# Project Title: AI-Driven Static Code Vulnerability Analyzer

---

### Short Project Description:

Develop an automated tool that uses Machine Learning to detect vulnerabilities in source code **without execution**. It should learn patterns of insecure coding practices and flag them.

---

| Component | Role |
|---|---|
| Source Code | The input to be analyzed |
| Code Parser | Parses code into an intermediate representation (AST, CFG, tokens) |
| Static Analyzer (Feature Extractor) | Extracts vulnerability-relevant patterns and features |
| Machine Learning Model | Predicts vulnerability likelihood based on patterns |
| Vulnerability Report Generator | Creates a report listing detected vulnerabilities |

---

### How Components Interact:

1. **Source Code** is submitted (uploaded) by the **User**.
2. **Code Parser** parses the source into structured formats (abstract syntax trees, control flow graphs, etc).
3. The **Static Analyzer** extracts important features from the parsed code:
    - Patterns like SQL queries
    - Unsafe function calls
    - Tainted input flows
    - Etc
4. These features are **fed into a Machine Learning Model**:
    - A trained model (e.g., Random Forest, LSTM, etc) predicts if a code snippet is vulnerable.
5. Based on model predictions:
    - The **Vulnerability Report Generator** creates a vulnerability report:
        - Detected issue
        - Severity
        - Line number
        - Suggested fix
6. **User** receives the final vulnerability report.

---

## Overall System Flow :

[Source Code Uploaded]
↓
[Code Parser] ➔ (AST, CFG, Tokens)
↓
[Static Analyzer] ➔ (Extracted Features, Data Flows)
↓
[Machine Learning Model] ➔ (Predicted Vulnerabilities + Scores)
↓
[Vulnerability Report Generator] ➔ (Human/CI-readable Report)

---

## Internal Functioning of Each Module:

## 1. Source Code (Input Handling)

**Functionality**:

- Accept uploaded source files (individual or full projects).
- Supported formats:
  - C, C++, Java, Python, JavaScript, etc.
- Preprocessing:
  - Sanitize filenames,
  - Normalize file encoding (UTF-8),
  - Handle archives (zip/tar.gz unpacking).

**Challenges handled (optional)**:

- Multi-language projects (detect file type per file).
- Partial code handling (functions without full projects).

---

## 2. Code Parser (Parser Engine)

**Functionality**:

- **AST (Abstract Syntax Tree) Generation**:
  - Parse code syntax into tree structures.
  - Identify:
    - Function definitions,
    - Variable declarations,
    - Control structures (if-else, for loops, etc.).
- **CFG (Control Flow Graph) Generation**:
  - Build graphs showing flow between program states.
  - Essential for detecting logical vulnerabilities (e.g., TOCTOU flaws, etc).
- **Tokenization**:

    - Lexical analysis:
        - Keywords,
        - Operators,
        - Identifiers,
        - Literals.

**Technologies (e.g.)**:

- Tree-sitter,
- Clang AST tools,
- Custom tokenizers.
- Etc.

**Output**:

- Structured representations (AST, CFG, token streams) for the next modules.

---

## 3. Static Analyzer (Feature Extractor)

**Functionality**:

- **Feature Extraction from AST and CFG**:
    - Extract vulnerability-relevant patterns such as:
        - Function calls (e.g., strcpy, system),
        - SQL query constructions,
        - Input validation patterns.
- **Taint Analysis**:
    - Track the flow of potentially dangerous inputs through the codebase.
    - Label sources (e.g., user input) and sinks (e.g., file write, exec functions).
- **Pattern Frequency Calculation**:
    - Compute feature vectors:
        - Number of unsafe calls,
        - Depth of nested control flows,
        - Presence of error handling,
        - Etc.

**Techniques**:

- Graph traversal,
- Taint propagation algorithms,
- Static pattern matching.

**Output**:

- Structured feature vectors representing each code sample.

---

## 4. Machine Learning Model (Vulnerability Predictor)

**Functionality**:

- **Training**:
  - Models trained offline on datasets like:
    - Juliet Test Suite for C/C++,
    - Big Code datasets (for Java/Python),
    - Manually labeled vulnerable code examples,
    - Etc.
- **Prediction Process**:
  - Input:
    - Feature vectors from Static Analyzer.
  - Model types:
    - Random Forest (tabular features),
    - LSTM/RNN (for sequential token streams),
    - Graph Neural Networks (for AST/CFG structures),
    - Etc.
  - Output:
    - Vulnerability likelihood score for each code snippet.

