Architecting the Hybrid Sentry: A Comprehensive Guide to Building a GenAI-Powered Secure Code Review Bot

1. Executive Summary: The Imperative for Intelligent Security

In the contemporary landscape of software engineering, the velocity of deployment has become a primary competitive differentiator. The paradigm of DevOps has successfully collapsed the barrier between development and operations, necessitating a parallel evolution in security practices, widely termed DevSecOps. However, the integration of security into high-velocity pipelines faces a critical bottleneck: the signal-to-noise ratio of traditional automated tooling. Static Application Security Testing (SAST) has served as the bedrock of automated code review for decades. While these deterministic engines are unparalleled in their ability to parse vast repositories and identify syntactic anomalies, they fundamentally lack semantic understanding. Research indicates that SAST tools often produce false positive rates exceeding 50%, and in complex legacy codebases, this can rise to nearly 90%.[1]

The consequence of this inefficiency is "alert fatigue." Security teams and developers, inundated with thousands of low-fidelity warnings, inevitably begin to treat security alerts as background noise. This desensitization leads to the dangerous practice of ignoring alerts or disabling security checks entirely to maintain deployment velocity, thereby paradoxically increasing organizational risk despite the presence of security tooling.[3]

The emergence of Large Language Models (LLMs) and Generative AI (GenAI) offers a transformative solution to this deadlock. Unlike deterministic AST-based scanners, LLMs possess the capability to infer intent, reason about variable states across disparate contexts, and understand the nuanced difference between a theoretical vulnerability and an exploitable flaw. However, relying solely on LLMs for code review is currently unfeasible due to prohibitive costs, high latency, and the risk of "hallucinations"—the generation of plausible but non-existent vulnerabilities.[4]

This report presents a rigorous, architectural blueprint for a **Hybrid Engine**—a system that fuses the high-recall capabilities of deterministic SAST with the high-precision reasoning of probabilistic LLMs. By utilizing SAST as a candidate generator and LLMs as a context-aware verification layer, organizations can reduce false positives by an order of magnitude while maintaining strict adherence to data privacy standards and OWASP Top 10 compliance. This document details the end-to-end construction of such a system, specifically targeting Python and JavaScript ecosystems, providing an exhaustive analysis of the architecture, privacy

controls, and evaluation metrics required for a production-grade deployment.

---

2. The Theoretical Framework: Merging Deterministic and Probabilistic Analysis

To understand the architecture of the proposed bot, one must first dissect the contrasting operational modalities of the two core technologies involved: Static Analysis and Generative AI. The proposed hybrid architecture is not merely a juxtaposition of tools but a functional integration where the output of one serves as the grounded input for the other.

2.1 The Deterministic Layer: Static Application Security Testing (SAST)

SAST tools operate by converting source code into an Abstract Syntax Tree (AST) or a Control Flow Graph (CFG). They then traverse these structures looking for specific patterns or "signatures" that match known vulnerability templates.

For instance, in Python, a SAST tool like **Bandit** might look for the usage of the exec() function. In the AST, this appears as a Call node where the function name identifier is exec. The tool flags this deterministically: if the pattern exists, the alert is triggered.[5]

Advantages:

- **Completeness (High Recall):** SAST tools are exhaustive. They will not "miss" a pattern they are programmed to find.
- **Speed:** Analysis happens in seconds, even for large codebases.
- **Local Execution:** No data leaves the environment.

Limitations:

- **Context Blindness:** A SAST tool struggles to determine if a dangerous function is being called with safe, static data or user-controlled input. It often lacks the ability to trace data sanitization routines defined in external libraries or complex middleware.[2]
- **False Positives:** Because of the bias towards safety, SAST tools default to flagging ambiguous cases as vulnerable.

2.2 The Probabilistic Layer: Large Language Models (LLMs)

LLMs operate on a fundamentally different principle. Trained on vast corpora of code (including GitHub repositories, StackOverflow discussions, and security advisories), they model the statistical probability of tokens. When presented with code, an LLM does not just see a syntax tree; it "reads" the variable names, comments, and logic flow to construct a semantic representation of the code's intent.[1]

Advantages:

- **Semantic Understanding:** LLMs can infer that a variable named cleaned_input likely went through a sanitization process, or that a function named internal_script_do_not_use is not an exposed attack surface.
- **Reasoning:** They can follow complex chains of logic, such as "If x is validated here, then

the usage of x in the SQL query three lines later is safe".[8]

Limitations:

- **Stochasticity:** The same input can produce different outputs.
- **Hallucination:** Without grounding, an LLM might confidently invent a vulnerability that defies the syntax of the language.
- **Resource Intensity:** High-quality inference (e.g., GPT-4 class) is slow and expensive.

