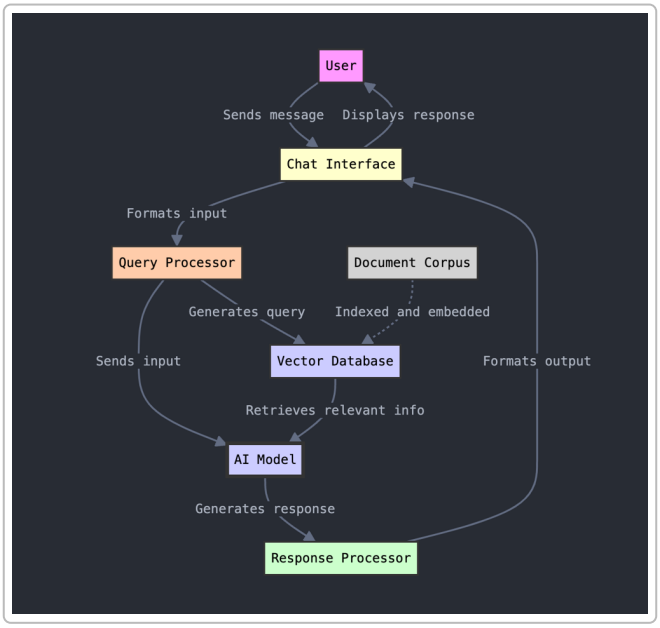
ChatGPT

Static RAG Security Scanner: Mechanics, Security Issues, and Framework Alignment

Introduction

Modern **retrieval-augmented generation (RAG)** architectures combine Large Language Models with external knowledge (via vector databases) to improve accuracy. However, these systems introduce new security and configuration challenges that traditional app security tools might miss. Developers are increasingly seeking *static scanning tools* (built in Python, in this case) to analyze RAG-based applications for vulnerabilities, misconfigurations, and policy compliance **early in development** – thereby reducing security issues before deployment. This research outlines how such a scanner might work, the key security issues it should catch (mapping to the latest **OWASP Top 10 for LLMs (2025)**), and how it aligns with broader AI risk frameworks like **NIST's AI Risk Management Framework (RMF)**. We also compare these frameworks and best practices to highlight their different focus areas and how they complement each other in guiding secure AI system development.

RAG and Vector Databases: Security Challenges



A typical Retrieval-Augmented Generation (RAG) pipeline: user queries are processed and relevant knowledge is retrieved from a vector database to augment the prompt, allowing the AI model to generate more contextually accurate responses [1]. This architecture improves factual accuracy but introduces security considerations around the retrieval layer – for example, protecting sensitive data in the vector store and ensuring malicious context cannot be injected into the model's prompt [2] [3]. Scanning the components of this pipeline (code, configuration, and data) is essential to catch vulnerabilities early.

In a RAG system, a **vector database** stores embeddings of documents or data, which the application retrieves to provide context to the LLM. This extra retrieval step opens up new attack surfaces. For

instance, weaknesses in how embeddings are **generated, stored, or fetched** can be exploited to inject malicious content or leak sensitive information [2]. A common example is *context injection*, where an attacker inserts harmful or misleading text into the data indexed by the vector store – if the system naively pulls this "poisoned" context, the model may output manipulated or unsafe responses [3]. Another risk is **unauthorized access**: if access controls on the vector database are misconfigured, one user or tenant could retrieve another tenant's private data (a form of multi-tenant data leakage) [4].

Embedding inversion attacks are also a concern – skilled adversaries might reconstruct original data from stored embeddings [5]. And if the RAG pipeline allows users to upload content (documents to be indexed), there is a risk of **data poisoning** where malicious uploads blend in with legitimate data and alter the model's behavior [6]. All these issues mean a static scanner should not only check the **LLM prompt and code**, but also how the vector database is configured and used. For example, the scanner could verify that the application implements proper multi-tenant isolation (e.g. row-level security or separate indexes per user) and input validation before data enters the vector store. As a rule of thumb, *no untrusted data* should be added to the knowledge base without screening – e.g. detecting hidden instructions or disallowing sensitive PII from being embedded [7] [8]. By flagging such design flaws, the tool helps developers fortify the retrieval layer, since "RAG is only as secure as its retrieval layer" [9].

Mechanics of a Python-Based Static Scanner for RAG

A static scanning tool works by **analyzing code, configuration, and other artifacts without executing the system**. In the context of a Python-based RAG application, this means parsing source code (and possibly YAML/JSON configs) to detect insecure patterns. The scanner might leverage Python's AST (Abstract Syntax Tree) to find how the LLM is used and how the vector database is accessed. Key mechanics and checks could include:

- **Prompt Construction Analysis:** Identify where user input is incorporated into prompts. If the code simply concatenates or formats user-provided text into the prompt with no sanitization or constraints, it's a red flag for **prompt injection** vulnerability [10]. The scanner can alert if, for example, system prompt strings or few-shot examples are not isolated from direct user content or if special tokens/escape sequences aren't handled.

