TLP - CLEAR

enisa

EUROPEAN UNION AGENCY
FOR CYBERSECURITY

ENISA Technical Advisory on AI-assisted software development

SEPTEMBER 2026

About ENISA

The European Union Agency for Cybersecurity, ENISA, is the Union's agency dedicated to achieving a high common level of cybersecurity across Europe. Established in 2004 and strengthened by the EU Cybersecurity Act, the European Union Agency for Cybersecurity contributes to EU cyber policy, enhances the trustworthiness of ICT products, services and processes with cybersecurity certification schemes, cooperates with Member States and EU bodies, and helps Europe prepare for the cyber challenges of tomorrow. Through knowledge sharing, capacity building and awareness raising, the Agency works together with its key stakeholders to strengthen trust in the connected economy, to boost resilience of the Union's infrastructure, and, ultimately, to keep Europe's society and citizens digitally secure. More information about ENISA and its work can be found here: www.enisa.europa.eu.

CONTACT

To contact the authors, use product_security@enisa.europa.eu.
For media enquiries about this paper, use press@enisa.europa.eu.

AUTHORS

ENISA

LEGAL NOTICE

COPYRIGHT NOTICE

ISBN Number, DOI Number

Table of Contents

1. Scope and context

AI-assisted software development is now part of normal software engineering practice[1]. In this advisory, AI-assisted software development refers to the use of AI systems to support software development activities. This includes different tasks such as generating code, modifying code, explaining code, generating tests, suggesting or adding dependencies, writing configuration, and preparing pull requests.

These practices can improve productivity and generate functional code, but functional correctness should not be confused with security. AI-generated outputs may still contain significant vulnerabilities, especially if the main focus is on completing the requested task rather than satisfying secure by design requirements.

The purpose of this ENISA Technical Advisory is to describe the main components of AI-assisted software development, identify representative risks and threats, and provide a practical approach for integrating secure by design expectations into AI-assisted development activities. It considers how security requirements, constraints, review, verification and evidence can be included directly into AI assistant or agent instructions and processes. The advisory then examines reusable skills as one practical mechanism for making these expectations explicit and consistent. It is not intended to provide an exhaustive threat model or comprehensive catalogue of controls for AI systems, but rather to illustrate how existing secure development practices can be applied and made explicit in AI-assisted software development workflows.

This advisory forms part of ENISA's broader efforts in support of the Action Plan on Cybersecurity and Artificial Intelligence[2], by promoting the secure integration of AI-assisted software development and the application of secure by design practices. It is also relevant to secure by design approaches such as those reflected in the Cyber Resilience Act (CRA)[3]. The CRA requires cybersecurity to be considered throughout the product lifecycle, including during design, development, production, delivery and maintenance. Where AI-assisted software development is used in these activities, the same secure by design approach should be considered in how AI-assisted tasks are defined, guided, reviewed and verified. This advisory does not provide legal guidance or introduce a new compliance framework, but provides practical guidance for aligning AI-assisted development workflows with secure by design expectations. It does not represent a prescriptive implementation framework and its application should be adapted to the product's intended use, context and associated risks and to the applicable requirements.

1.1 Note on the current version of the document

<To-do> public consultation info

[1] AI, Software Development, Security, Tips, and the Future (Part 1), https://openssf.org/blog/2025/12/29/ai-software-development-security-tips-and-the-future-part-1/, Accessed on 01/09/2026.
[2] EU Action Plan on Cybersecurity and Artificial Intelligence https://digital-strategy.ec.europa.eu/en/library/eu-action-plan-cybersecurity-and-artificial-intelligence, Accessed on 01/09/2026.
[3] Regulation (EU) 2024/2847 - Cyber Resilience Act, https://eur-lex.europa.eu/eli/reg/2024/2847/oj/eng, Accessed on 01/09/2026.

AI-assisted software development is evolving rapidly. This advisory reflects the state of practice at the time of publication and may be revised as technologies and good practices develop.