2.3 The Hybrid "Filter-Verify" Architecture

The proposed architecture employs a **Candidate Generation and Verification** pattern. This mirrors search engine architectures where a cheap, fast algorithm retrieves a candidate set of documents, and a heavy, expensive machine learning model re-ranks them.

The Workflow:

1. **Signal Generation (Recall Layer):** The SAST tool scans the repository. It is configured to be "noisy"—we explicitly want high recall. We accept that 50-70% of these findings may be false positives.
2. **Context Extraction:** The system algorithmically extracts the relevant code snippet surrounding the SAST finding. This is the "grounding" step.
3. **Verification (Precision Layer):** The LLM acts as a judge. It is prompted not to "find bugs" (which encourages hallucination) but to "verify the SAST finding" (which constrains the search space).
4. **Verdict:** The system outputs only those findings verified by the LLM.

This approach leverages the strengths of both: the exhaustive coverage of SAST and the intelligent filtering of the LLM.[1]

---

3. System Architecture and Technology Stack

The Secure Code Review Bot is designed as a distributed, asynchronous system. A synchronous design (where the HTTP request waits for the scan to finish) would be prone to timeouts given the latency of LLM inference. Therefore, we adopt an event-driven microservices architecture.

3.1 High-Level Component Design

Component	Technology Choice	Justification
API Gateway	**FastAPI (Python)**	High-performance, native async support, and auto-generated Swagger documentation for integration.[9]
Task Queue	**Celery**	robust distributed task queue that handles retries, scheduling, and worker

		management effectively.[10]
Message Broker	**Redis**	In-memory data structure store, ideal for the high-throughput, low-latency requirements of a task broker.[9]
Scanner (Python)	**Bandit**	The de facto standard for Python static analysis, designed for security and offering JSON output.[5]
Scanner (JS)	**ESLint**	Extremely extensible; with security plugins, it covers the node.js ecosystem comprehensively.[11]
Sanitizer	**Microsoft Presidio**	Enterprise-grade PII recognition using NLP and regex, essential for data privacy compliance.[13]
LLM Interface	**OpenAI API / Local LLM**	Provides the reasoning engine. The architecture supports hot-swapping models (e.g., GPT-4o to Llama 3) via an adapter pattern.
Database	**PostgreSQL**	Relational storage for audit logs, findings history, and false positive tracking.

3.2 Detailed Data Flow

1. **Trigger:** A developer pushes code to a Git repository (GitHub/GitLab). A webhook fires, sending a payload to the FastAPI /webhook endpoint.
2. **Ingestion:** FastAPI validates the webhook signature (HMAC) to ensure authenticity. It then creates a ScanJob in the database and pushes a task to the Redis queue. It returns a 202 Accepted response immediately.
3. **Orchestration:** A Celery worker picks up the task.
    - **Step 3a:** The worker clones the repository to an ephemeral volume.
    - **Step 3b (SAST):** Depending on the language detected, it runs Bandit (Python) or ESLint (JS) in "json-reporter" mode.
4. **Contextualization:** For each finding in the SAST JSON output, the **Context Engine** uses AST parsing to extract the specific function and class definitions surrounding the vulnerable line.
5. **Sanitization:** The extracted context is passed through the **Privacy Firewall**, where secrets and PII are redacted.
6. **Verification:** The sanitized snippet and the SAST error message are sent to the LLM

with a specific verification prompt.
7. **Reporting:** If the LLM validates the finding, the bot uses the Git provider's API to post a comment directly on the Pull Request line.

---

4. Component Implementation: The Signal Generators (SAST)

The foundation of the hybrid engine is the "Candidate Generator." We must configure these tools to align with the OWASP Top 10.