- **Output Handling Checks:** Trace how the LLM's output is used. Static analysis can check if outputs are being rendered in a web page or executed as code/commands. If so, the tool should verify output is properly validated or escaped. (OWASP notes that failing to sanitize LLM outputs can lead to classic exploits like XSS, CSRF, SSRF, or even remote code execution in downstream systems [11].) The scanner could search for patterns like `exec(output)` or insertion of model text into HTML templates, warning the developer to implement stricter output filtering [11].

- **Vector DB Configuration:** Inspect how the app connects to and queries the vector store. The scanner might flag absence of authentication (e.g., if a vector DB URL has no token or uses default credentials), or lack of tenant separation (e.g., the code doesn't use user-specific namespaces or filters in queries). It could also parse SQL/NoSQL queries to ensure **row-level security (RLS)** or similar controls are in place when using a multi-tenant vector database. For example, if using pgVector, the tool might look for `CREATE POLICY` statements or query conditions that enforce `WHERE tenant_id = current_user` – if missing, it signals a risk of cross-tenant data leakage [12] [13].

- **Policy and Parameter Scanning:** The tool should also enforce organizational security policies. For instance, if policy dictates that **no hardcoded secrets or API keys** should exist in code, the

scanner will search for patterns resembling keys/tokens and ensure secrets are pulled from secure vaults or env variables instead (preventing accidental info disclosure) [14]. It can also scan configuration parameters against policy: e.g., checking that the LLM model being used is an approved version (using only vetted models to mitigate supply-chain risks [15]), or that certain settings (like model temperature, max tokens, etc.) are within safe bounds for the use-case. Even parameters like the context window length could be important – a scanner might warn if the application allows unbounded user input that could overflow the model's context window, an attack that can push out system instructions ("prompt overflow") [16] [17].

- **Dependency and Supply Chain Review:** Since many RAG apps rely on third-party libraries and pre-trained models, a static scanner should inventory these. It can cross-check Python dependencies for known vulnerabilities (using databases of CVEs) and ensure models or model weights come from trusted sources. Using an outdated or unmaintained model is highlighted by OWASP as a risk [18], so the tool could maintain a list of recommended model versions. Additionally, if the code downloads model files or calls external APIs, the scanner can prompt the developer to verify the integrity and authenticity of those resources (mitigating *supply chain vulnerabilities* in the LLM pipeline [15]).

- **Secure Coding Patterns:** The scanner may also incorporate general Python security scans (akin to Bandit or linters) to catch issues like use of `eval()`, weak cryptography settings, lack of error handling, etc. While not specific to LLMs, these could impact the overall security of the application. For example, if the code catches an exception and prints it back to the user or logs it verbosely, it might inadvertently leak a system prompt or internal implementation detail (related to **system prompt leakage** risk [19]). The static tool would flag overly verbose error messages or `print()` statements that expose internal context.

Importantly, a static scanner for RAG should be continuously updated with new vulnerability patterns. The field of LLM security is evolving quickly, and hard-coded rules can become outdated. To stay current, the tool might even leverage **RAG techniques itself** – for instance, retrieving the latest known exploit patterns or OWASP recommendations from a vector store of security knowledge and incorporating those into its analysis logic. Research has shown that integrating **LLMs and retrieval** with traditional static analysis can catch more complex or emerging vulnerabilities than rules alone [20] [21]. By combining deterministic checks with AI-based analysis (in a controlled, offline manner for privacy), such a scanner can provide robust coverage without "hallucinating" its own issues. In practice, this means the scanner could use an internal LLM (not a third-party API to avoid leaking code) to reason about code segments – for example, to detect subtle logic that could lead to prompt injection – but always with references to known secure coding guidelines or vulnerability examples (retrieved from its local knowledge base). This approach aligns with the idea of *"augmenting static analysis with AI"* in a reliable way [22] [23], ensuring the scanner's findings are explainable and based on authoritative patterns rather than guesswork.

Common Vulnerabilities in RAG/LLM Applications (OWASP Top 10 – 2025)

The **OWASP Top 10 for LLM Applications (2025)** is an authoritative list of the most critical risk areas for AI apps. It provides a roadmap for what our static scanner should look for. Below we summarize these top risks and how a scanning tool addresses them:

- **1. Prompt Injection:** *The scanner must detect places where untrusted input can manipulate prompts.* Prompt injection occurs when user input alters the model's behavior or bypasses its

instructions [10]. For example, a malicious user might input `"Ignore previous instructions and ..."` to trick the system. The scanner should flag any code that directly feeds raw user text into high-privilege prompt sections (system or developer prompts) without filters. It can also suggest mitigations like input validation or prompt segmentation to isolate user queries [24]. This category includes detecting attempts at *jailbreaking* (inputs crafted to evade safety filters) [25].