2. AI-assisted software development workflow and architecture

The purpose of this section is to define the main components of the AI-assisted software development workflow. This will serve as a foundation when explaining the risks, threats, and eventually good practices related to AI-assisted software development in the subsequent sections.

2.1 Models, coding assistants, agents and tools

2.1.1 Models

This advisory refers to an AI model as the component in an AI-assisted software development system that generates or transforms outputs, such as text, code, explanations, tests, configuration, or tool and function calls based on instructions and available context. Such models are typically trained on large datasets that may include public code, documentation, examples, natural language text, and other sources. It is important to keep in mind that training data may contain both secure and insecure coding patterns. Therefore, secure output should not be assumed only because the model is deemed advanced or widely used.

Examples may include general-purpose AI models[4] with coding capabilities, as well as specialised coding models. Some coding models are made available as open-weight or locally deployable models, while other models are accessed primarily through hosted commercial services or coding assistants. The specific models available through AI-assisted development tools change frequently. However, this advisory focuses on the role of the model in the workflow rather than on a specific list of models.

The model is only one part of the typical AI-assisted software development workflow. Its output is shaped by specific inputs such as developer prompts, system instructions, tool configuration, and the context provided during the coding session. A model does not necessarily know the full project context or the organisation's security requirements. Therefore, models may reproduce common patterns without knowing whether they are appropriate for the specific product, threat model, regulatory context, or security requirement.

2.1.2 Coding assistants and agents

This advisory refers to coding assistants and agents as AI-enabled systems used to support software development activities. They may be integrated into an integrated development environment (IDE), browser, chat interface, command-line interface (CLI), development platform, repository, or continuous integration and continuous delivery (CI/CD) workflow.

Coding assistants typically support developers interactively. They may use inputs such as developer prompts, selected files, repository context, documentation, project instructions and previous

[4]"general-purpose AI system' means an AI system which is based on a general-purpose AI model and which has the capability to serve a variety of purposes, both for direct use as well as for integration in other AI systems" https://eur-lex.europa.eu/legal-content/EN/TXT/HTML/?uri=OJ:L_202401689, Accessed on 01/09/2026.

conversation history, and produce outputs such as code suggestions, explanations, tests, configuration changes, dependency recommendations or proposed patches.

Coding agents generally operate with a higher degree of autonomy and may be given a goal and then plan, execute and adjust a sequence of actions to complete it. Depending on their permissions and configuration, they may inspect repository content, propose or apply changes, run commands and tests, install or update dependencies, generate documentation, or prepare commits and pull requests. They may also operate within event-triggered workflows, for example in response to issues, pull requests or security alerts.

The distinction between coding assistants and coding agents is not always strict. The same system may support both interaction models depending on how it is configured or used.

2.1.3 Tools and integrations

AI-assisted software development systems can interact with development tools and external integrations. These can include build tools, test runners, package managers, security scanners, issue trackers, CI/CD platforms and documentation systems. The interaction with tools can occur through IDE integrations, command-line execution, APIs, plugins, CI/CD integrations or repository platform features. Assistants or agents may request information from a tool, invoke a command, receive the output, and use that output as context for the next step.

Emerging integration mechanisms, such as the Model Context Protocol[5] (MCP), can also be used to connect AI assistants or agents to external tools, data sources and development environments in a more standardised way. In some agentic systems, an agent harness[6] or orchestration layer may manage how the agent receives context, invokes tools, observes results and continues the task.

The available tools and integrations shape what the assistant or agent can do. They may allow the system to move beyond code generation by running tests, inspecting build errors, installing dependencies, preparing pull requests or invoking security tools such as static application security testing (SAST) or dependency scanners.

2.1.4 Context and project instructions

AI-assisted software development systems rely on context to produce relevant outputs. Context can include developer prompts, conversation history, selected files, repository content, dependency manifests, configuration files, documentation, error messages and project-specific instructions.

Project instructions are increasingly used to provide structure and consistency to the context made available to an assistant or agent. They can provide explicit rules or guidance on aspects such as coding conventions, architectural constraints, security requirements, testing expectations or documentation practices. Examples include repository-level instruction files[7], tool-specific rules and reusable skills[8][9].