4.1 Python Security Scanning with Bandit

Bandit processes Python code by building an AST and running plugins against it. Unlike a linter (like Flake8) which checks style, Bandit checks for security logic.[5]
Configuration Strategy:
To maximize recall for OWASP categories, we configure Bandit to be aggressive. We define a custom profile that specifically targets the OWASP Top 10 for 2021.14
Implementation:
We create a bandit.yaml configuration file. Note that we explicitly include tests that might generate false positives (like B101 assert used) because we rely on the LLM to filter them (e.g., to see if the assert is in test code or production code).

YAML

```
# bandit.yaml
profiles:
  owasp_hybrid:
    include:
      # A03:2021 - Injection
      - B102  # exec_used
      - B601  # paramiko_calls
      - B602  # subprocess_popen_with_shell_equals_true

      # A02:2021 - Cryptographic Failures
      - B303  # md5
      - B324  # hashlib_new_insecure_functions

      # A07:2021 - Identification and Authentication Failures
      - B105  # hardcoded_password_string
      - B106  # hardcoded_password_funcarg
```

```
    # A05:2021 - Security Misconfiguration
    - B101  # assert_used (often used to protect logic, removed in optimization)

    # A01:2021 - Broken Access Control
    # (Note: SAST is weak here, but we check for dangerous file perms)
    - B108  # hardcoded_tmp_directory
```

Programmatic Execution:
Rather than running Bandit via the CLI and parsing stdout, we invoke it via subprocess to ensure we capture the full JSON output safely. The JSON output 16 is critical because it provides the lineno, filename, and code snippet, which are the keys for our Context Engine.

Python

```
import subprocess
import json
from typing import List, Dict

def run_bandit_scan(target_directory: str) -> List:
    """
    Runs Bandit SAST and returns a list of finding objects.
    """
    command = [
        "bandit",
        "-r", target_directory,
        "-f", "json",
        "-c", "bandit.yaml"  # Use our aggressive profile
    ]

    try:
        result = subprocess.run(command, capture_output=True, text=True)
        # Bandit returns exit code 1 if issues are found, so we don't check_returncode
        output_data = json.loads(result.stdout)
        return output_data.get("results",)
    except json.JSONDecodeError:
        # Log error in production
        return
```

4.2 JavaScript Security Scanning with ESLint

For JavaScript and Node.js, **ESLint** is the platform of choice. Its plugin architecture allows us to add security-specific rulesets.[11]

Plugin Selection:

1. eslint-plugin-security: Identifies generic security hotspots (e.g., child_process.exec with variables).
2. eslint-plugin-no-unsanitized: Specifically targets DOM-based XSS attacks, crucial for the "Client-Side" risks in OWASP Top 10.[11]

Configuration (.eslintrc.json):

JSON

```
{
  "env": {
    "node": true,
    "es6": true
  },
  "plugins": ["security", "no-unsanitized"],
  "extends":,
  "rules": {
    "security/detect-object-injection": "warn", // High FP rate, ideal for LLM verification
    "security/detect-eval-with-expression": "error",
    "no-unsanitized/method": "error"
  }
}
```

Node.js API Integration:
Using the ESLint class from the Node.js API is preferred over the CLI for performance, as it avoids the overhead of spawning new processes for every scan in a high-throughput worker environment.12

JavaScript

```
const { ESLint } = require("eslint");

async function runEslintScan(filePaths) {
  const eslint = new ESLint();
  const results = await eslint.lintFiles(filePaths);

  // Flatten results to a list of findings
  const findings =;
  results.forEach(result => {
    result.messages.forEach(msg => {
      if (msg.ruleId && (msg.ruleId.startsWith('security/') |
```

```
| msg.ruleId.startsWith('no-unsanitized/'))) {
        findings.push({
          file: result.filePath,
          line: msg.line,
          ruleId: msg.ruleId,
          message: msg.message
        });
      }
    });
  });
  return findings;
}
```

---

5. Component Implementation: The Context Engine (Semantic Slicing)

This component is the intellectual core of the non-AI logic. A raw SAST finding often provides only one line of code:
cursor.execute("SELECT * FROM users WHERE id = " + user_id)
If sent to an LLM, the LLM will correctly identify this as SQL Injection. However, if the line immediately preceding it was:
user_id = int(request.args.get('id'))
...then the code is safe (in Python, casting to int neutralizes the injection).
The **Context Engine** must "slice" the code to provide the LLM with the necessary visibility to make this determination.