- **2. Sensitive Information Disclosure:** *Ensure no secrets are exposed and personal data is handled properly.* Our tool should search for hardcoded credentials, API keys, or any secrets in code (which could be leaked if the model reveals them or if code is public) [14]. It also examines prompts or system messages that might contain sensitive business logic or data. If the LLM is allowed to access confidential info (or if prompts inadvertently echo internal data), the scanner would warn about potential data leakage. OWASP recommends **sanitizing training data and inputs** to remove sensitive content [14] – something a proactive scan of the knowledge base can attempt by searching for PII patterns. In short, the scanner enforces the rule "don't put anything into the model (prompt or embeddings) that you aren't okay with possibly coming out" [14].

- **3. Supply Chain (Model and Libraries):** *Verify the integrity and trust of all components.* The scanner will list all third-party models and packages used and cross-reference known vulnerabilities. If a model is loaded from an unverified source or an open model hub, the tool might flag it for manual review (as open-access models might have backdoors or hidden behaviors [15]). It will also highlight if the application relies on an outdated LLM version that is no longer getting security updates [18]. This extends to checking how model updates are applied – e.g., warning if auto-update is disabled or if fine-tuning data sources aren't validated. By treating the model and data pipeline as part of the *"software supply chain"*, the scanner helps prevent issues like malicious model weights or tampered datasets entering the system [15].

- **4. Data and Model Poisoning:** *Identify any untrusted data flows into model training or vector stores.* The scanner can't directly see an ongoing poisoning attack, but it can flag design choices that allow poisoning. For example, if any user can upload content that gets embedded for retrieval, the tool should warn that this could be abused to poison the knowledge base [26]. It might suggest requiring approvals for data ingestion or scanning new data for anomalies. Similarly, if the code performs on-the-fly fine-tuning or continual learning from user data, the scanner ensures there are authenticity checks and the ability to rollback model changes. Poisoning could introduce biases or hidden triggers ("backdoors") in the model [27] [28], so static analysis will encourage robust data provenance and verification steps.

- **5. Improper Output Handling:** *Check that the application treats LLM outputs with the same caution as any user-supplied content.* Our static tool looks at every point where model output is used. For instance, if the output is incorporated into an HTML page, it should be encoded to prevent XSS [11]. If the output is used to make a system call or database query, it must be sanitized to avoid injection in those contexts. The scanner can simulate scenarios (statically) by recognizing dangerous patterns – like constructing an OS command string with parts coming from the model – and then flag with warnings about command injection or privilege escalation risks [29]. Essentially, it enforces that LLM output cannot be blindly trusted. This is particularly crucial if the app has **excessive agency** (the next point): e.g. autonomous agents that take actions based on LLM decisions must have strict output vetting.

- **6. Excessive Agency:** *Limit an AI agent's freedom and privileges.* "Excessive agency" refers to giving an LLM-based system too much autonomy or access, such as the ability to execute functions, file operations, or external API calls without proper limits [30]. The static scanner

should detect integration points where the LLM can invoke tools or code (for example, a function-calling mechanism or plugins). It will then assess if the code imposes restrictions on those actions. Are there permission checks or sandboxing when the LLM tries to use an OS command? Is there a whitelist of allowed functions? If the scanner finds code like `exec(generated_code)` or an unrestricted plugin interface, it will raise an alert. This ties back to classic security principles – least privilege and whitelisting. An LLM agent shouldn't have open-ended access to the system. The scanner can also ensure **rate limiting** and usage monitoring are in place (to prevent the AI agent from overusing resources or making too many high-privilege calls, which connects to Unbounded Consumption below) [31].

- **7. System Prompt Leakage:** *Protect the hidden instructions and configurations guiding the AI.* LLM applications often have a system prompt or hidden context that defines behavior (like "You are a helpful assistant…" plus possibly API keys or DB connection info embedded). If attackers can trick the model into revealing these, it's a serious breach [19]. The scanner can't run the prompt, but it can check for risky clues: e.g., if the system prompt text is constructed from environment variables or contains sensitive tokens (like `<API_KEY>`), it should ensure the model is not allowed to output that. It could flag any occurrence of likely credentials in prompt strings. Moreover, it will warn if the app lacks a check for queries like "What are the system's instructions?" – a common way to test for leakage. OWASP notes that even if not fully disclosed, an attacker might infer system instructions by probing behavior [32]. Therefore, the scanner might also encourage **response monitoring**: code that inspects outputs for signs of revealing internal info. If such monitoring is absent, the tool might suggest adding it. In summary, this category pushes developers to keep system prompts truly hidden and free of secrets.