**Prediction refinement (optional)**:

- Multi-class labeling:
  - Vulnerability types (buffer overflow, SQL injection, XSS, etc.)
- Confidence scoring (based on model certainty).

**Output**:

- Predicted vulnerabilities and associated metadata.

---

## 5. Vulnerability Report Generator

**Functionality**:

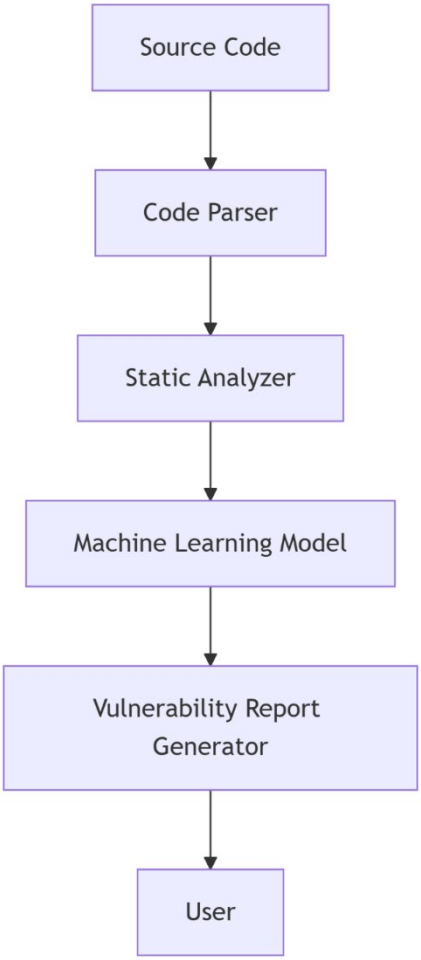
- **Report Content**:
  - Vulnerability type,
  - Severity (low, medium, high, critical),
  - Affected line numbers or functions,
  - Suggested remediation actions.
- **Formatting Options**:
  - HTML detailed report,
  - SARIF (Static Analysis Results Interchange Format) export,
  - JSON/YAML for CI/CD pipelines (optional).
- **Developer Integration (optional)**:
  - Plugins for:
    - VSCode,
    - IntelliJ IDEA,
    - GitHub PR comments integration.

**Output**:

- Structured, actionable vulnerability reports for users.