5.1 AST Slicing Strategy

We avoid using simple text windowing (e.g., "get 5 lines before and after") because code structure is hierarchical, not linear. A "previous line" might be a comment or a blank space. We need the *logical* context.
The Algorithm:

1. **Parse the File:** Generate the full AST of the source file containing the finding.
2. **Scope Resolution:** Traverse the AST to find the "Parent Scope" of the vulnerable line. This is usually a FunctionDef (Python) or FunctionDeclaration (JS).
3. **Global Context:** Identify global variables, imports, and class attributes that are referenced within that scope.
4. **Extraction:** Reconstruct the source code for just that function and the relevant globals.[18]

5.2 Python Implementation Details

We utilize Python's ast module. The ast.walk() function allows us to iterate through all nodes. We look for the node that functionally encloses the line number reported by Bandit.

Python

```
import ast

def get_semantic_context(source_code: str, target_lineno: int) -> str:
    """
    Extracts the function definition containing the target line,
    plus relevant imports.
    """
    try:
        tree = ast.parse(source_code)
    except SyntaxError:
        return source_code  # Fallback to raw text if parsing fails

    context_lines =

    # 1. Extract Imports (Vital for LLM to know libraries used)
    # e.g., knowing 'from sqlalchemy import text' changes the verdict
    for node in tree.body:
        if isinstance(node, (ast.Import, ast.ImportFrom)):
            context_lines.append(ast.get_source_segment(source_code, node))

    # 2. Extract The Target Function
    target_function = None
    # We want the *innermost* function containing the line
    for node in ast.walk(tree):
        if isinstance(node, (ast.FunctionDef, ast.AsyncFunctionDef)):
            # Check if line is within this function's bounds
            if node.lineno <= target_lineno <= node.end_lineno:
                target_function = node
                # Don't break; continue to find nested functions

    if target_function:
        context_lines.append("\n" + ast.get_source_segment(source_code, target_function))
    else:
        # Fallback: If code is at module level, take a window
        lines = source_code.splitlines()
```

```
            start = max(0, target_lineno - 5)
            end = min(len(lines), target_lineno + 5)
            context_lines.append("\n".join(lines[start:end]))

    return "\n".join(context_lines)
```

Insight: This approach significantly reduces token usage compared to sending whole files, while maintaining the semantic integrity required for analysis. For JavaScript, a similar logic is applied using the parser acorn or traversing the AST provided by ESLint.[20]

---

6. Component Implementation: The Privacy Firewall (Data Sovereignty)

A significant barrier to GenAI adoption in security is the risk of data leakage. Sending proprietary code to an external API (like OpenAI) carries risks. The **Privacy Firewall** ensures that sensitive data is scrubbed *before* it leaves the controlled environment.

6.1 Requirements

1. **Secret Scrubbing:** Remove API keys, passwords, and tokens.
2. **PII Scrubbing:** Remove names, emails, and IP addresses (often found in comments or test data).
3. **Utility Preservation:** The scrubbing must not destroy the code structure (e.g., removing variable names that indicate taint).

6.2 The Scrubbing Pipeline

We employ a multi-stage scrubber utilizing **Microsoft Presidio** for NLP-based PII detection and a regex-based engine for secret detection.[13]

Stage 1: High-Entropy Secret Detection

We use patterns similar to detect-secrets or trufflehog. We look for high-entropy strings assigned to variables like api_key, password, secret.

- *Action:* Replace value with "<REDACTED_SECRET>".
- *Implementation Note:* We do *not* redact the variable name (e.g., aws_secret_key), only the value string. The variable name is a critical signal for the LLM to identify the sensitivity of the data.

Stage 2: PII Anonymization

We use Presidio's Analyzer Engine. We restrict the analysis primarily to comments and string literals, avoiding code keywords.