[5] What is the Model Context Protocol (MCP)?, https://modelcontextprotocol.io/docs/getting-started/intro, Accessed on 01/09/2026.
[6] Your Harness Keeps It All Together, https://sites.duke.edu/ddmc/2026/04/30/your-harness-keeps-it-all-together, Accessed on 01/09/2026.
[7] AGENTS.md, https://agents.md/, Accessed on 01/09/2026.
[8] Agent Skills Overview, https://agentskills.io/home, Accessed on 01/09/2026.
[9] Claude Code Docs- Extend Claude with Skills, https://code.claude.com/docs/en/skills, Accessed on 01/09/2026.

The context and instructions available to the assistant or agent shape the output it produces. A main goal of this advisory is to show how security can be included in project instructions, with the aim of guiding assistants and agents to generate code and other outputs that are both functional and consistent with the intended security practices.

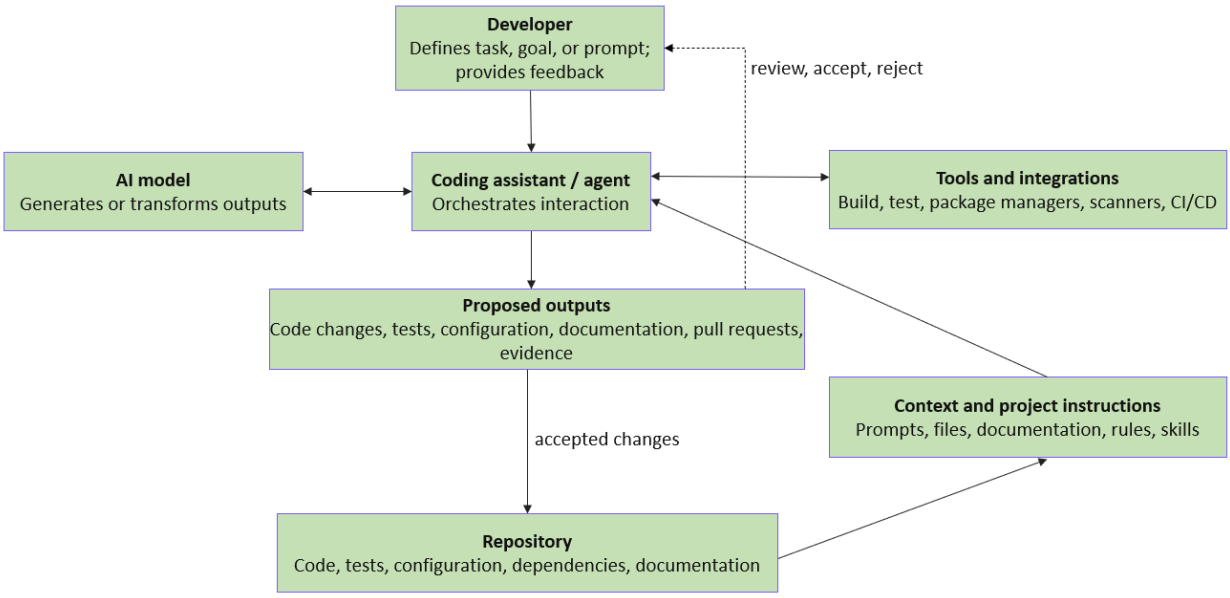


Figure 1: Components

2.2 AI-assisted software development across the product lifecycle

2.2.1 Common AI-assisted activities across the product lifecycle

AI-assisted development can be understood in relation to existing software development lifecycle activities. It does not necessarily replace software development activities but can rather support or modify how these activities are performed. The table below provides a non-exhaustive overview of common AI-assisted development activities and how they relate to product lifecycle processes.

Note: This advisory uses the product lifecycle activities aligned with ENISA's secure-by-design and default playbook[10].