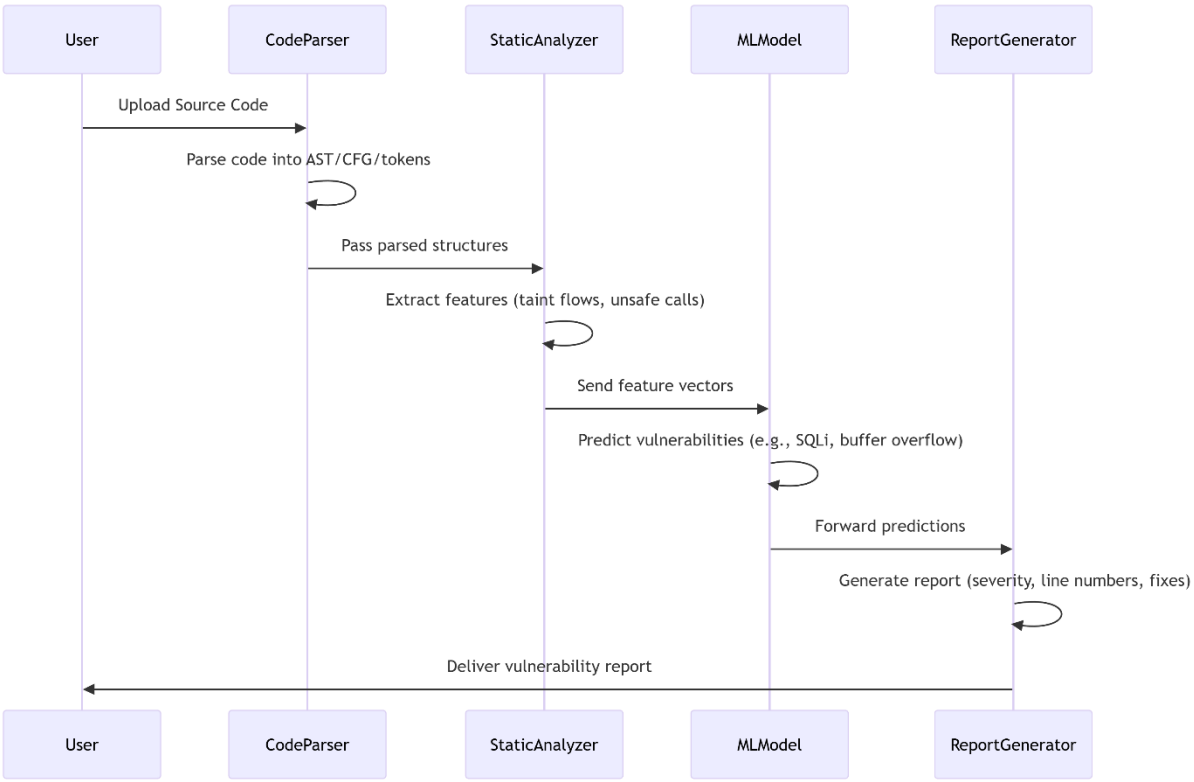
---

## Component Diagram:




- **Source Code** is processed by the **Code Parser** into structured formats (AST/CFG).
- The **Static Analyzer** extracts vulnerability-relevant features (e.g., unsafe function calls, tainted data flows).
- The **Machine Learning Model** uses these features to predict vulnerabilities.
- The **Vulnerability Report Generator** compiles predictions into a report for the **User**.

## Sequence Diagram




- The **User** uploads source code, triggering the analysis pipeline.
- The **Code Parser** converts code into structured representations.
- The **Static Analyzer** identifies critical patterns and sends features to the **ML Model**.
- The **ML Model** predicts vulnerabilities and shares results with the **Report Generator**.
- The **Report Generator** formats findings into an actionable report and returns it to the **User**.

## Detailed Project Description: AI-Driven Static Code Vulnerability Analyzer

A machine learning-powered static code analyzer for detecting vulnerabilities in source code. The system parses code, extracts security-relevant features, predicts vulnerabilities using ML models, and generates actionable reports.

---

## 1. System Overview

The tool detects vulnerabilities in source code **without execution** by combining static analysis with machine learning.

### Key Components

1. **Source Code Input Handler**
2. **Code Parser**
3. **Static Analyzer (Feature Extractor)**
4. **Machine Learning Model**
5. **Vulnerability Report Generator**

---

## 2. Component Design & Implementation

### 2.1 Source Code Input Handler

**Functionality**:

- Accepts source code (individual files or entire projects).
- Handles archives (ZIP, TAR.GZ) and preprocessing.

**Implementation Steps (e.g.)**:

1. **File Upload**
    - **User-defined**

2. **Language Detection**:
    - Detect file type using extensions or tools like `linguist` (GitHub's language detector).
3. **Preprocessing**:
    - Normalize encodings to UTF-8.
    - Extract archives using libraries like `libarchive` (C) or `zipfile` (Python). **(optional)**

**Challenges (optional)**:

- Mixed-language projects: Process each file separately based on detected language.
- Partial code: Use placeholder stubs for unresolved dependencies.

---

### 2.2 Code Parser

**Functionality**:
Generate structured representations of code:

- **Abstract Syntax Tree (AST)**
- **Control Flow Graph (CFG)**
- **Token Stream**

**Implementation Steps (e.g.)**:

1. **AST/CFG Generation**:
    - Use existing parsers:
        - **Tree-sitter** (supports C, Python, JavaScript, etc.).
        - **Clang AST** for C/C++.
        - **JavaParser** for Java.
    - For CFGs, use tools like `Code2Flow` or implement graph traversal on ASTs.
2. **Tokenization**:
    - Split code into tokens (keywords, identifiers, literals) using lexical analyzers.

**Output**:

- AST/CFG in JSON or Protocol Buffers format for downstream processing.

