USING AI TO BUILD CODE? READ THIS FIRST.

BUILD WITH INTENT. **VALIDATE WITH CONFIDENCE. DEPLOY WITH CLARITY.**
Generative AI has changed who can create software. Developers and non-developers alike can describe a workflow in plain language and produce a working service in minutes. Entire features can move from prompt to deployment faster than traditional security review cycles were designed to handle.

The security goal itself hasn't changed. What has changed is that AI-generated code doesn't trigger the same fail-safes that were built around human-written software, and model-level safeguards are not, by themselves, strong security controls. Mission-critical guardrails must be enforced outside the AI, at the infrastructure and workflow level.

Before releasing AI-developed software, use this security guardrails checklist to learn how to constrain generated code, enforce security controls, and prevent silent risk from prompt to production.

15 SECURITY GUARDRAILS TO IMPLEMENT

1. START FROM A HARDENED PROJECT TEMPLATE.

When AI generates code, it often "fills in the blanks" based on patterns learned from vast data sources, many of which may not adhere to secure coding practices. Templates enforce consistency, ensuring that AI-generated services are structured according to approved security and architectural standards.

What to do:

- ☐ Create opinionated templates for common service patterns.
- ☐ Lock in approved languages, frameworks, and pinned dependency versions.
- ☐ Bake in code standards and baseline security controls.
- ☐ Include example tests that fail when core flows are modified unsafely.
- ☐ Treat tests as constraints, not suggestions the model can rewrite.

2. CENTRALIZE AUTHENTICATION AND AUTHORIZATION.

AI-generated code may introduce inconsistencies in how authentication is handled, leading to fragmented and unreliable access control across services. Centralizing authentication through an approved library reduces complexity, improves security posture, and ensures that access is enforced uniformly.

What to do:

- ☐ Provide a shared authentication and authorization library.
- ☐ Centralize enforcement behind an API or access gateway when possible.
- ☐ Make the approved library mandatory.
- ☐ Configure CI to fail if required access controls are missing.

3. ENFORCE SECURITY EXPECTATIONS IN THE IDE AND CI PIPELINE.

Security issues should be detected and mitigated as early as possible in the development lifecycle. Failing fast in the IDE or CI pipeline helps catch vulnerabilities before they make it to production, reducing the risk of breaches or vulnerabilities.

What to do:

- ☐ Bake required middleware and defensive patterns into templates.
- ☐ Configure IDE warnings for missing security components.
- ☐ Reject pull requests that lack required controls.
- ☐ Prevent generated code from weakening or rewriting security tests.

4. MAKE HARDCODED SECRETS IMPOSSIBLE.

Hardcoded secrets (like API keys, credentials, or tokens) are one of the easiest attack vectors. Ensuring secrets are stored securely at runtime and not in the codebase drastically reduces the risk of credential leaks and unauthorized access.

What to do:

- ☐ Block secrets in source code through policy or automated scanning.
- ☐ Require runtime retrieval from approved secret stores.
- ☐ Provide helper utilities that make the secure pattern easier than the insecure one.

5. ROUTE ALL DATA ACCESS THROUGH APPROVED LAYERS.

By routing all data access through approved layers (e.g., SDKs, ORM frameworks), you minimize the risk of SQL injection, unauthorized access, or data manipulation. Allowing AI to write raw queries or API calls directly opens the door for attackers to inject malicious commands or gain unauthorized access.

What to do:

- ☐ Enforce database access through approved SDKs or repositories.
- ☐ Centralize parameterization and query construction.
- ☐ Restrict direct query execution from service logic.

6. DENY OUTBOUND TRAFFIC BY DEFAULT.

Unrestricted outbound traffic can leak data or be used to exfiltrate sensitive information without detection. By denying outbound traffic by default, you enforce a principle of least privilege that minimizes the risk of data leaks or unauthorized communication.

What to do:

- ☐ Configure firewalls to deny outbound traffic by default.
- ☐ Explicitly allowlist required domains and ports.
- ☐ Restrict DNS recursion where possible.
- ☐ Treat outbound permissions as intentional declarations.

7. BUILD IN RESOURCE AND RATE LIMITS

AI-generated code can easily create resource-intensive operations. Ensuring resource and rate limits are in place helps contain failures, protects systems from unnecessary strain, and ensures that AI-generated code doesn't overwhelm the infrastructure.

What to do:

- ☐ Enforce CPU and memory constraints.
- ☐ Configure execution timeouts.
- ☐ Add provider-aware rate limits for internal systems and SaaS APIs.
- ☐ Prevent retry storms and unbounded loops.