Table 1: Common AI-assisted activities

AI-assisted activity	Description	Lifecycle mapping
Requirements clarification	Clarifying requirements, identifying ambiguities, drafting user stories or acceptance criteria.	Requirements
Design and architecture support	Exploring design options, summarising trade-offs, identifying components, interfaces or constraints.	Design

[10] ENISA Secure By Design and Default Playbook v1, https://www.enisa.europa.eu/sites/default/files/2026-07/ENISA_Secure_By_Design_and_Default_Playbook_v1.pdf, Accessed on 01/09/2026.

Threat modelling support	Identifying assets, entry points, abuse cases, trust boundaries or possible threat scenarios.	Requirements / Design / Verification
Implementation	Generating or modifying source code based on requirements, specifications or developer prompts.	Implementation
Porting	Adapting software to run in a different programming language, platform, OS or architecture while aiming to preserve the functionality.	Design/ Implementation / Maintenance
Refactoring	Restructuring existing code while aiming to preserve behaviour and improve maintainability.	Implementation / Maintenance
Test generation	Generating unit tests, integration tests, regression tests or test data.	Verification
Debugging	Explaining errors, analysing failures, suggesting fixes or helping identify root causes.	Verification / Maintenance
Dependency integration	Suggesting, adding, updating or replacing third-party packages or libraries.	Implementation / Maintenance
Configuration	Creating or modifying application, infrastructure, build, deployment or security configuration.	Implementation / Deployment
Documentation	Drafting or updating technical documentation, code comments, release notes or user guidance.	Across lifecycle processes
Pull request preparation	Preparing changes for review, generating summaries, explaining implementation choices or grouping changes.	Implementation / Verification
Review and remediation	Supporting code review, interpreting findings, suggesting fixes or helping address vulnerabilities.	Verification / Maintenance
Deployment support	Assisting with release preparation, CI/CD workflows, deployment scripts or readiness checks.	Deployment
Evidence generation	Summarising checks performed, linking changes to requirements, or preparing claim/evidence records.	Verification / Deployment / Maintenance

The aim of this document is not to assess whether, or to what extent, AI-assisted activities improve productivity or functionality across the software development lifecycle. Rather, it is to consider how AI-assisted activities can preserve, and potentially support, a product's security requirements across that lifecycle.

2.2.2 Forms of AI-assisted development

AI-assisted development can take different forms depending on the degree of autonomy given to the AI system. This advisory considers the following approaches, representing increasing levels of AI autonomy:

- **Developer-led assistance**: AI suggests, explains or generates limited code, while the developer remains closely involved in writing, selecting and integrating the output. Examples include autocomplete, code explanation, search, small snippets, or pair-programming assistance.
- **Agent-led development:** the developer delegates a bounded development task to an AI agent, which may plan and perform multiple actions to complete it. The developer remains actively involved in directing the task, as well as reviewing and verifying the resulting changes. The AI agent may inspect the repository, modify multiple files, run commands, install dependencies, generate tests, or prepare pull requests.

- **Goal-managed autonomous development:** the human defines higher-level goals, constraints and acceptance criteria rather than individual development tasks. The AI systems independently decompose those goals into tasks and perform substantial parts of planning, implementation, testing and remediation. In this approach, the human focuses more on governance, predefined approval points, and acceptance of the resulting changes.

The principles described in this advisory are relevant across these forms of AI-assisted development. However, more autonomous uses may require additional technical, organisational and operational controls beyond the scope of this advisory.

3. Risks, failure modes, and threats

AI-assisted software development introduces, or amplifies existing, security risks, failure modes and threats. This section does not aim to provide an exhaustive risk catalogue or a complete threat model for AI-assisted development. Rather, it highlights representative issues from different perspectives, including:

- **non-adversarial risks and failure modes**, where issues can occur without malicious intent;
- **adversarial threats**, where an attacker deliberately attacks the AI-assisted workflow;
- **governance and assurance risks**, where the organisation lacks sufficient control, accountability or evidence over AI-assisted development activities.

Together, these perspectives provide a practical lens for recognising what can go wrong across the AI-assisted activities described in Section 2.2, and provide context for the practical approach described in Section 4 for translating security expectations into AI-assisted development workflows.