---

### 2.3 Static Analyzer (Feature Extractor)

**Functionality**:
Identify vulnerability patterns and extract feature vectors.

**Implementation Steps (e.g.)**:

1. **Pattern Matching**:
    - **Unsafe Function Calls**:
        - Maintain a database of risky functions (e.g., `strcpy`, `eval`, `exec`, `etc`).
        - Traverse AST to detect calls to these functions.
    - **SQL Injection**:
        - Detect dynamic SQL queries (e.g., string concatenation with `SELECT`, `etc`).
    - **Input Validation Gaps**:
        - Check for missing sanitization (e.g., no regex checks on user inputs, etc).
2. **Taint Analysis**:
    - Track data flow from sources (e.g., `request.getParameter()`) to sinks (e.g., `executeQuery()`).
    - Use tools like **CodeQL** or implement a custom taint propagation algorithm.
3. **Feature Vector Creation**:
    - Compute metrics:
        - Count of unsafe calls.
        - Depth of nested loops/conditionals.
        - Presence/absence of error handling.

**Output**:

- Feature vectors (CSV/JSON) for ML model input.

---

### 2.4 Machine Learning Model

**Functionality**:
Predict vulnerability likelihood using extracted features.

**Implementation Steps (e.g.)**:

1. **Training Data**:
    - Use labeled datasets (e.g.,):
        - **Juliet Test Suite** (C/C++ vulnerabilities).
        - **SARD** (Software Assurance Reference Dataset).
        - **BigVul** (real-world vulnerabilities from GitHub).
    - Eventually augment with synthetic data (e.g., inject vulnerabilities into clean code).
2. **Model Selection**:
    - **Random Forest/GBM**: For tabular feature vectors.
    - **LSTM/Transformer**: For token sequences.
    - **Graph Neural Networks (GNN)**: For AST/CFG structures (use PyTorch Geometric).
    - **Etc**.
3. **Training Pipeline**:
    - Preprocess data: Normalize features, balance classes.
    - Train models using frameworks like Scikit-learn (tabular) or TensorFlow/PyTorch (deep learning).
4. **Prediction**:
    - Deploy models via REST APIs (FastAPI) or embedded inference (ONNX Runtime).

**Output**:

- Vulnerability labels (e.g., "SQL Injection") with confidence scores.

---

### 2.5 Vulnerability Report Generator

**Functionality**:
Generate actionable reports for developers.

**Implementation Steps (e.g.)**:

1. **Report Formatting**:
    - **HTML**: Use Jinja2 templates for human-readable reports.
    - **SARIF**: Use the `sarif-tools` library for CI/CD integration.
    - **JSON/YAML**: For automated pipeline consumption.
2. **Developer Integration (optional)**:
    - **IDE Plugins**:
        - VSCode: Use the Language Server Protocol (LSP).
        - IntelliJ: Develop a custom plugin with JetBrains SDK.
    - **GitHub Integration**: Post results as PR comments using GitHub API.

**Output**:

- Reports include:
    - Vulnerability type, severity (CVSS-based scoring).
    - Line numbers, code snippets.
    - Remediation suggestions (e.g., "Use parameterized queries").

---

## 3. Evaluation & Validation

1. **Model Performance**:
    - Metrics: Precision, Recall, F1-score, etc.

    - Baseline: Compare against SAST tools like SonarQube or Checkmarx.
2. **False Positive Reduction**:
    - Use manual validation on open-source projects (e.g., Apache Commons, etc).
3. **User Testing**:
    - Collect feedback from developers on report clarity and usability.

---

## 4. Technology Stack (e.g.)

- **Parsing**: Tree-sitter, Clang, JavaParser, etc.
- **Static Analysis**: CodeQL, custom taint analyzers, etc.
- **ML**: Scikit-learn, PyTorch, HuggingFace Transformers, etc.
- **Backend**: Python (Flask/FastAPI), Node.js (for IDE plugins), etc.
- **Reporting**: Jinja2, SARIF SDK, etc.

---

## 5. Development Roadmap

1. **Phase 1**: Implement code parsing and static analysis.
2. **Phase 2**: Train and validate ML models.
3. **Phase 3**: Build report generator and integrations.
4. **Phase 4 (optional)**: Pilot testing and refinement.

---

## 6. Challenges & Mitigations (optional)

- **Multi-Language Support**: Prioritize common languages first (C, Python).
- **Model Accuracy**: Use ensemble models and active learning.
- **Performance**: Optimize AST traversal with parallel processing.

---