**PhishGuard AI Sentinel ++**

**Documentation**

**Team Name / Members**: AI BOTS / K S SREEHARI, ADITYA M ANEGUNDI   
**Hackathon**: CloudSEK AI in Cybersecurity Hackathon  
**Date**: April 25, 2025

**1. Solution Overview and Objectives**

**1.1. Problem Statement**

Phishing and social engineering are among the most prevalent cyber threats, contributing to a significant portion of global security breaches. Attackers continuously refine their methods, utilizing advanced techniques such as AI-generated lures, highly personalized content, and convincing brand impersonation. These sophisticated threats frequently evade traditional signature-based filters and static rule sets. Security analysts are overwhelmed by ambiguous alerts, leading to alert fatigue, while end-users struggle to identify malicious communications. This results in credential theft, malware infections, data exfiltration, and substantial financial losses. There is an urgent need for tools that provide deeper contextual analysis and deliver clear, actionable intelligence.

**1.2. Proposed Solution: PhishGuard AI Sentinel ++**

PhishGuard AI Sentinel ++ is an advanced analysis tool designed to overcome the limitations of conventional phishing defenses. It harnesses the power of Google's Gemini Large Language Model (LLM) to conduct in-depth contextual analysis of email and message content.

Unlike traditional systems that rely solely on known malicious indicators, PhishGuard AI evaluates text for underlying intent (e.g., credential harvesting, malware delivery), psychological manipulation tactics (e.g., urgency, authority, scarcity), linguistic anomalies, and signs of impersonation. When raw email source data is available, the tool enhances its analysis by examining header information (e.g., sender authenticity via SPF/DKIM/DMARC hints, sender/From mismatches) and assessing the risk of attachment filenames.

A standout feature is its **explainable AI** capability. Instead of merely assigning a risk score, PhishGuard AI uses Gemini to produce concise, human-readable explanations that detail *why* a message is deemed risky, referencing specific identified factors. The system has been carefully tuned to minimize false positives on legitimate business communications, striking a balance between robust security and practical usability.

**1.3. Key Objectives**

The PhishGuard AI Sentinel ++ prototype achieves the following primary objectives:

* **Detect Diverse Threats**: Leverage AI to identify sophisticated phishing, social engineering, and potential malware indicators based on content and context.
* **Flexible Input**: Support analysis of messages via direct manual input (Sender/Subject/Body) or by parsing full raw email source (Headers + Body + Attachments).
* **Contextual Enrichment**: Incorporate header analysis (when available) and attachment filename checks into the risk assessment.
* **Simulated URL Analysis**: Evaluate URL risks using blocklists and AI-driven pattern analysis, optimized for known good domains.
* **Nuanced Risk Scoring**: Compute a weighted risk score (0-100) based on multiple indicators, with weights adjusted to reduce false positives.
* **Explainable Alerts**: Deliver clear, AI-generated natural language explanations that justify the risk score and recommend appropriate actions.
* **Functional Prototype**: Provide a working Streamlit application showcasing the core analysis pipeline and features within the hackathon timeframe.