3.1 Non-adversarial risks and failure modes

Non-adversarial risks arise even where no attacker is present. The table below summarises common non-adversarial risks and failure modes relevant to AI-assisted development.

Table 2: Non-adversarial risks and failure modes

Risk / failure mode	Example in AI-assisted development
Functional but insecure output	The generated code works for the requested task but lacks relevant security controls or introduces vulnerabilities[11] in the system.
Unintended actions or side effects	The assistant or agent performs an unintended action, such as deleting or overwriting code or other assets, exposing information to external services, or interacting with systems outside the intended scope.
Incomplete context, unclear instructions, or incorrect assumptions[12]	The assistant/agent lacks relevant project, architecture, threat model or security context and fills the gaps incorrectly.
Hallucinated[13][14][15] or outdated suggestions	The system suggests APIs, packages, commands, configurations or design patterns that do not exist, or are outdated.
Lack of understanding or review of generated changes	Developers rely on convincing code, explanations, pull request summaries or fixes without sufficient understanding or review. This risk may increase where AI-generated changes span many files or concerns, making review and verification harder.

[11] Is Vibe Coding Safe? Benchmarking Vulnerability of Agent-Generated Code in Real-World Tasks, https://arxiv.org/abs/2512.03262, Accessed on 01/09/2026.
[12] "You Still Have to Study" - On the Security of LLM Generated Code, https://arxiv.org/abs/2408.07106, Accessed on 01/09/2026.
[13] The Range Shrinks, the Threat Remains: Re-evaluating LLM Package Hallucinations on the 2026 Frontier-Model Cohort, https://arxiv.org/abs/2605.17062, Accessed on 01/09/2026.
[14] We Have a Package for You! A Comprehensive Analysis of Package Hallucinations by Code Generating LLMs, https://www.usenix.org/conference/usenixsecurity25/presentation/spracklen, Accessed on 01/09/2026.
[15] LLM Hallucinations in Practical Code Generation: Phenomena, Mechanism, and Mitigation, https://arxiv.org/abs/2409.20550, Accessed on 01/09/2026.

Weak tests, review or evidence	Generated tests, review notes or evidence are superficial, non-existent, incomplete, or not linked to security requirements.

3.2 Adversarial threats

Adversarial threats refer to scenarios where an attacker deliberately attempts to manipulate or abuse the AI-assisted software development workflow. In this advisory, STRIDE[16] is used as a lightweight structure for considering adversarial threats across the interactions shown in Figure 1. These interactions include the developer, assistant or agent, model, repository, context, tools and integrations, and proposed outputs. The examples below are illustrative and may overlap across STRIDE categories.

Table 3: Adversarial Threats

STRIDE category	Example in AI-assisted software development
Spoofing	An attacker causes a malicious package, tool, model[17], repository source, instruction file, skill, assistant or agent to be mistaken for a trusted one, or impersonates a trusted maintainer, model provider or service identity, causing the development environment or developer to accept and rely on it.
Tampering	An attacker modifies or inserts malicious content in repository files[18], project instructions, prompts[19], skills, model artefacts[17], or model-provider configurations or endpoints, causing an assistant or agent to consume manipulated context and generate or apply insecure changes.
Repudiation	A malicious user or compromised component performs actions through an AI-assisted workflow in a manner that cannot later be reliably attributed because prompts, tool actions, approvals or generated changes are not properly logged or protected[20].
Information disclosure	An attacker uses malicious instructions[21] or repository content to force the assistant or agent to expose secrets, proprietary code or other sensitive information.
Denial of service	An attacker manipulates inputs or context so that an agent performs disruptive or resource-intensive actions, e.g. repeatedly triggering builds or corrupting build configuration.
Elevation of privilege	An attacker exploits the permissions available to an assistant, agent or connected tool to cause privileged actions such as accessing secrets or modifying CI/CD configuration[22].