- **8. Vector Database and Embedding Weaknesses:** *Harden the retrieval layer.* We discussed many of these in the RAG section – the scanner should ensure strong **access control** on the vector DB (no open ports or default creds, and proper query filtering per user) [4] [13]. It also checks that embeddings are handled securely: for example, if the code does any custom embedding, is it using a proven library (to avoid flaws in embedding algorithms)? Another aspect is **encryption**: the scanner could note if the vector store is configured to encrypt data at rest or if embeddings containing sensitive info are encrypted or anonymized. *Embedding inversion* risk means sensitive data shouldn't be stored in vector form without considering encryption or dimensionality reduction to make inversion harder [5]. Additionally, the scanner might ensure that when combining data from multiple sources, the app tags data with origin or classification – preventing contradictory or inappropriate mixes (as recommended by OWASP to avoid "federation knowledge conflicts" between data sources) [33]. If the static analysis finds no evidence of such labeling or separation, it alerts that combined context could inadvertently leak info across boundaries.

- **9. Misinformation and Overreliance:** *Detect where the system might propagate incorrect outputs without checks.* An LLM can confidently produce wrong answers (hallucinations). OWASP flags this as a vulnerability – if users or systems **overly trust the AI's output** without validation, it can cause real harm [34]. A static scanner can't fact-check, but it can identify critical decision points in code. For example, if the application uses the LLM's answer directly in an automated decision (like approving a transaction or diagnosing a patient), the scanner should question that. It might recommend adding verification steps or human review for high-stakes usage. This aligns with the idea of *"human in the loop"* for critical tasks. The scanner can also encourage enabling any available **confidence scores or citations** from the model. If the code doesn't log model uncertainty or provide traceability (like which documents were retrieved in RAG), that's a weakness. Essentially, the tool reminds developers that an AI's response quality must be

monitored – e.g., by logging when the model refuses or when it's unsure – to avoid blindly relying on output that could be false or biased [34].

- **10. Unbounded Resource Consumption (Denial-of-Service and "Model Exhaustion"):** *Make sure the app has usage limits and cost controls.* LLMs can be abused by malicious or inadvertent heavy usage – sending huge prompts, very long conversations, or rapid-fire queries can lead to denial of service or exorbitant costs (sometimes dubbed "Denial of Wallet"). The static scanner will look for mitigations like **rate limiting, payload size limits, and timeouts**. If the code accepts user input, is there a max length enforced? If the system streams responses, is there a safeguard to cut off overly long outputs? OWASP notes that uncontrolled usage can lead to model theft (extracting the model by repeated queries) or just degrade service [35]. Thus, the scanner might also check that the API calls to the model have sensible settings – e.g., limiting tokens per response, and that the application monitors for abuse patterns [35]. It could flag missing controls by scanning for usage of any rate limiter library or absence of logic that tracks user request counts. Additionally, static analysis might ensure the app handles failures gracefully (to avoid infinite loops retriggering the model). This prevents scenarios where an attacker intentionally causes errors that prompt the system to retry endlessly. By enforcing bounded use of the model, the scanner helps avoid both DoS and unexpected bills or model exposure.

Each of these OWASP categories comes with recommended mitigations, and a robust static scanner will effectively serve as a "checklist enforcer" for those mitigations. For example, it can prompt developers: *"Are you validating and sanitizing all user inputs to the LLM?"* [36], *"Do you have logging and monitoring in place for LLM interactions?"* [37], *"Is access to the AI feature properly authenticated and authorized?"* [38], and so on. By aligning scanner rules with the OWASP Top 10, we ensure the tool is covering the most critical risks known in 2025 for AI applications.

Alignment with NIST AI Risk Management Framework (AI RMF)

While OWASP Top 10 focuses on specific technical vulnerabilities, the **NIST AI RMF** provides a broader *risk management* perspective for AI systems. It outlines a structured approach with four core functions – **Govern, Map, Measure, and Manage** – to help organizations build trustworthy AI [39]. Our static scanning tool can play a key role in the **"Measure" and "Manage"** aspects by identifying technical risks and verifying that mitigations are in place, thus feeding into the overall risk management process.

NIST's Trustworthiness Criteria: NIST emphasizes characteristics of *"trustworthy AI"* such as being *valid and reliable, safe, secure and resilient, accountable & transparent, explainable & interpretable, privacy-enhanced, and fair* [40]. A static scanner contributes to several of these:

- **Secure and Resilient:** By detecting vulnerabilities (like those in OWASP Top 10) and weak configurations, the scanner helps ensure the AI system is secure against attacks and resilient to misuse. For example, flagging missing input validation or lack of access control directly improves the *safety* and *security* of the system. This aligns with the NIST RMF goal of proactively identifying and mitigating security risks in AI (part of the *Manage* function) [41].

