How to Build Secure AI Applications

An SDLC Checklist

When I talk to SaaS teams about securing their AI features, the conversation almost always starts with the model. Can it be jailbroken? Can it leak data? Those are real questions, but they are not the whole picture. The bigger risk is the application *around* the model: what it can access, retrieve, trigger, and expose.

This checklist organizes testing by **when in the build process** each check belongs, across five layers: LLM API, RAG, MCP, Agents, and AI-assisted dev tooling. Whatever layer you're testing, watch for the same three failures: **untrusted input reaches a privileged context**, **output is trusted too early**, and **authorization is checked in the wrong place**.

01 Design & Threat Modelling

- [ ] Classify what data the AI feature can reach (PII, PHI, financial records, source code, business logic) and label each as a potential disclosure path.
- [ ] Define which actions the system can *trigger*, not just what it can *say* — record updates, deletions, external emails, API calls, code execution.
- [ ] Establish the intended permission model: does tool/retrieval access map to the *requesting user's* permissions, or a shared service credential?
- [ ] For agentic features, document where human review is required for sensitive actions (deletes, external comms, permission changes, data exports).
- [ ] Define logging requirements up front: can your team reconstruct what the AI read, decided, and called?

02 Development

LLM API Layer

- [ ] Input handling separates user-supplied content from system/developer instructions (so user input can't override the system prompt).
- [ ] Model output is treated as untrusted before it's rendered, stored, or passed downstream — no raw HTML/Markdown injection, no unsafe links, no unsanitized output into Slack/Teams/PDF/email.
- [ ] Output flowing into downstream functions (API calls, DB writes, command execution) is validated and parameterized.
- [ ] Token usage and call frequency are bounded to prevent cost-based denial of service.

RAG Layer

- [ ] Retrieval is scoped to the requesting user's permissions — a user from one account cannot retrieve another tenant's documents.
- [ ] Retrieved content is treated as untrusted: instructions hidden inside documents, PDFs, tickets, or KB articles should not change model behavior.
- [ ] Responses don't leak retrieval metadata (document names, author details, internal IDs, file paths, source references).
- [ ] Index updates propagate deletions/restrictions — when a source doc is removed, the AI can no longer answer from it.

MCP / Tools Layer

- [ ] Tool execution is tied to the user's permissions, not only the MCP server's credentials.
- [ ] Tools validate file paths, command arguments, API parameters, and all user-controlled inputs.
- [ ] Dangerous tools (file write, shell execution, DB access, privileged API calls) have scopes, rate limits, and approval gates.
- [ ] Tool descriptions and results can't manipulate the model into unsafe follow-up actions.

Dev Tooling Layer (SDLC Controls)

- [ ] AI-generated code goes through the same review gates as human-written code.
- [ ] Reviewers specifically check authorization, tenant isolation, secrets handling, payment/billing logic, and admin functionality in generated code.
- [ ] Secrets, API keys, tokens, and env values are not exposed in generated code, logs, public assets, or client-side bundles.

03 Pre-Deployment Testing

LLM API

- [ ] Attempt prompt injection: override the system prompt, extract hidden instructions, bypass guardrails.
- [ ] Test every output destination separately — UI, Slack/Teams, email, PDF export, DB, downstream API — for XSS, injection, and unsafe rendering.

RAG

- [ ] Attempt cross-tenant retrieval and indirect/second-order injection via planted content in documents, tickets, or emails.
- [ ] Test whether an attacker can rank their content as 'most relevant' to poison answers.

MCP

- [ ] Attempt prompt-injection-driven invocation of destructive or high-privilege tools.
- [ ] Test tool argument validation with malicious paths, injected arguments, and out-of-scope parameters.

Agents

- [ ] Test whether content the agent reads (ticket, doc, email, customer message) can silently change its goal.
- [ ] Test second-order privilege escalation: low-privileged user plants content → higher-privileged agent acts on it.
- [ ] Verify permission is re-checked on every tool call, not just at session start.
- [ ] Test agent-to-agent handoff — can one agent delegate to another with more access or more powerful tools?
- [ ] Confirm sensitive actions actually hit a human-review gate before executing.

Dev Tooling

- [ ] Probe AI-generated routes/APIs/admin functions for missing authentication.
- [ ] Test tenant isolation on the backend (not just hidden in the frontend) and verify DB queries are scoped to the current user/tenant.
- [ ] Hunt for authorization logic that looks right but fails on edge cases, role changes, or direct API access.

04 Deployment

- [ ] Authentication and authorization are enforced on all LLM-backed endpoints.
- [ ] Guardrails, content filters, and output sanitizers validated in the production environment (config drift between staging and prod is common).
- [ ] Rate limits and token/cost caps are active on live endpoints.
- [ ] Audit logging for tool execution and agent actions is on and capturing enough to reconstruct an incident.
- [ ] Human-in-the-loop approval gates for sensitive actions are enabled in production config, not just present in code.

05 Runtime & Monitoring

- [ ] Monitor for abuse patterns: injection attempts, token-inflation, anomalous tool-call volume.
- [ ] Alert on cost spikes (denial-of-wallet signal).
- [ ] Detect retrieval drift — newly added/changed source documents that introduce poisoned or over-permissioned content.
- [ ] Maintain an incident-response path specific to AI: reconstruct what the agent read, decided, called, and why.
- [ ] Re-test after material changes to prompts, tools, MCP servers, or agent workflows.

The Common Thread

■ **Did untrusted input reach a privileged context?**
A user prompt reaching the system layer, a planted document reaching the model context, a user-level request reaching admin-scoped tool credentials.

■ **Was output trusted too early?**
Model output driving actions or rendering without validation; generated code executed without review; retrieved content used without authorization checks.

■ **Was authorization checked in the wrong place?**
UI-layer controls that miss the retrieval layer; session-level checks that don't apply at tool invocation; user-level assumptions that ignore service-credential scope.

Test the full stack, not just the model.