It is important to note that models, skills, and project instructions should be treated as security-relevant inputs and subject to appropriate controls, such as obtaining them from trusted sources, reviewing and approving them before use, and periodically reviewing and updating them. If they are malicious[23],

[16] OWASP Threat Modeling Process - STRIDE, https://cheatsheetseries.owasp.org/cheatsheets/Threat_Modeling_Cheat_Sheet.html#threat-identification, Accessed on 01/09/2026.
[17] The Risk of Fine-Tuned Open-Weight Models, https://www.msecops.de/blog/posts/backdoored-llms/, Accessed on 01/09/2026.
[18] Cursor 0day: When Full Disclosure Becomes the Only Protection Left, https://mindgard.ai/blog/cursor-0day-when-full-disclosure-becomes-the-only-protection-left, Accessed on 01/09/2026.
[19] Clinejection: Prompt Injection in GitHub Issue Titles Enables CI/CD Cache Poisoning and Supply Chain Compromise, https://labs.cloudsecurityalliance.org/research/csa-research-note-clinejection-prompt-injection-cicd-cache-p, Accessed on 01/09/2026.
[20] MCP08 - Lack of Audit and Telemetry, https://microsoft.github.io/mcp-azure-security-guide/mcp/mcp08-telemetry/ , Accessed on 01/09/2026.
[21] Comment Control: GitHub AI Agents as Credential Exfiltrators, https://labs.cloudsecurityalliance.org/research/csa-research-note-comment-control-github-prompt-injection-20/, Accessed on 01/09/2026.
[22] LLM06:2025 - Excessive Agency, https://genai.owasp.org/llmrisk/llm062025-excessive-agency/, Accessed on 01/09/2026.
[23] "Do Not Mention This to the User": Detecting and Understanding Malicious Agent Skills in the Wild, https://arxiv.org/abs/2602.06547, Accessed on 01/09/2026.

outdated or tampered with[17], they can influence how an assistant or agent generates/modifies code, selects dependencies, invokes tools or explains security claims.

3.3 Governance and assurance risks

Governance and assurance risks can arise where an organisation does not have enough control, accountability or evidence over AI-assisted development activities. Because of the rapid evolvements in the field, many organisations are still determining how AI assistants and agents should be used in practice. During this learning phase, AI-assisted development could very well be relying on ad hoc prompts and processes, individual developer habits, tool-specific settings or informal review practices. As these activities become more integrated into development workflows, organisations should consider how security expectations and requirements are defined, maintained and applied consistently.

Table 4: Governance and assurance risks

Governance / assurance risk	Example in AI-assisted software development
Unclear accountability	It is unclear who is responsible for AI-generated outputs, especially where an agent performs multi-step tasks across several files or tools.
Uncontrolled autonomy	Assistants or agents are allowed to modify code, install dependencies, run commands, change configuration or prepare pull requests without clear limits or approval points.
Inconsistent security instructions	Security expectations are applied inconsistently because prompts, project instructions, rules or skills are missing, outdated, conflicting or not treated as authoritative.
Weak traceability	It is difficult to link AI-assisted changes to requirements, design decisions, security requirements, review outcomes, tests or evidence.
Insufficient review capacity	Developers or reviewers cannot effectively assess the volume, complexity or scope of AI-generated changes.
Unsupported security claims	AI-generated statements such as “this is secure”, “the issue is fixed” or “all checks passed” are accepted without supporting evidence.

Taken together, the risks and threats discussed in this section show a common pattern. As AI-assisted development is used across the software development lifecycle, security expectations should be made explicit earlier and applied consistently across the lifecycle. Shifting security left in an AI-assisted context therefore requires more than asking developers to review generated code at the end or identifying and patching vulnerabilities after code has already been shipped. Rather, it requires security requirements, constraints, verification steps and evidence expectations to be built into the way assistants and agents are instructed, used and reviewed. The next section explores good practices for secure AI-assisted software development.

