained exclusively on the 17 URL features (excluding all OSINT data), the baseline performance metrics exhibited a nominal, fractional increase (Accuracy shifted by ~+0.53%). This phenomenon suggests that within this specific, balanced dataset, the structural URL anomalies were sufficiently profound to classify the targets independently. The inclusion of OSINT features, while contributing 14.36% to the model's SHAP explanations and providing critical human-readable context for the heuristic Scorer module, acted slightly as a regularizer in the pure ML context, mitigating over-reliance on purely lexical characteristics.

Chapter 6

Conclusion and Discussion

6.1 Interpretation of Results

6.1.1 The Paradox of Threat Intelligence: Explainability versus Predictive Power

The empirical evaluation of the PhishGuard architecture (detailed in Chapter 5) yielded a profound observation regarding the integration of Open-Source Intelligence (OSINT) within machine learning (ML) models. The ablation study demonstrated that the 4 dynamic OSINT features (`usesCdn`, `dnsRecordCount`, `hasValidDns`, `hasValidMx`) contributed exactly 14.36% to the model's SHapley Additive exPlanations (SHAP) global feature importance. However, retraining the model exclusively on the 17 static structural URL features resulted in a nominal, fractional increase in overall accuracy (approximately +0.53%).

This phenomenon initially appears paradoxical: how can the inclusion of high-fidelity threat intelligence (such as domain age and blacklist presence) marginally decrease the raw predictive performance of the classifier?

The underlying mechanism driving this behavior is the "Time-to-Detect (TTD) Lag" inherent in reactive cybersecurity infrastructure. Phishing campaigns are increasingly characterized by their ephemeral nature; malicious domains are registered programmatically, utilized for hours or days, and rapidly discarded. When a zero-day phishing URL is subjected to analysis, its structural features (e.g., extensive path depth, suspicious Top-Level Domains (TLDs), high digit ratios, and explicit port numbers) are immediately apparent to the lexical analyzer. Conversely, because the

domain was instantiated moments prior, it possesses no historical reputation data. It exists as a "clean" entity within the VirusTotal and AbuseIPDB databases.

When the XGBoost classifier processes a vector containing highly anomalous structural features alongside perfectly benign OSINT metrics (due to the TLD lag), the model encounters conflicting signals. This dissonance marginally depresses the prediction confidence, occasionally resulting in false negatives for sophisticated zero-day threats.

However, removing OSINT entirely from the architecture is not viable for a production-grade system. While structural features drive raw accuracy, they operate as a "black box." The end-user cannot be simply presented with an abstract probability score. The OSINT data serves a critical, non-mathematical function: heuristic explainability. The orchestrator synthesizes the OSINT metrics to generate human-readable justifications (e.g., "Domain registered within the last 7 days" or "WHOIS privacy protection enabled"). Therefore, the architectural trade-off is deliberate; the system sacrifices a fraction of a percent in theoretical accuracy to provide the user with actionable, comprehensible intelligence regarding the threat vector.

6.1.2 Architectural Trade-offs: Latency and Concurrency

A core engineering challenge in developing the PhishGuard backend was reconciling the necessity for comprehensive, multi-source analysis with the imperative for low-latency, synchronous Application Programming Interface (API) responses.

The integration of external OSINT services—specifically synchronous `dnspython` and `whois` libraries—introduced significant I/O-bound latency. Sequential execution of these queries would predictably violate acceptable user experience (UX) thresholds, potentially taking upwards of 30 seconds to resolve a single URL if a DNS server was unresponsive or throttling connections.

To mitigate this, the `backend/api/orchestrator.py` module was engineered utilizing Python's asynchronous I/O paradigm (`asyncio.gather`), dispatching the NLP, WHOIS, DNS, and Reputation analysis tasks concurrently. Crucially, this execution block is encapsulated within a strict `15.0` second global timeout (`asyncio.wait_for`).

This design represents a conscious prioritization of system availability over analytical completeness. If a downstream threat intelligence provider experiences an outage, or if a malicious domain's authoritative name server deliberately tarpits incoming requests (a common anti-analysis tactic), the orchestrator gracefully intercepts the `TimeoutError`. The system drops the incomplete OSINT payload and seamlessly delegates the final verdict entirely to the XGBoost classifier and the NLP module. This ensures that the platform remains highly available and consistently responsive, preventing a degraded external dependency from cascading into a total denial of service for the PhishGuard platform.