- **Accountable and Transparent:** NIST RMF encourages logging and traceability for AI decisions. Our tool can check that the code implements proper logging of model interactions and outputs. It might also verify that there's a mechanism to trace which data or prompts led to a given decision (important for audits and accountability). If, say, the scanner finds no logging around the LLM API calls or vector DB queries, it highlights a transparency gap. NIST's *Govern* function calls for policies and roles to manage AI risk [42] – a static scanner can enforce some of those

policies automatically (e.g. “all prompts must be logged except sensitive ones”, or “developers must document the data sources used in RAG”). By integrating policy templates (like those aligned with NIST) into the scanner’s rules, we ensure that governance requirements translate into concrete checks in code [43].

- **Privacy-Enhanced:** Privacy is a big part of trustworthiness. The scanner should thus identify any personal data usage and ensure it’s handled per privacy guidelines (for instance, warning if user inputs are stored indefinitely or if outputs could contain private data without a scrubbing mechanism). NIST RMF would have organizations map out privacy risks and address them [44] [40] – the scanner aids this by pointing out places where personal data might leak or lack protection (like unredacted PII in a knowledge base, which we mentioned earlier). In doing so, the tool supports compliance with privacy principles and even regulations (by ensuring things like GDPR-related data handling are considered in the code).

- **Explainable & Interpretable / Fairness:** These aspects are trickier to assess via static code. However, the scanner can at least encourage documentation and use of model cards or bias evaluation results. For instance, if the project has a model selection step, the tool could remind the developer to refer to the model’s known limitations or bias mitigations (maybe via comments or config). It might not actively measure fairness, but by enforcing that certain checkpoints or notes are present (e.g., “verify the dataset covers diverse groups – see NIST guidance”), it nudges the development towards those best practices. In a broader DevSecOps pipeline, static scanning results could be combined with additional analysis (like bias testing tools) to satisfy NIST’s *Measure* function (which calls for measuring AI risks like fairness, robustness, etc.) [45].

Framework Integration: In practice, aligning with NIST RMF means our scanner’s findings should feed into an organization’s risk register and mitigation tracking. For example, if the scanner flags a **prompt injection risk**, that maps to a risk of *“unreliable or unsafe output”* in NIST terms, with a certain severity. Under the *Manage* function, the organization would decide how to treat that risk – perhaps by implementing a new guardrail. Our scanner would then be used to verify that guardrail code is present, creating a feedback loop for continuous risk management. NIST’s *Govern* function also stresses having the right processes and training [46] – a static tool can be a form of automated process ensuring developers consistently check code against security standards. It effectively acts as a *“digital policy enforcement”* mechanism that ensures AI development teams adhere to the policies derived from frameworks like NIST. If NIST RMF says *“establish organizational policies for AI security”*, the static scanner can encode those policies as rules. This way, compliance is not just a one-time checklist but an ongoing automated check in CI/CD.

One concrete example: NIST recommends *“documenting and tracking AI risks over time and having a plan to update mitigations”* [41]. Our scanner could output a report (JSON/HTML) of all issues found (prompt injection possibilities, missing encryption, etc.), which can be stored and tracked release by release. Over time, this provides an audit trail showing how risks were identified and fixed – aligning with NIST’s call for accountability and iterative risk management [47]. By integrating such scanning reports with governance tools (like dashboards or even specialized platforms like Trustible or SD Elements), an organization can demonstrate adherence to frameworks in a tangible, developer-friendly way [48] [49].

In summary, the static scanner is a crucial tactical tool that supports the strategic objectives of NIST’s AI RMF. It translates high-level principles (like “AI systems should be safe, secure, and transparent” [40]) into specific actionable checks in code. This not only hardens the application but also provides evidence of risk management – useful for internal audits, compliance, and building trust with users and regulators.

Other Relevant Frameworks and Best Practices

Beyond OWASP and NIST, there are additional frameworks and guidelines that inform a RAG security scanner's development:

- **EU AI Act (Draft Regulation):** The forthcoming European AI Act defines **risk categories for AI systems** (minimal, limited, high, unacceptable) and imposes requirements on high-risk AI (like documentation, transparency, and risk controls). While not a vulnerability standard, it underscores certain practices our scanner should encourage – e.g. *data governance*, *record-keeping*, *transparency to users*, and *robustness* for high-risk systems. If our Python scanner is used in a product that might fall under "high-risk" (like AI in healthcare or finance), it should help ensure compliance by flagging missing notices or lack of human oversight in sensitive decisions. In essence, the EU AI Act sets expectations (legal obligations) and the scanner can assist by checking some technical aspects of those (for instance, verifying that the system can provide an explanation or that it has an interface for human intervention, etc.). Keep in mind the Act is broad; the static scanner addresses mainly the security and technical robustness parts, which still contributes to fulfilling some requirements.