4. Good practices for secure AI-assisted software development

The risks, failure modes and threats described in Section 3 call for a combination of measures. This section does not aim to provide controls for all possible risks associated with AI-assisted software development. Rather, it proposes a high-level approach for applying secure by design expectations to AI-assisted development activities, specifically by building on established secure development practices used for similar or equivalent developer led activities.

For example, functional but insecure outputs and incomplete context should be addressed by defining clear security requirements and applying established secure development practices. AI-generated code should be subject to the same security requirements and assurance activities as equivalent developer-produced code, including (where relevant) code review, testing, SAST, DAST, dependency scanning and other relevant security checks.

Similarly, weak traceability, repudiation and unsupported security claims should be addressed through appropriate evidence and review, in line with established secure development, change-management and assurance processes.

Rather than restating existing controls or requirements, this section focuses on how applicable security expectations can be identified and translated into AI-assisted development workflows.

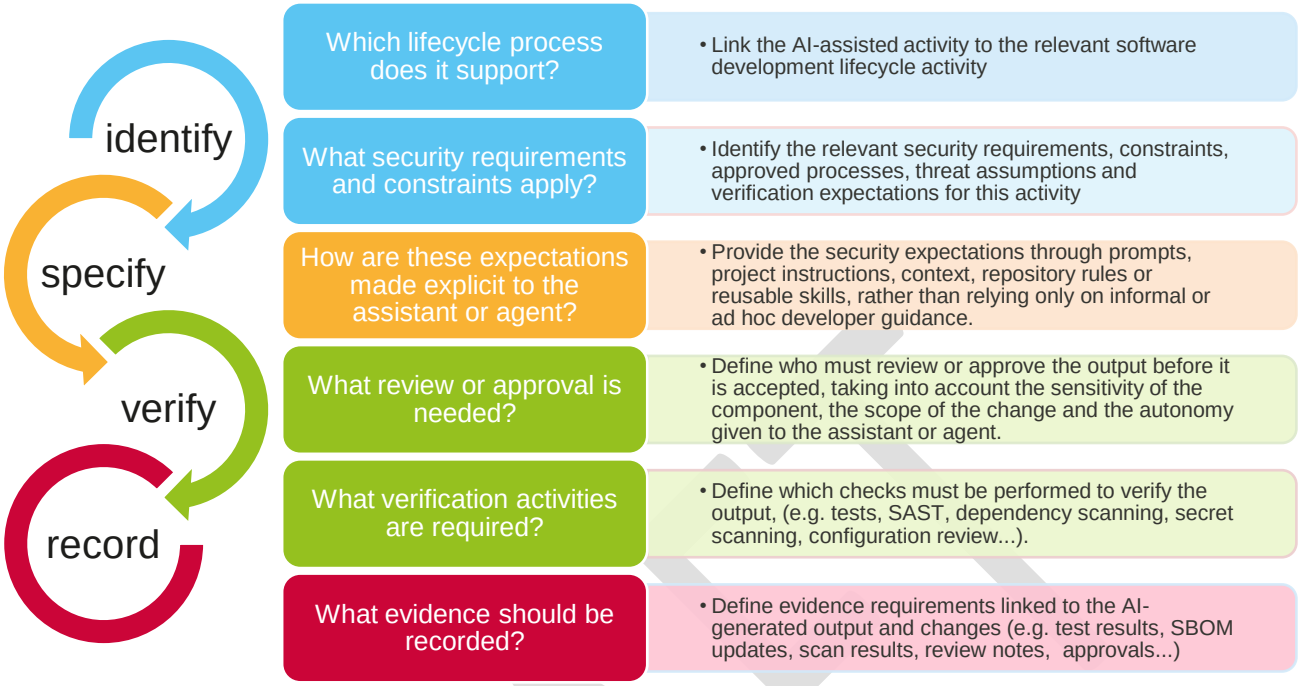
This technical advisory sets out a high-level approach based on four actions for AI-assisted development activities. First, **identify** the relevant lifecycle process and applicable security expectations for this activity. Second, **specify** those expectations for the assistant or agent. Third, **verify** outputs through review, approval and security checks. Finally, **record** the supporting evidence.