6.1.3 Graceful Degradation and Deterministic Fallbacks

Resilience within the ML pipeline was a primary design objective. Machine learning models, particularly those deployed in constrained or containerized environments, are susceptible to memory exhaustion, serialization corruption, or filesystem permission errors during instantiation.

The `PhishingPredictor` class addresses this vulnerability by implementing a robust state management system (`self._isLoaded`). If the XGBoost model binary (`phishingModel.json`) fails to load during the application lifecycle, the `PhishingScorer` module detects this degraded state and automatically reroutes the feature vector to a deterministic, heuristic fallback algorithm.

This fallback mechanism applies predefined scalar weights to the independently calculated sub-scores: 25% for URL structure, 35% for OSINT anomalies, and 40% for combined feature indicators. While this static heuristic lacks the non-linear relationship modeling capabilities of the trained XGBoost ensemble, it guarantees that the core phishing detection pipeline remains operational. The system continues to identify overt threats (e.g., IP-based URLs lacking HTTPS with suspicious keywords) even when the predictive engine is offline, demonstrating a sophisticated defense-in-depth approach to software engineering.

6.2 Threat Modeling and Limitations

While the empirical results indicate a highly effective detection capability (96.45% accuracy), a rigorous academic evaluation requires a critical analysis of

the system's inherent limitations and the potential vectors adversaries might utilize to evade detection.

6.2.1 Linguistic Constraints and Homograph Attacks

The NLP analyzer (`backend/analyzer/nlpAnalyzer.py`) relies heavily on the `spaCy` library, specifically utilizing the `en_core_web_sm` model. Consequently, the `PhraseMatcher` pipelines and urgency detection heuristics are explicitly optimized for English-language content. Phishing emails constructed in alternate languages, or utilizing complex multilingual idioms, will largely bypass the semantic analysis, relying entirely on the URL structural heuristics.

Furthermore, advanced Internationalized Domain Name (IDN) homograph attacks pose a significant challenge. An adversary might register a domain such as `paypal.com` (utilizing a Cyrillic 'a' rather than a Latin 'a'). To the human eye, the URL appears legitimate, and to the basic structural analyzer, it lacks overt obfuscation characteristics (such as excessive hyphens or deep paths). While modern browsers often mitigate this by enforcing Punycode display, the underlying ML model must be trained specifically on Punycode representations to effectively classify these sophisticated spoofing attempts.

6.2.2 Image-Based Exploitation and OCR Deficiencies

A prominent evasion tactic employed in contemporary phishing campaigns is the total omission of parsable text within the email body. Attackers frequently embed the malicious message, branding, and explicit instructions within a single, hyperlinked image.

The current PhishGuard architecture processes raw string payloads and URL vectors. Because the system lacks an Optical Character Recognition (OCR) module or computer vision capabilities, it is entirely blind to the semantic content of image-based emails. The system would only evaluate the destination URL embedded within the `href` attribute, bypassing the sophisticated urgency and credential-harvesting NLP pipelines entirely.

6.2.3 Compromised Legitimate Infrastructure

The most profound limitation of any reputation-based OSINT or ML system is the exploitation of compromised legitimate infrastructure. If an adversary breaches a vulnerable, decade-old WordPress installation on a highly reputable domain (e.g., `https://university-biology-dept.edu/wp-content/uploads/secure-login`), the resulting phishing URL inherits the pristine OSINT characteristics of the host.

In this scenario: 1. The domain age is massive (lowering the risk score). 2. The DNS and MX records are perfectly valid (lowering the risk score). 3. The domain has zero presence on VirusTotal or AbuseIPDB (lowering the risk score). 4. The SSL certificate (HTTPS) is valid and signed by a trusted authority.

The PhishGuard XGBoost model must rely exclusively on the anomalous structural depth (`/wp-content/uploads/secure-login`) and the detection of credential-harvesting keywords within the path to flag the anomaly. If the attacker obfuscates the path structure, the reliance on OSINT metrics becomes a significant vulnerability, reinforcing the necessity for continuous, dynamic content analysis beyond static features.

6.3 Comparison with State-of-the-Art Solutions

To fully contextualize the achievements of the PhishGuard architecture, its empirical results must be juxtaposed against existing paradigms in phishing detection, specifically traditional blacklist aggregation (e.g., Google Safe Browsing, PhishTank) and contemporary single-domain machine learning models.