Python

```
from presidio_analyzer import AnalyzerEngine
from presidio_anonymizer import AnonymizerEngine
from presidio_anonymizer.entities import OperatorConfig

# Initialize Engines
analyzer = AnalyzerEngine()
anonymizer = AnonymizerEngine()

def scrub_code_context(text: str) -> str:
    # Analyze text for PII entities
    results = analyzer.analyze(
        text=text,
        entities=,
        language='en'
    )

    # Anonymize found entities
    anonymized_result = anonymizer.anonymize(
        text=text,
        analyzer_results=results,
        operators={
            "DEFAULT": OperatorConfig("replace", {"new_value": "<PII_REDACTED>"})
        }
    )
    return anonymized_result.text
```

Stage 3: Corporate Terminology (Optional)
For highly sensitive environments, a custom "Denylist" of internal project names (e.g., "Project_Manhattan") can be added to the scrubber to replace them with generic terms like "Project_Alpha".

---

7. Component Implementation: The Verification Engine (LLM)

The Verification Engine is the decision maker. It receives the *Contextualized* and *Sanitized* code and determines if the SAST finding is valid.

7.1 Prompt Engineering: The "Filter" Persona

The prompt is the interface to the LLM's reasoning. We must use specific techniques to ensure reliability.
Technique 1: Role Playing (Persona)

We explicitly instruct the LLM to adopt the persona of a "Senior Security Engineer." This primes the model to access the relevant latent space associated with security auditing, vulnerability triage, and OWASP standards.24
Technique 2: Chain-of-Thought (CoT)
We require the LLM to explain its reasoning before giving a verdict. This forces the model to generate intermediate logic steps (e.g., "I see user input entering at line X...", "I see a validation function at line Y..."), which significantly reduces hallucination rates compared to asking for a simple Yes/No.26
The System Prompt:
You are an expert Application Security Engineer specializing in Python and JavaScript.
Your task is to Triage and Verify vulnerability reports generated by automated SAST tools.
SAST tools are deterministic and often lack context, leading to False Positives.
You must analyze the provided Code Context and the SAST Finding to determine if the vulnerability is a TRUE POSITIVE or a FALSE POSITIVE.
GUIDELINES:

1. Trust the code context over the SAST finding. If the code shows sanitization, it is a False Positive.
2. Assume data from 'request', 'input', 'event' is tainted unless validated.
3. If the code is clearly a Test File (e.g., mocks, unit tests), classify as 'False Positive' but note it is 'Test Code'.
4. Provide a step-by-step reasoning for your decision.

7.2 Structured Outputs via Pydantic

To build a reliable software system, we cannot parse natural language. We need structured JSON. We use the Pydantic library in conjunction with the OpenAI API's "Structured Outputs" mode (or equivalent JSON-mode for other models).[27]
The Data Model:

Python

```
from pydantic import BaseModel, Field
from typing import Literal, Optional

class VulnerabilityAssessment(BaseModel):
    is_vulnerable: bool = Field(
        ...,
        description="True if the code contains a genuine, exploitable vulnerability."
    )
    confidence_score: float = Field(
        ...,
        ge=0.0, le=1.0,
        description="Confidence in the assessment (0.0 to 1.0)."
```

```
    )
    severity: Literal = Field(
        ...,
        description="The severity of the vulnerability."
    )
    category: str = Field(
        ...,
        description="The OWASP Top 10 category (e.g., 'A03: Injection')."
    )
    reasoning: str = Field(
        ...,
        description="A concise, step-by-step explanation of the vulnerability or why it is a false positive."
    )
    remediation: Optional[str] = Field(
        None,
        description="A suggested code fix or mitigation strategy if vulnerable."
    )
```

Integration Code:

Python

```
from openai import OpenAI

client = OpenAI(api_key="...")

def verify_finding(code_context: str, sast_message: str) -> VulnerabilityAssessment:
    completion = client.beta.chat.completions.parse(
        model="gpt-4-2024-08-06",
        messages=,
        response_format=VulnerabilityAssessment,
    )
    return completion.choices.message.parsed
```

This ensures that the output is always parseable and strictly typed, preventing downstream crashes in the bot.[27]

---

8. Evaluation Metrics and Performance Validation

Building the bot is only half the battle. Proving it works requires a rigorous evaluation

framework. We cannot rely on anecdotal evidence ("it feels smarter"). We must use statistical metrics derived from Information Retrieval theory.