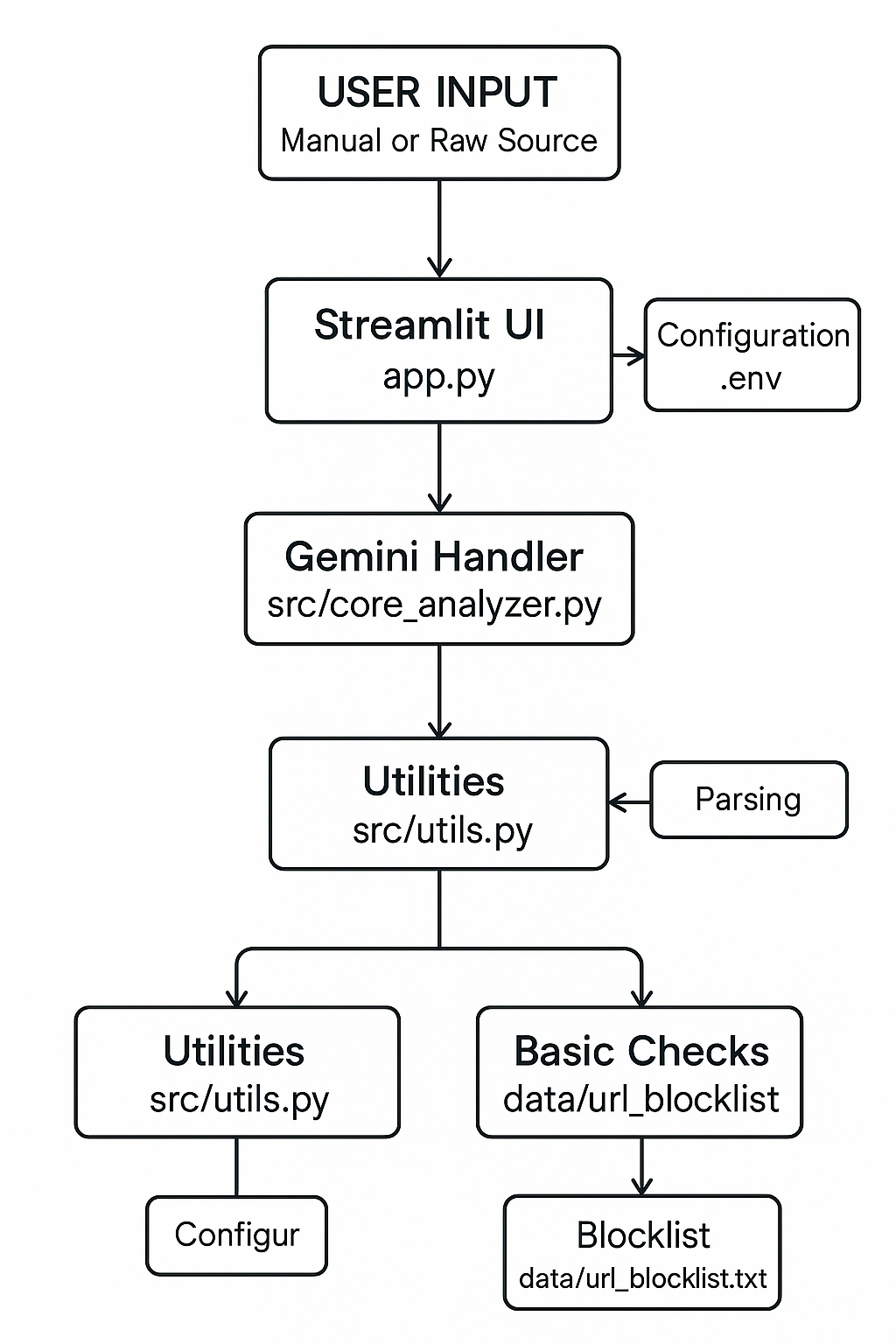
**2. Tech Stack and Architecture**

**2.1. Technology Stack**

* **Programming Language**: Python (v3.9+)
* **AI Service**: Google Gemini API (Model: gemini-1.5-flash)
* **AI SDK**: google-generativeai Python library
* **Web Framework**: Streamlit (for interactive UI)
* **Core Libraries**:
  + python-dotenv: Secure management of API keys via .env file.
  + regex: Advanced regular expression matching for URL extraction.
  + pandas: Data manipulation and display (used for presenting parsed headers).
  + email (Standard Library): Parsing raw email source files (RFC 5322 format).
* **Environment Management**: venv (Python virtual environments)

**2.2. Architecture Diagram**

[Architecture Diagram Description: A diagram illustrating user input (Manual or Raw Source) flowing to the Streamlit UI (app.py). The UI interacts with the Core Analyzer (src.core\_analyzer.py), which orchestrates calls to: the Gemini Handler (src.gemini\_handler.py) for interaction with the Google Gemini API; the Utilities module (src.utils.py) for parsing; and the Basic Checks module (src.basic\_checks.py) utilizing the URL blocklist. Configuration data, such as the API key (.env), feeds into the Gemini Handler, while the blocklist (data/url\_blocklist.txt) supports Basic Checks.]