Traditional blacklist mechanisms possess a fundamental flaw: they are entirely reactive. As established in the background literature (Chapter 2), the median lifespan of a phishing domain has contracted to mere hours. By the time a malicious URL is verified, categorized, and propagated through global blacklist CDNs, the campaign has often already concluded. PhishGuard's hybrid approach circumvents this limitation. While it queries reputation APIs (VirusTotal, AbuseIPDB) to catch known offenders immediately, its primary reliance on XGBoost evaluating structural heuristics (which yielded an accuracy of 96.45%) allows for the proactive detection of "zero-day" phishing infrastructure before any human analyst has reviewed it.

Furthermore, compared to pure Natural Language Processing (NLP) models that scan email bodies, PhishGuard introduces structural redundancy. Pure NLP models are highly susceptible to adversarial perturbations—attackers frequently inject invisible HTML text, utilize zero-width spaces, or employ synonym replacement to bypass Bayesian filters or Transformer models. By pairing an NLP semantic scanner with an independent URL/OSINT XGBoost classifier, PhishGuard forces the adversary to successfully obfuscate *both* the linguistic payload and the structural infrastructure simultaneously, exponentially increasing the cost and complexity of the attack.

6.4 The Operational Cost of Classification Errors

In the deployment of cybersecurity classification systems, the raw accuracy metric (96.45%) is less operationally significant than the distribution of its errors. The confusion matrix generated on the 5,009-sample holdout set revealed 52 False Positives (FP) and 126 False Negatives (FN).

This disparity highlights a deliberate, conservative thresholding strategy within the ML pipeline. In a corporate or enterprise environment, False Positives carry a heavy operational cost; legitimately blocking a crucial vendor portal or internal authentication gateway generates "alert fatigue" and overwhelms IT support channels. Users rapidly lose trust in a security system that consistently cries wolf. Therefore, the model's high Precision (97.86%) ensures that when PhishGuard flags a URL as "Dangerous" or "Critical," the probability of it being a genuine threat is overwhelming.

Conversely, the 126 False Negatives represent sophisticated evasion. These are phishing URLs that successfully mimicked benign structural patterns and lacked negative OSINT reputation. Addressing this gap without dramatically increasing the False Positive rate represents the primary frontier for future algorithmic refinement. This operational reality dictates that ML-based detection systems must not be deployed in a vacuum; they must be layered alongside multi-factor authentication (MFA) and zero-trust network architectures to mitigate the inevitable percentage of advanced threats that evade algorithmic detection.

6.5 The Shifting Paradigm of HTTPS in Phishing

The SHAP feature importance analysis (Figure 4.1) highlighted `isHttps` as the single most influential predictive feature, contributing approximately 33.4% of the model's global decision-making weight. From an academic perspective, the historical context of this metric is deeply fascinating and warrants critical discussion.

A decade ago, the presence of a valid SSL/TLS certificate (HTTPS) was a near-guarantee of a domain's legitimacy. The financial cost and identity verification required to obtain a certificate acted as a natural deterrent to phishers. However, the advent of automated, free Certificate Authorities (e.g., Let's Encrypt, ZeroSSL) has completely inverted this paradigm. Today, the vast majority of phishing domains utilize HTTPS to exploit the psychological trust users place in the browser's "padlock" icon.

The fact that `isHttps` remains highly predictive in the PhishGuard dataset indicates that while sophisticated phishers utilize SSL, a massive volume of low-effort, bulk phishing campaigns still operate over plaintext HTTP via compromised IoT devices, legacy servers, or free hosting providers. However, the predictive power of `isHttps` is guaranteed to decay over time. As HTTPS adoption approaches 100% across both legitimate and malicious actors, the variance in this feature will approach zero, forcing future iterations of the model to rely more heavily on dynamic metrics like `hasValidDns`, `dnsRecordCount`, and lexical path depth.

6.6 Explainable AI (XAI) and Security Awareness

A significant architectural triumph of the PhishGuard platform is its commitment to Explainable AI (XAI). Traditional cybersecurity tools operate as opaque "black boxes," blocking content without providing justification. This approach is detrimental to long-term security posture because it fails to educate the end-user.

By utilizing the Next.js frontend to dynamically render the `ScoreComponent` data (aggregated by the `PhishingScorer`), PhishGuard transitions from a passive filter to an active pedagogical tool. When a user inputs a URL and receives a "Dangerous" verdict, they are simultaneously presented with the exact reasons driving that classification (e.g., "Uses IP address instead of domain name" or "Domain registered within the last 7 days").