- **ENISA's AI Cybersecurity Guidance:** The European cybersecurity agency (ENISA) has published guidance and a threat taxonomy for AI/ML systems. They highlight stages of the ML pipeline where threats manifest and recommend **controls at each stage** (data, model, infrastructure). Our scanner aligns well with these recommendations by examining the code for those controls. For example, ENISA (and similarly the UK NCSC) emphasize *"never trust user inputs"* and *"secure the data supply chain"* in ML – which translate to many of the checks we described (input sanitization, integrity checks on data, etc.). ENISA also points out the need for **adversarial testing** of AI; while static analysis is pre-emptive, we can integrate its use with dynamic testing. For instance, after running the static scan, teams should also run red-team tests (like feeding tricky inputs to the model) to catch anything static analysis might miss. Our focus is static, but acknowledging this best practice is important – defense in depth means using multiple methods. In fact, some tools (e.g. Promptfoo's red teaming toolkit) can complement static scans by simulating attacks like prompt injection and verifying if they succeed [50] [51]. A well-rounded approach uses the static scanner to enforce secure coding *by design*, and dynamic tests to validate at runtime.

- **Adversarial ML Threat Matrix (MITRE ATLAS):** This is a framework categorizing attacks on AI (ranging from data poisoning to model evasion and extraction). It's more for understanding threats than directly guiding development, but we can use it as a reference for what our scanner should cover. For instance, MITRE's matrix includes "Model Theft via Query" – which maps to OWASP's unbounded consumption issue; "Data Poisoning" and "Model Poisoning" – which we handle via checks on training and input; "Evasion" attacks – relevant to output handling and filtering. By ensuring each category of attack in the threat matrix has some preventive control that the static tool checks for, we improve coverage. If an item in the matrix has no corresponding check in our scanner, that might highlight a gap (for example, adversarial input that causes misclassification might be mitigated by input preprocessing – is our scanner checking for use of such preprocessors or at least the ability to add them? These are advanced considerations perhaps, but worth noting in design).

- **Secure Software Development Lifecycle (SSDLC) and DevSecOps Practices:** Our scanner should integrate into the development workflow (IDE and CI/CD) so that issues are caught early (shift-left security). This echoes general secure coding frameworks (like OWASP SAMM or

Microsoft SDL) which advocate for automated code review tools. Because AI systems iterate quickly, automating these scans on every pull request or build is crucial. It's also recommended to have **policy as code** – essentially, the security rules (derived from OWASP, NIST, etc.) are encoded in our tool's config. That means if tomorrow OWASP updates its guidance, we update the scanner rules accordingly and all new code is checked against them. Security Compass's platform example shows mapping requirements from frameworks like OWASP LLM Top 10 and NIST RMF into developers' workflow [43] [52] – our tool is a lightweight version of that idea, purpose-built for static analysis of RAG apps.

- **Best Practices for Vector DB Security:** In addition to high-level frameworks, there are domain-specific best practices (like the Medium article on implementing row-level security in vector DBs). We incorporate those as rules or guidelines: e.g., **layered security** (the scanner checks that both DB-level and app-level access controls exist) [13], **audit logging** (ensure the code logs access attempts to the vector store) [13], **parameterized policies** (prefer config-driven access rules rather than hard-coding roles, which the scanner could partially verify by seeing if policies are read from config) [13], **query anonymization** (if the app might log or share queries, ensure it's done in a privacy-preserving way) [53], and **secure container deployment** if using Docker (the scanner might not fully analyze Dockerfiles, but we could extend it to check for common issues like running container as root, not updating base images, etc., as those indirectly affect security of the RAG system) [54]. By embedding these best practices, the tool becomes more than a vulnerability scanner – it's also a *configuration/policy scanner* ensuring the system is set up securely end-to-end.

In summary, many overlapping efforts (regulatory, standards, and community best practices) are converging on making AI secure and trustworthy. Our static scanner is informed by all of them: **OWASP's list gives concrete app vulnerabilities to check**, **NIST RMF gives a risk management and governance context**, and frameworks like **EU AI Act or ENISA guidance emphasize compliance and holistic security measures**. It's important to understand the scope of each: OWASP Top 10 is very technical and dev-focused, whereas NIST and EU AI Act are broader (process and legal focus). A **comparison** is outlined below to clarify their differences and how a tool might address each:

Comparison of Key Frameworks and Approaches

- **OWASP Top 10 for LLM (2025):** *Focus:* Technical vulnerabilities unique to LLM and GenAI applications. It enumerates issues like prompt injection, data leakage, etc., with examples and mitigations. *Use for Scanner:* Provides direct guidance on what to scan for in code and configuration (essentially forms our rule set). It's highly actionable for developers – e.g., if OWASP says "prompt injection" is #1, the scanner can implement specific checks for that. However, OWASP Top 10 doesn't cover governance or ethics; it's narrowly about security failures to avoid. **Comparison:** Compared to others, OWASP is granular and dev-centric – it "rarely tells development teams exactly what to do" but highlights where to look [55]. Our scanner bridges that gap by turning OWASP's listed risks into explicit "to-do" items for the developer.