**2.3. Component Description**

* app.py **(Streamlit UI)**:  
  Provides a web-based user interface using Streamlit. It handles user input through text boxes and radio buttons, displays sample data loading buttons, initiates analysis by calling core\_analyzer.analyze, presents results (risk score, AI explanation, contributing factors, and detailed breakdown), manages session history, and supports the "Export Summary" functionality.
* src/core\_analyzer.py **(Orchestration & Scoring)**:  
  Serves as the central logic unit. It receives input data and analysis mode from app.py, calls utils.py for parsing, basic\_checks.py for blocklist lookups, and gemini\_handler.py for AI analysis. It aggregates results, calculates the final risk score using tuned RISK\_WEIGHTS, generates the natural language explanation via gemini\_handler.py, and returns the complete analysis results to app.py.
* src/gemini\_handler.py **(AI Interaction)**:  
  Manages all communication with the Google Gemini API. It includes tuned prompts for analyzing text content, simulating URL risk assessment, and generating actionable alert explanations. The module handles API key configuration, model setup, safety settings, and implements error handling with retry logic (\_call\_gemini\_with\_retry).
* src/utils.py **(Parsing & Utilities)**:  
  Provides essential parsing functions, including extract\_urls (URL extraction), extract\_basic\_headers (header parsing), extract\_email\_body (body extraction and decoding), and extract\_attachment\_names (attachment filename extraction). It also includes check\_suspicious\_attachment\_names to flag risky extensions or patterns and generate\_sample\_input for UI test cases.
* src/basic\_checks.py **(Simple Lookups)**:  
  Implements basic checks, including load\_blocklist to read known bad domains/URLs from data/url\_blocklist.txt, check\_url\_blocklist to match extracted URLs against the blocklist, and simulate\_sender\_checks as a placeholder for SPF/DKIM checks in manual mode.
* data/url\_blocklist.txt:  
  A simple, configurable text file listing known malicious domains or URLs for static blocking.
* .env:  
  Securely stores the GOOGLE\_API\_KEY, excluded from version control via .gitignore.

**3. Implementation Challenges and Resolutions**

* **Challenge 1: Direct Email Integration Complexity**
  + **Problem**: Securely authenticating (OAuth 2.0) and interacting with live email services (IMAP/API) was infeasible within the 20-hour hackathon timeframe and posed security risks.
  + **Resolution**: Focused on the core AI analysis engine. Implemented two input modes—"Manual Input" for quick tests and "Paste Raw Email Source" to simulate real-world workflows with headers and attachments. Positioned live integration as a future enhancement.
* **Challenge 2: AI Prompt Engineering & Tuning**
  + **Problem**: Initial prompts produced generic responses or were overly sensitive, flagging legitimate business emails (e.g., standard signatures or deadlines) as threats.
  + **Resolution**: Iteratively refined prompts with explicit instructions to differentiate normal communication from malicious tactics (e.g., "Do not flag standard deadlines unless combined with threats"). Incorporated domain reputation for URL analysis and used lower temperature settings (e.g., 0.4) for consistent AI output.
* **Challenge 3: Handling AI API Responses**
  + **Problem**: Gemini API responses could be invalid JSON, blocked by safety filters, or affected by transient errors.
  + **Resolution**: Developed \_call\_gemini\_with\_retry with retry logic for network errors, checks for blocked/empty responses, pre-parsing JSON validation, and robust error handling to provide informative UI feedback.
* **Challenge 4: False Positives from Scoring Logic**
  + **Problem**: Initial scoring weights overly penalized legitimate emails with common elements (e.g., links, keywords, formal titles).
  + **Resolution**: Adjusted RISK\_WEIGHTS in core\_analyzer.py to reduce penalties for ambiguous factors. Added contextual logic in calculate\_risk\_score to verify domain reputation before penalizing URL structures.
* **Challenge 5: Safe URL Analysis**
  + **Problem**: Directly accessing potentially malicious URLs during the hackathon was insecure and impractical.
  + **Resolution**: Adopted a simulation approach, using Gemini to analyze URL strings for suspicious patterns (domain, TLD, path keywords, HTTP) while considering domain reputation, avoiding unsafe interactions.
* **Challenge 6: Email Parsing Complexity**
  + **Problem**: Emails have complex, nested structures and varied encodings, complicating parsing.
  + **Resolution**: Leveraged Python’s email.policy.default for robust parsing of common email structures. Focused on extracting key elements (headers, body, attachment names) and acknowledged limitations with malformed emails as acceptable for a prototype.