This transparency achieves two critical objectives: 1. **Calibrated Trust:** Users are more likely to heed a warning when the system rationally justifies its conclusion, reducing the likelihood of users intentionally bypassing security controls. 2. **Behavioral Conditioning:** By consistently exposing users to the structural hallmarks of phishing (suspicious TLDs, missing MX records, homograph patterns), the system passively trains the user's inherent cognitive heuristics, improving the human firewall.

6.7 Concept Drift and Future-Proofing

Machine learning models deployed in adversarial environments suffer from a phenomenon known as "concept drift." The statistical properties of the target variable change over time as adversaries actively adapt their tactics to bypass detection mechanisms.

The XGBoost model instantiated in this thesis achieved 96.45% accuracy against the current distribution of phishing attacks. However, as threat actors realize that deep paths and excessive subdomains are heavily penalized by lexical analyzers, they will pivot. They will increasingly utilize URL shorteners (e.g., `bit.ly`, `t.co`), open redirects, and decentralized web hosting (e.g., IPFS) to flatten the structural topology of their malicious URLs.

Therefore, the current implementation of PhishGuard, while highly effective, is a static snapshot. For the system to maintain its efficacy in a production environment, the architecture must evolve to incorporate continuous retraining pipelines. This would require an automated ingestion engine that continuously scrapes newly verified phishing URLs from live feeds (e.g., PhishTank, OpenPhish), re-extracts the 21-dimensional feature vectors, and dynamically updates the XGBoost ensemble weights. Furthermore, the reliance on statically compiled lists of "suspicious keywords" and "legitimate brand names" within `urlAnalyzer.py` must eventually be replaced by dynamic clustering algorithms to autonomously identify emerging brand impersonation trends.

6.8 Conclusion

6.8.1 Summary of Contributions

The exponential proliferation of phishing attacks—characterized by automated, low-cost domain generation and sophisticated social engineering—has fundamentally outpaced the defensive capabilities of traditional reactive security paradigms. This thesis proposed, designed, and implemented a novel, proactive cybersecurity architecture designated as PhishGuard. The primary objective was to engineer a real-time, hybrid detection system capable of mitigating zero-day phishing infrastructure while simultaneously providing human-readable heuristic explainability to educate end-users.

The PhishGuard architecture represents a synthesis of machine learning (ML), natural language processing (NLP), and dynamic Open-Source Intelligence (OSINT) aggregation. By migrating away from monolithic blacklist reliance and towards a predictive, feature-engineered ensemble model, the system successfully addresses the "Time-to-Detect (TTD) Lag" inherent in modern threat intelligence.

The empirical evaluation of the core classification engine yielded exceptional results. Utilizing a highly optimized XGBoost classifier trained on a 21-dimensional feature space (comprising 17 structural URL anomalies and 4 dynamic OSINT heuristics), the model achieved a comprehensive accuracy of 96.45% on a strictly held-out, balanced dataset of 5,009 samples. More critically for enterprise deployment, the architecture was explicitly tuned for high precision, achieving a rate of 97.86%. This deliberate calibration significantly mitigates the operational friction and "alert fatigue" typically associated with overly aggressive algorithmic detection systems.

Furthermore, the architectural implementation via the FastAPI asynchronous orchestrator demonstrated that comprehensive, multi-source threat analysis can be executed synchronously within strict Service Level Agreements (SLAs). By enforcing a 15.0-second concurrency window on the potentially volatile WHOIS, DNS, and Reputation API lookups, the system guarantees high availability and graceful degradation to standalone ML and NLP heuristics when external dependencies fail.

6.9 Fulfillment of Research Objectives

This research successfully fulfilled its predefined objectives through the following critical implementations:

1. **Proactive Zero-Day Detection:** By mathematically analyzing lexical patterns (e.g., path depth, high digit ratios, suspicious top-level domains) rather than relying solely on historical reputation, the system demonstrated the capacity to classify novel phishing infrastructure prior to its categorization by global threat intelligence networks.

2. **Multimodal Threat Analysis:** The system is not strictly bound to URL evaluation. The integration of the `spaCy`-driven `NlpAnalyzer` allows the platform to parse semantic intent, identifying urgency markers and credential-harvesting nomenclature within email bodies and arbitrary text payloads.