- **NIST AI RMF:** *Focus:* Comprehensive risk management and trustworthiness of AI systems. It's voluntary guidance that covers organizational process (Govern), context understanding (Map), risk analysis (Measure), and mitigation/monitoring (Manage) [39]. *Use for Scanner:* Sets overarching goals (like AI should be secure, explainable, fair, etc.) which the scanner supports by enforcing security and logging (for security, transparency) and pointing out issues that need risk treatment. NIST is less about specific code flaws and more about having a structured approach to identify and mitigate risks over the AI lifecycle [56] [57]. **Comparison:** NIST RMF differs in level

– it's high-level and not specific to LLMs. Where OWASP might say "sanitize outputs to prevent XSS" (technical detail), NIST would say "ensure AI systems are safe and secure, and mechanisms are in place to handle incidents". Thus, NIST complements OWASP by covering areas like governance, bias, and overall risk process that OWASP doesn't detail. A successful AI security program will use OWASP Top 10 for concrete dev practices and NIST RMF to ensure those practices are part of a broader, continuous risk management strategy.

- **Other Frameworks (e.g. EU AI Act, ENISA, etc.):** *Focus:* Varies – EU AI Act is legal/regulatory, defining compliance requirements for AI, while ENISA provides security best practices and threat analysis. *Use for Scanner:* These inform certain checks or features. For instance, the EU AI Act would require high-risk AI systems to have record-keeping and transparency – so the scanner could remind to include features for that (like user-facing disclosure if applicable). ENISA's guidance, being security-focused, overlaps a lot with OWASP and general cybersecurity (e.g., secure communications, access control, data integrity). It might add emphasis on things like cryptography, secure deployment, and incident response plans. **Comparison:** The "other" category often covers areas not fully addressed by OWASP or NIST. For example, the EU Act touches on data quality and human oversight – things a static code tool might not enforce, but the dev team must plan for. ENISA might highlight upcoming threats (like adversarial attacks) that aren't explicitly in OWASP Top 10. So these frameworks expand the view: they ensure our scanner (and the team using it) aren't myopically fixing code bugs while ignoring compliance or architectural security. In practice, aligning with all of them is beneficial. As noted by one source, regulatory frameworks like EU AI Act, NIST RMF, and OWASP Top 10 "offer differing levels of granularity" [55] – our approach is to take the **most concrete parts of each** into account. OWASP guides the code-level fixes, NIST ensures we integrate those fixes into a risk program, and others like EU AI Act impose additional requirements (which our processes and possibly tooling must support, though not all can be automated by static analysis).

- **Static vs Dynamic Security Testing for AI:** It's also worth comparing scanning approaches. Static analysis (our tool) catches issues in code and config *before* the system runs – this is great for finding straightforward vulnerabilities and enforcing good practices early (fast feedback to developers). However, some AI-specific issues might only manifest at runtime under certain inputs. **Dynamic testing** (such as fuzzing the prompt or simulating attacks on a running model) can reveal vulnerabilities like prompt injection success or information leaks in context that static analysis predicted but didn't fully confirm. Ideally, the static scanner's findings are validated by dynamic tests. For example, if our tool warns "possible prompt injection vector here," a dynamic test can actually try a known malicious prompt to see if the model is misled. Conversely, dynamic tests might find an issue the static scan missed (maybe a subtle chain-of-thought exposure); that feedback can be used to improve the scanner's rules. Thus, these methods are complementary. The **advantage of static** is speed and integration into development (pre-deployment), while dynamic (red teaming, penetration testing) is crucial for a final verification and for catching context-dependent flaws. A comprehensive security strategy for RAG/LLM systems uses both – the static scanner as a **preventive checkpoint** in CI, and dynamic testing as a **validation and discovery** step before release and continuously in production.

To conclude this comparison: **OWASP Top 10 (2025)** gives us the checklist of "what can go wrong" in LLM apps and thus what to scan for. **NIST AI RMF** ensures we address those in a systematic, risk-managed way and consider broader trust factors, not just code bugs. **Other frameworks** like the EU AI Act remind us of compliance and ethical aspects, ensuring the tool's use fits into legal obligations and industry best practices. By understanding the strengths of each, we can develop our static scanning tool to not only find bugs, but to embed itself as a key component in an organization's secure and compliant AI development lifecycle.