8.1 The Benchmark Dataset: Juliet Test Suite

We utilize the **Juliet Test Suite** (NIST) for our ground truth. The Juliet suite contains thousands of test cases in C/C++ and Java, and importantly, a subset for Python.[31] Each test case is labeled as "Good" (secure) or "Bad" (vulnerable).

Experimental Setup:

1. **Ingestion:** We load the Juliet Python test cases for Injection (CWE-78, CWE-89) and Authentication.
2. **Baseline:** Run raw Bandit on the dataset. Record the False Positive Rate.
3. **Experiment:** Run the Hybrid Engine (Bandit + LLM) on the same dataset.
4. **Comparison:** Compare the confusion matrices.

8.2 Metrics Definition

To quantify performance, we track the following metrics [33]:

Metric	Formula	Business Interpretation
False Positive Rate (FPR)	$FPR = \frac{FP}{FP + TN}$	The percentage of safe code incorrectly flagged as dangerous. **Goal: < 5%.** This is the primary metric for "Developer Trust."
False Negative Rate (FNR)	$FNR = \frac{FN}{FN + TP}$	The percentage of real vulnerabilities missed. **Goal: < 5%.** This is the primary metric for "Security Risk."
Recall (Sensitivity)	$Recall = \frac{TP}{TP + FN}$	How many bugs did we find? (Inverse of FNR). SAST usually has 100% recall; the Hybrid engine should maintain this.
Precision	$Precision = \frac{TP}{TP + FP}$	When the bot speaks, is it telling the truth? High precision reduces alert fatigue.
F2-Score	$F_2 = (1 + 2^2) \cdot \frac{Precision \cdot Recall}{(2^2 \cdot Precision) + Recall}$	We use F2 instead of F1 because **Recall is more important than Precision** in security. Missing a vulnerability (FN) is costlier than a false alarm (FP).

8.3 The Confusion Matrix of Hybrid Analysis

The introduction of the LLM changes the nature of the confusion matrix.

- **SAST-TP / LLM-TP: Verified Vulnerability.** (Action: Block PR).
- **SAST-FP / LLM-TN: Successful Filtering.** The SAST tool was wrong, the LLM caught it. (Action: Suppress Alert).
- **SAST-TP / LLM-FN: Dangerous Silence.** The SAST tool was right, but the LLM hallucinated safety (e.g., misinterpreting a validation function). (Action: Log for retraining).
- **SAST-FN: Missed Opportunity.** The SAST tool never flagged it, so the LLM never saw it. This highlights the dependency on the quality of the SAST rules.[1]

---

9. Operational Infrastructure: Deployment

To ensure scalability and reliability, the system is containerized and orchestrated.

9.1 Containerization strategy

We define a Dockerfile for the worker node that includes the necessary runtimes.

Dockerfile

```
FROM python:3.10-slim

# Install Node.js for ESLint
RUN apt-get update && apt-get install -y nodejs npm git

# Install Python Dependencies
COPY requirements.txt.
RUN pip install --no-cache-dir -r requirements.txt

# Install JS Dependencies
RUN npm install -g eslint eslint-plugin-security eslint-plugin-no-unsanitized

# Copy Application Code
COPY. /app
WORKDIR /app

# Entrypoint for Celery
CMD ["celery", "-A", "app.tasks", "worker", "--loglevel=info"]
```

9.2 Infrastructure as Code (docker-compose)

For easy deployment, we coordinate the services using Docker Compose.[9]

YAML

```yaml
version: '3.8'

services:
  redis:
    image: redis:alpine
    ports:
      - "6379:6379"

  api:
    build:.
    command: uvicorn app.main:app --host 0.0.0.0 --port 8000
    ports:
      - "8000:8000"
    environment:
      - CELERY_BROKER_URL=redis://redis:6379/0
      - OPENAI_API_KEY=${OPENAI_API_KEY}
    depends_on:
      - redis

  worker:
    build:.
    command: celery -A app.worker worker --loglevel=info --concurrency=4
    environment:
      - CELERY_BROKER_URL=redis://redis:6379/0
      - OPENAI_API_KEY=${OPENAI_API_KEY}
    depends_on:
      - redis
```

Insight: The concurrency flag in the worker is crucial. Since the worker spends most of its time waiting for the LLM API (I/O bound), we can run multiple concurrent threads per container to maximize throughput.