**4. Future Scope and Productionization Plan**

**4.1. Future Scope**

* **Real-time Threat Intelligence**: Integrate with Cyber Threat Intelligence (CTI) APIs (e.g., VirusTotal, PhishTank, Recorded Future) for dynamic reputation checks on URLs, IPs, domains, and file hashes.
* **Safe URL Analysis**: Implement secure, sandboxed browsing (e.g., Browserless.io, URLScan.io API) to analyze rendered page content and detect login forms or cloaking.
* **Attachment Sandboxing**: Integrate with sandbox services (e.g., ANY.RUN, Joe Sandbox) for behavioral analysis of attachments and deeper static analysis (e.g., metadata, macros).
* **Direct Email Integration**: Develop secure OAuth 2.0 flows and leverage APIs (e.g., Gmail API, Microsoft Graph) for automated scanning of incoming emails or specific folders.
* **User Feedback Loop**: Enable users/analysts to flag analysis results (True/False Positive/Negative) to fine-tune prompts and potentially retrain classification models.
* **Advanced Header Analysis**: Implement DMARC alignment validation, analyze the full Received: header path for anomalies, and detect header spoofing techniques.
* **Multi-Vector Support**: Extend analysis to other communication channels, such as SMS, Slack, or Microsoft Teams messages, via their respective APIs.
* **Analytics Dashboard**: Build a dashboard to visualize trends, such as common tactics detected, risk score distributions, and detection rates over time.
* **Hybrid AI Approach**: Combine the LLM’s contextual analysis with traditional ML models (e.g., XGBoost, SVM) trained on specific features (header anomalies, URL characteristics, text statistics) for faster triage or complementary scoring.

**4.2. Productionization Plan**

1. **Deployment Architecture**: Transition from local Streamlit to a scalable cloud-based architecture, such as:
   * **Serverless**: Google Cloud Run or AWS Lambda with API Gateway.
   * **Containerized**: Dockerized application deployed to Kubernetes (GKE, EKS, AKS) or managed container services (ECS, App Service).
2. **Asynchronous Processing**: Use task queues (e.g., Celery with Redis/RabbitMQ, Google Cloud Tasks, AWS SQS) to handle AI API calls and long-running tasks without blocking user requests.
3. **Secure Secret Management**: Store API keys and sensitive configurations using services like Google Secret Manager, AWS Secrets Manager, or HashiCorp Vault.
4. **Robust Logging & Monitoring**: Implement structured logging (e.g., Cloud Logging, ELK stack) and monitoring dashboards (e.g., Cloud Monitoring, Datadog, Grafana) with alerts for errors and performance issues.
5. **CI/CD Pipeline**: Set up automated pipelines (e.g., GitHub Actions, GitLab CI, Jenkins) for code linting, testing, building container images, and deploying updates to staging/production.
6. **Database (Optional)**: Integrate a database (e.g., PostgreSQL, MongoDB) for persistent storage of analysis history, user data, or fine-tuning feedback.
7. **Scalability & Cost Management**: Configure auto-scaling, monitor API usage costs (especially Gemini), implement caching strategies, and optimize prompts for efficiency.
8. **Security Hardening**: Apply best practices, including input validation, dependency scanning, network security rules, and the principle of least privilege.