Conclusion

Developing a static security scanning tool for RAG-based AI systems is an ambitious but crucial project. By leveraging Python to parse code and configurations, such a tool can **"shift left"** AI security – identifying issues from prompt injection to vector database misconfigurations long before they cause real harm. Our research emphasizes that the scanner's rule set should be informed by the latest industry knowledge: the OWASP Top 10 for LLMs (2025) gives a roadmap of vulnerabilities to cover, while frameworks like NIST's AI RMF ensure the tool's output feeds into a larger effort of building **trustworthy, risk-managed AI** [40] . In building this scanner, we must also remain adaptable. AI threats evolve quickly; what's a benign prompt today could be an exploit tomorrow. Therefore, the scanner should incorporate retrieval of up-to-date threat intelligence (perhaps using RAG techniques itself to update its checks) [20] , and it should be used in tandem with runtime testing and human review.

Ultimately, the goal is to empower development and DevOps teams to **move fast *and* stay secure**. A static scanner, integrated into CI/CD, will act like a specialized code review assistant – one that knows the mechanics of RAG and the pitfalls of LLMs, and tirelessly points them out. This reduces the burden on individual developers to recall every guideline or new risk area. Instead, they get immediate feedback and *actionable insights* (e.g., "Enable input filtering here to prevent prompt injection [58] " or "Implement row-level access control on your vector DB to prevent data leakage [4] "). By catching issues early, we not only prevent potential breaches or compliance failures, but also save on costly fixes down the line. Secure design and deployment of AI is a multi-faceted challenge, but with tools like this static scanner aligned to OWASP, NIST, and other best practices, we can significantly raise the baseline security posture of RAG applications. The result will be AI systems that organizations (and their users) can trust – systems that are robust against attacks, respectful of policies and privacy, and compliant with emerging AI regulations.

Sources:

- OWASP GenAI Security Project – *LLM Top 10 for 2025* (Prompt Injection, Data Leakage, etc.) [10] [59]
- OWASP GenAI Security Project – *Vector and Embedding Weaknesses* (RAG-specific risks and mitigations) [4] [5]
- Promptfoo.dev Blog – *"How Do You Secure RAG Applications?"* (Context injection, data poisoning, guardrail measures) [26] [24]
- Medium (M. Hannecke) – *Row-Level Security in Vector DBs for RAG* (multi-tenant access controls, audit logging best practices) [13] [60]
- Trend Micro – *Overview of OWASP Top 10 for LLMs 2025* (summaries of each risk and impacts) [11] [19]
- Security Compass – *"Securing the AI-Enabled Future of Products"* (mapping AI security standards to developer requirements) [55] [43]
- Trustible.ai – *NIST AI RMF Summary* (core functions and trustworthy AI characteristics) [40] [39]
- ArXiv (Keltek et al. 2024) – *LSAST: LLM-supported Static Analysis* (benefits of integrating LLM and retrieval to enhance vulnerability scanning) [20] [21]
- GitHub – *olegnazarov/rag-security-scanner* (example tool with features like prompt injection testing and data leakage checks) [61] [62]

---

[1] [3] [6] [8] [16] [17] [24] [26] [50] [51] [58] How Do You Secure RAG Applications? | Promptfoo
https://www.promptfoo.dev/blog/rag-architecture/

2 4 5 7 33 LLM08:2025 Vector and Embedding Weaknesses - OWASP Gen AI Security Project
https://genai.owasp.org/llmrisk/llm082025-vector-and-embedding-weaknesses/

9 Governance, Vector DB Security & LLMOps for Compliant GenAI
https://petronellatech.com/blog/securing-enterprise-rag-governance-vector-db-security-llmops-for-compliant-genai/

10 11 14 15 19 25 27 28 29 30 31 32 34 35 36 37 38 59 What are the OWASP Top 10 risks for LLMs? | Trend Micro (UK)
https://www.trendmicro.com/en_gb/what-is/ai/owasp-top-10.html

12 13 53 54 60 Row Level Security in Vector DBs for RAG | Medium
https://medium.com/@michael.hannecke/implementing-row-level-security-in-vector-dbs-for-rag-applications-fdbccb63d464

18 OWASP Top 10 for LLM Applications 2025
https://owasp.org/www-project-top-10-for-large-language-model-applications/assets/PDF/OWASP-Top-10-for-LLMs-v2025.pdf

20 21 22 LSAST: Enhancing Cybersecurity through LLM-supported Static Application Security Testing
https://arxiv.org/html/2409.15735v2

23 43 48 52 55 Securing the AI-Enabled Future of Products | Security Compass
https://www.securitycompass.com/blog/securing-ai-enabled-future-products/

39 Elements of the NIST AI RMF: What you need to know
https://www.holisticai.com/blog/nist-ai-rmf-core-elements

40 41 42 44 45 46 47 49 56 57 NIST AI RMF - Trustible
https://trustible.ai/nist-ai-rmf/

61 62 GitHub - olegnazarov/rag-security-scanner: RAG/LLM Security Scanner identifies critical vulnerabilities in AI-powered applications, including chatbots, virtual assistants, and knowledge retrieval systems.
https://github.com/olegnazarov/rag-security-scanner