3. **Explainable AI (XAI) Integration:** A persistent flaw in modern cybersecurity tooling is the deployment of opaque, "black-box" decision engines. The PhishGuard `PhishingScorer` module mathematically decomposes the XGBoost probability score and the NLP confidence metrics into discrete, human-readable rationales (e.g., "Domain registered within the last 7 days"). This transparent feedback loop acts as a pedagogical mechanism, actively calibrating user trust and reinforcing secure behavioral conditioning.

6.10 Future Work: Algorithmic Enhancements

While the PhishGuard architecture achieved its core objectives, the adversarial nature of cybersecurity necessitates continuous evolution. Several avenues exist for significant algorithmic enhancement in future iterations of this research.

6.10.1 Dynamic Online Learning and Concept Drift Mitigation

The current XGBoost classifier operates as a static, pre-trained ensemble. As threat actors inevitably pivot away from heavily penalized structural anomalies (e.g.,

migrating toward decentralized IPFS hosting or utilizing obfuscated URL shorteners), the static model will experience concept drift, and its predictive efficacy will decay. Future iterations must implement an automated, continuous online learning pipeline. This architecture would dynamically ingest newly verified phishing URLs, asynchronously re-extract the 21-dimensional feature vectors, and update the decision tree weights without requiring manual offline retraining and deployment cycles.

6.10.2 Transitioning from NLP Heuristics to Large Language Models (LLMs)

The `NlpAnalyzer` module currently relies on predefined `spaCy PhraseMatcher` pipelines optimized exclusively for English-language threat detection. This static, rule-based approach is vulnerable to synonym replacement, grammatical obfuscation, and multilingual attacks. Future research should replace the static NLP heuristics with a lightweight, fine-tuned Transformer model (such as RoBERTa or a quantized generative LLM). A transformer-based architecture would capture the deeper semantic context of an email payload—detecting sophisticated spear-phishing and Business Email Compromise (BEC) attempts that do not rely on overt, hardcoded "urgent" keywords, while natively supporting cross-lingual threat detection.

6.11 Future Work: Architectural Scaling and Expansion

To transition the PhishGuard prototype into an enterprise-grade, globally distributed threat intelligence platform, substantial architectural restructuring is required.

6.11.1 Event-Driven Microservices

The current implementation utilizes a centralized FastAPI orchestrator managing asynchronous tasks via `asyncio`. While highly efficient for a prototype, horizontal scaling under massive, concurrent enterprise traffic requires an event-driven architecture. Decomposing the monolith into discrete microservices (e.g., an OSINT ingestion service, an ML inference service, an NLP parsing service) coordinated via

a distributed message broker (such as Apache Kafka or RabbitMQ) would allow the system to ingest, queue, and process millions of URLs per minute while independently scaling the most computationally expensive nodes.

6.11.2 Optical Character Recognition (OCR) Integration

As highlighted in the discussion on evasion techniques, a critical blind spot in the current architecture is the inability to analyze image-based phishing payloads. Attackers frequently embed their fraudulent branding and semantic instructions within a single PNG or JPEG file, utilizing the email text merely as a delivery vehicle for an embedded `href` link. To counter this vector, future iterations must integrate a robust Computer Vision and OCR pipeline (e.g., Tesseract or a cloud-based Vision API). This would allow the system to extract the text rendered within the image, subjecting it to the same rigorous NLP semantic analysis currently applied to plaintext emails, thereby eliminating a primary evasion tactic utilized by modern threat actors.

6.11.3 Concluding Remarks

The development of the PhishGuard platform confirms that a hybrid, multi-layered approach to phishing detection—combining the raw predictive power of machine learning, the semantic understanding of natural language processing, and the historical context of open-source intelligence—significantly outperforms isolated, singular detection methodologies.

By prioritizing high-precision classification and transparent heuristic explainability, this research bridges the critical gap between theoretical algorithmic performance and operational, user-centric cybersecurity. As the threat landscape continues to evolve in complexity and scale, the principles of asynchronous orchestration, structural redundancy, and explainable AI demonstrated in this thesis will remain foundational to the next generation of proactive defense mechanisms.

End of Thesis Document

Project Links

Table 6.1: Project Links and Resources

Resource	**Link**
GitHub Repository	`https://github.com/ishaq2321/phishing-detection-osint`
Issue Tracker	`https://github.com/ishaq2321/phishing-detection-osint/issues`
Contact Email	`pxprgk@inf.elte.hu`
Deployed Application	
Frontend (Vercel)	`https://project-4soy4.vercel.app`
Backend API (Render)	`https://phishguard-api-upl2.onrender.com`
API Documentation	`https://phishguard-api-upl2.onrender.com/docs`

Note: The complete source code, setup instructions, and documentation are available in the GitHub repository. For bug reports or feature requests, please use the GitHub Issues tracker.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Md. Easin Arafat, for his invaluable guidance and support throughout this research. I also extend my deepest appreciation to my family and friends for their unwavering encouragement during my studies.