---

10. Future Horizons: From Review to Remediation

The architecture described above focuses on **Detection** and **Triage**. However, the structured nature of the LLM output (specifically the remediation field in our Pydantic model) paves the way for **Agentic Remediation**.
Future iterations of this bot could:

1. **Auto-Fix:** Instead of just commenting, the bot creates a new commit with the suggested fix applied.
2. **Test Verification:** The bot could spawn a sandbox, apply the fix, run the project's unit tests to ensure no regression, and *then* push the code.
3. **Cross-File Taint Analysis:** By integrating Graph RAG (Retrieval Augmented Generation), the bot could understand data flow across the entire repository, solving the remaining "Context Blindness" issues of the current AST slicing method.

11. Conclusion

The construction of a GenAI-powered Secure Code Review Bot is a sophisticated engineering challenge that moves beyond simple API calls. It requires a deep integration of legacy security tooling (SAST) with cutting-edge probabilistic models (LLMs). The Hybrid Architecture proposed in this report—anchored by the "Filter-Verify" pattern, robust Context Slicing, and strict Privacy Controls—provides a viable path to solving the alert fatigue crisis.
By implementing this system, organizations can expect to see a drastic reduction in false positives, a measurable increase in developer trust, and ultimately, a more secure software supply chain. The technology is no longer theoretical; the tools and frameworks (FastAPI, Celery, LangChain/Pydantic, Bandit, ESLint) are mature and ready for production integration. The era of intelligent, context-aware security automation has arrived.

Works cited

1. [2509.15433] LLM-Driven SAST-Genius: A Hybrid Static Analysis Framework for Comprehensive and Actionable Security - arXiv, accessed on December 16, 2025, https://arxiv.org/abs/2509.15433
2. Large Language Models Versus Static Code Analysis Tools: A Systematic Benchmark for Vulnerability Detection - arXiv, accessed on December 16, 2025, https://arxiv.org/html/2508.04448v1
3. Evaluating Static Analysis Alerts with LLMs - Software Engineering Institute, accessed on December 16, 2025, https://www.sei.cmu.edu/blog/evaluating-static-analysis-alerts-with-llms/
4. LLM-Driven SAST-Genius: A Hybrid Static Analysis Framework for Comprehensive and Actionable Security - arXiv, accessed on December 16, 2025, https://www.arxiv.org/pdf/2509.15433
5. json — Bandit documentation, accessed on December 16, 2025, https://bandit.readthedocs.io/en/latest/formatters/json.html
6. Getting Started — Bandit documentation, accessed on December 16, 2025, https://bandit.readthedocs.io/en/latest/start.html
7. A Survey on Code Generation with LLM-based Agents - arXiv, accessed on December 16, 2025, https://arxiv.org/html/2508.00083v1
8. LLM-Driven SAST-Genius: A Hybrid Static Analysis ... - arXiv, accessed on December 16, 2025, https://arxiv.org/pdf/2509.15433
9. SteliosGian/fastapi-celery-redis-flower - GitHub, accessed on December 16, 2025, https://github.com/SteliosGian/fastapi-celery-redis-flower