8. REQUIRE CI SECURITY GATES FOR ALL GENERATED CODE

Without security checks in the CI pipeline, AI-generated code could easily bypass established security measures. Integrating security directly into CI ensures that any code, generated by AI or not, meets the organization's security standards before it's ever merged.

What to do:

- ☐ Enforce linting, SAST, dependency scanning, and unit tests.
- ☐ Require documented exemptions for bypassing checks.
- ☐ Make risk acceptance explicit and visible.

9. MAKE SECURE HTTP DEFAULTS AUTOMATIC

Security should be the default, not an afterthought. Having secure HTTP defaults like TLS encryption, HTTP headers, and timeouts pre-configured in generated services protects data in transit and reduces the risk of man-in-the-middle (MitM) attacks.

What to do:

- ☐ Enable TLS by default where appropriate.
- ☐ Include security headers automatically.
- ☐ Configure sane timeouts.
- ☐ Centralize web server logging.

10. STANDARDIZE ERROR AND AUDIT LOGGING

AI-generated code can create services without consistent or useful error handling, which complicates troubleshooting and root-cause analysis. Standardizing logging improves visibility into what's happening, making it easier to detect and respond to security incidents quickly.

What to do:

- ☐ Provide a logging SDK that standardizes error and audit events.
- ☐ Correlate logs with trace IDs and user IDs.
- ☐ Ensure logs are structured and centralized.

11. SEPARATE ENVIRONMENTS AND AUTOMATE PROMOTION

Without a clear separation of environments, AI-generated code can be pushed directly into production without sufficient testing or review. Enforce a structured deployment process that prevents unintentional mistakes and protects production systems from untested code.

What to do:

- ☐ Clearly isolate development, staging, and production environments.
- ☐ Block direct production edits.
- ☐ Gate promotion through automated workflows.
- ☐ Include structured review steps for AI-assisted changes.

12. CENTRALIZE INPUT VALIDATION

Input validation is often overlooked or missed in AI-generated code, leaving services open to injection attacks or unexpected behavior. Ensuring that validation is centralized and consistently applied to all entry points drastically reduces the risk of attack.

What to do:

- ☐ Enforce strongly typed schemas at entry points.
- ☐ Reject unexpected input by default.
- ☐ Centralize validation libraries.

13. ENFORCE LEAST-PRIVILEGE ROLES

AI-generated services, including autonomous agents, should operate with narrowly scoped permissions. Access should be intentional, minimal, and continuously evaluated. Least privilege limits blast radius and prevents lateral movement.

What to do:

- ☐ Assign minimal permissions per service or agent identity.
- ☐ Prevent lateral movement through network and IAM constraints.
- ☐ Avoid shared credentials or overly broad roles.
- ☐ Treat AI agents as service identities; grant only necessary access and monitor continuously.

14. GOVERN YOUR PACKAGE REGISTRY

Allowing AI to generate dependencies without checking their source or version opens the door to supply chain risks. By controlling dependencies, enforcing versioning, and using a governed registry, you ensure that dependencies are trusted and secure.

What to do:

- ☐ Use an allowlisted package registry.
- ☐ Pin approved versions.
- ☐ Block arbitrary dependency installation from prompts.
- ☐ Monitor for supply-chain risk.

15. EMBED SECURITY GUIDANCE WHERE DECISIONS HAPPEN

Security guidance embedded inside templates and code patterns ensures the safest option is always the easiest one. Embedding security directly where developers are making decisions increases the chances that security is prioritized throughout the development process.

What to do:

- ☐ Include inline comments explaining required security patterns.
- ☐ Provide agent instructions that steer AI toward approved libraries.
- ☐ Keep short architecture guidance inside repositories.

FINAL REFLECTION

These guardrails are not about slowing development or limiting what AI can generate. They are about shaping the environment in which that generation occurs.

When authentication, secrets management, network boundaries, dependency choices, and deployment controls are predefined, the model is no longer responsible for making those decisions. The system itself defines the boundaries. In doing so, you reduce the amount of security context that AI must reason about and eliminate entire categories of variability before they can appear.

This is what disciplined AI-assisted development looks like: deterministic infrastructure, constrained execution paths, enforced validation, and clear promotion controls. The model produces business logic. The platform enforces structure.

Speed becomes sustainable when guardrails are structural. With human oversight and embedded controls in place, teams can scale AI-generated code without scaling unintended risk.

If you'd like help securing your AI-developed code, check out Bishop Fox's secure code reviews, threat modeling, and architecture assessments.

READY TO TEST YOUR DEFENSES?

ABOUT BISHOP FOX

Bishop Fox is the leading authority in offensive security, providing solutions ranging from continuous penetration testing, red teaming, and attack surface management to product, cloud, and application security assessments.

LEARN MORE AT BISHOPFOX.COM