Bibliography

[1] Anti-Phishing Working Group (APWG). *Phishing Activity Trends Report, Q4 2023*. Tech. rep. 2023. URL: https://apwg.org/trendsreports/.

[2] Federal Bureau of Investigation (FBI). *Internet Crime Report 2022*. Tech. rep. Internet Crime Complaint Center (IC3), 2023.

[3] G Ramesh et al. "Phishing URL Detection: A Machine Learning and Web Mining-based Approach". In: *International Journal of Computer Applications* 123.13 (2015), pp. 46–50.

[4] Webroot. *2023 Webroot BrightCloud Threat Report*. Tech. rep. 2023.

[5] J Hong. "The State of Phishing Attacks". In: *Communications of the ACM* 55.1 (2012), pp. 74–81.

[6] M Jakobsson and S Myers. *Phishing and Countermeasures: Understanding the Increasing Problem of Electronic Identity Theft*. Wiley, 2006.

[7] PhishLabs. *2023 Phishing Trends and Intelligence Report*. Tech. rep. 2023. URL: https://www.phishlabs.com/.

[8] Google. *Google Safe Browsing*. URL: https://safebrowsing.google.com/.

[9] *PhishTank*. URL: https://phishtank.org/.

[10] *OpenPhish*. URL: https://openphish.com/.

[11] APWG. *eCrime Exchange (eCX)*. URL: https://apwg.org/ecx/.

[12] G Ramesh et al. "Phishing URL Detection: A Machine Learning and Web Mining-based Approach". In: *Int. J. Computer Applications* 123.13 (2015).

[13] R M Mohammad et al. "Predicting phishing websites based on self-structuring neural network". In: *Neural Computing and Applications* 25 (2014), pp. 443–458.

[14] O K Sahingoz et al. “Machine learning based phishing detection from URLs”. In: *Expert Systems with Applications* 117 (2019), pp. 345–357.

[15] K L Chiew et al. “A new hybrid ensemble feature selection framework for machine learning-based phishing detection system”. In: *Information Sciences* 484 (2019), pp. 153–166.

[16] R S Rao and A R Pais. “Detection of phishing websites using an efficient feature-based machine learning framework”. In: *Neural Computing and Applications* 31 (2019), pp. 3851–3873.

[17] P Yang et al. “MTD: A Multi-Task Deep Learning Framework to Predict Drug-Target Interactions”. In: *Proc. IJCAI*. 2019.

[18] M Somesha et al. “Classification of Phishing Websites using XGBoost and Deep Neural Networks”. In: *Proc. ICCIDS*. 2020.

[19] E Buber et al. “Detecting phishing attacks from URL by using NLP techniques”. In: *Proc. IDAP*. 2021.

[20] M Korkmaz et al. “Phishing web sites detection using hybrid model based on deep belief network and autoencoder”. In: *Multimedia Tools and Applications* 81 (2022), pp. 24159–24178.

[21] A S Aljofey et al. “An effective phishing detection model based on character level convolutional neural network from URL”. In: *Electronics* 9.9 (2020).

[22] Y Li et al. “A convolutional neural network-based approach for phishing website detection”. In: *Proc. Trustcom*. 2019.

[23] A Hiransha et al. “PhishDef: URL-Based Phishing Detection Using BERT”. In: *Proc. ICTCS*. 2022.

[24] A K Jain and B B Gupta. “Towards detection of phishing websites on client-side using machine learning based approach”. In: *Telecommunication Systems* 68 (2018), pp. 687–700.

[25] R Dhamija et al. “Why phishing works”. In: *Proc. CHI*. 2006.

[26] S M Lundberg and S-I Lee. “A unified approach to interpreting model predictions”. In: *Proc. NeurIPS*. 2017.

[27] M T Ribeiro et al. “"Why should I trust you?": Explaining the predictions of any classifier”. In: *Proc. KDD*. 2016.

[28] L Daigle. *WHOIS Protocol Specification*. Tech. rep. RFC 3912, 2004. URL: https://www.rfc-editor.org/rfc/rfc3912.

List of Figures