10. FastAPI Tutorial #12: Background Jobs From Base to Production with Celery, Redis & Docker - YouTube, accessed on December 16, 2025, https://www.youtube.com/watch?v=V9z13NUJDhs
11. OWASP Top 10 Client-Side Security Risks, accessed on December 16, 2025, https://owasp.org/www-project-top-10-client-side-security-risks/
12. Node.js API Reference - ESLint - Pluggable JavaScript Linter, accessed on December 16, 2025, https://eslint.org/docs/latest/integrate/nodejs-api
13. A Python library for anonymizing sensitive information in text data. Focused on Swiss French banking data. - GitHub, accessed on December 16, 2025, https://github.com/idiap/anonymization
14. The Pythonista's Guide to the OWASP Top 10 - devmio, accessed on December 16, 2025, https://devm.io/python/python-owasp-app-security
15. OWASP Top 10:2021, accessed on December 16, 2025, https://owasp.org/Top10/2021/
16. bandit/bandit/formatters/json.py at main · PyCQA/bandit - GitHub, accessed on December 16, 2025, https://github.com/PyCQA/bandit/blob/master/bandit/formatters/json.py
17. Node.js API - ESLint - Pluggable JavaScript linter - GitHub Pages, accessed on December 16, 2025, https://denar90.github.io/eslint.github.io/docs/developer-guide/nodejs-api
18. ast — Abstract syntax trees — Python 3.14.2 documentation, accessed on December 16, 2025, https://docs.python.org/3/library/ast.html
19. Learn Python ASTs by building your own linter - DeepSource, accessed on December 16, 2025, https://deepsource.com/blog/python-asts-by-building-your-own-linter
20. Extract line numbers of all docstrings? - python - Stack Overflow, accessed on December 16, 2025, https://stackoverflow.com/questions/11608747/extract-line-numbers-of-all-docstrings
21. How to get source corresponding to a Python AST node? - Stack Overflow, accessed on December 16, 2025, https://stackoverflow.com/questions/16748029/how-to-get-source-corresponding-to-a-python-ast-node
22. IFCA-Advanced-Computing/anjana: ANJANA is a Python library for anonymizing sensitive data - GitHub, accessed on December 16, 2025, https://github.com/IFCA-Advanced-Computing/anjana
23. Secrets Sprawl in AI: Secure Non-Human Identities Before LLM Deployment, accessed on December 16, 2025, https://blog.gitguardian.com/before-you-deploy-that-llm-nhi/
24. Prompt pattern for reducing false-positive safety responses in reflective/extended reasoning workflows : r/OpenAI - Reddit, accessed on December 16, 2025, https://www.reddit.com/r/OpenAI/comments/1pbmavs/prompt_pattern_for_reducing_falsepositive_safety/
25. How to Prompt LLMs for Better, Faster Security Reviews - Crash Override,

accessed on December 16, 2025, https://crashoverride.com/blog/prompting-llm-security-reviews
26. Using LLMs to filter out false positives from static code analysis - Datadog, accessed on December 16, 2025, https://www.datadoghq.com/blog/using-llms-to-filter-out-false-positives/
27. Structured model outputs | OpenAI API, accessed on December 16, 2025, https://platform.openai.com/docs/guides/structured-outputs
28. Pydantic Model Responses API - OpenAI Developer Community, accessed on December 16, 2025, https://community.openai.com/t/pydantic-model-responses-api/1147202
29. The Complete Guide to Using Pydantic for Validating LLM Outputs, accessed on December 16, 2025, https://machinelearningmastery.com/the-complete-guide-to-using-pydantic-for-validating-llm-outputs/
30. Introducing Structured Outputs in the API - OpenAI, accessed on December 16, 2025, https://openai.com/index/introducing-structured-outputs-in-the-api/
31. LorenzH/juliet_test_suite_c_1_3 · Datasets at Hugging Face, accessed on December 16, 2025, https://huggingface.co/datasets/LorenzH/juliet_test_suite_c_1_3
32. Juliet Test Suite v1.2 for C/C++ User Guide | SAMATE | NIST, accessed on December 16, 2025, https://samate.nist.gov/SARD/resources/Juliet_Test_Suite_v1.2_for_C_Cpp_-_User_Guide.pdf
33. Classification: Accuracy, recall, precision, and related metrics | Machine Learning, accessed on December 16, 2025, https://developers.google.com/machine-learning/crash-course/classification/accuracy-precision-recall
34. Performance Metrics: Confusion matrix, Precision, Recall, and F1 Score, accessed on December 16, 2025, https://towardsdatascience.com/performance-metrics-confusion-matrix-precision-recall-and-f1-score-a8fe076a2262/
35. Precision-Recall — scikit-learn 1.8.0 documentation, accessed on December 16, 2025, https://scikit-learn.org/stable/auto_examples/model_selection/plot_precision_recall.html
36. Accuracy, Recall, Precision, & F1-Score with Python | by Max Grossman | Medium, accessed on December 16, 2025, https://medium.com/@maxgrossman10/accuracy-recall-precision-f1-score-with-python-4f2ee97e0d6
37. Dockerizing Celery and FastAPI - TestDriven.io, accessed on December 16, 2025, https://testdriven.io/courses/fastapi-celery/docker/