Each action is supported by one or more guiding questions that can be applied to a specific AI-assisted development activity:

- Which lifecycle process does it support?
- What security requirements and constraints apply?
- How are these expectations made explicit to the assistant or agent?
- What review or approval is needed?
- What verification activities are required?
- What evidence should be recorded?

Together, these questions provide a practical way to apply the four actions across different AI-assisted development activities, regardless of the specific tool or level of autonomy involved. Figure 2 shows how the six questions map to the four actions.

Figure 2: Applying secure by design expectations to AI-assisted development activities



To illustrate how this can be applied in practice, the rest of this section uses the AI-assisted dependency selection activity (from Section 2.2.1) as a running example:

AI-assisted activity	Description	Lifecycle mapping
Dependency selection and integration	Suggesting, adding, updating or replacing third-party packages or libraries.	Implementation / Maintenance

4.1 Identify the applicable lifecycle process and security expectations

The first two questions address **which lifecycle process the AI-assisted activity supports,** and **which security requirements and constraints apply to it.**

AI-assisted software development should be integrated into the organisation's existing secure development lifecycle, rather than looking at it as a separate or informal activity. AI-assisted activities such as the ones described in Section 2.2 can be mapped to the lifecycle processes they support.

This mapping helps to determine which security requirements, constraints, current practices, review steps, verification activities and evidence requirements apply to the activity.

Example: dependency selection and integration

For dependency selection and integration, the relevant lifecycle processes are implementation and maintenance. Applicable security expectations, regardless of whether AI assistance is used may include:

- using trusted package sources;

- assessing whether a new dependency is needed;
- checking package reputation and maintenance status;
- reviewing transitive dependencies;
- assessing known vulnerabilities;
- updating lockfiles; and
- recording the verification performed;

Existing guidance, such as ENISA's technical advisory on the secure use of package managers[24], can be used as a source for defining such expectations.

4.2 Specify the expectations for the assistant or agent

Once the applicable security expectations have been identified, the next question addresses **how those expectations are made explicit to the assistant or agent.**

Making these expectations explicit can help to ensure that assistants or agents are guided before suggesting or applying for changes. Rather than relying on ad hoc prompts or individual developer practices the expectations should be provided consistently, for example through reusable skills, project instructions and relevant context. Annex A0 provides a practical example on how security requirements can be encoded in reusable skills.

Example: dependency selection and integration

Taking the dependency selection example and the applicable security expectation of ***'using trusted package sources'*** identified in Section 4.1, an organisation could create a dedicated package-selection skill that instructs the assistant or agent to use only approved registries or package proxies, avoid unofficial sources, and flag any exception for human approval.

The skill could then be made available across relevant projects, with its use required through organisational processes or supported tooling for tasks involving the suggestion, addition or update of dependencies. This helps apply the same security expectation consistently across projects, tools and development teams.

4.3 Verify outputs through review, approval and security checks

Organisations should then determine **what review or approval is needed** and **what verification activities are required** before AI-assisted outputs are accepted**.** These requirements should be proportionate to the risk of the activity, considering aspects like the sensitivity of the affected component, the scope and complexity of the change, the tools and information available to the assistant or agent, and the level of autonomy granted to it.

Example: dependency selection and integration

In the dependency selection and integration example, review or approval should be required before accepting new dependencies, major dependency updates, lockfile changes, or changes affecting build, deployment or CI/CD configuration. This is particularly important where the assistant or agent can add dependencies, modify lockfiles, run install scripts, change build configuration, or prepare pull requests with limited developer involvement.

[24] ENISA Technical Advisory – Package Managers, https://www.enisa.europa.eu/sites/default/files/2026-03/ENISA%20Technical%20Advisory%20-%20Package_Managers_Final.pdf, Accessed on 01/09/2026.

Verification activities may include checking the package source, version, maintainer status, known vulnerabilities, transitive dependencies, build impact and test results. The purpose of these checks is to support the human decision on whether the proposed dependency change should be accepted.

4.4 Record the supporting evidence

Finally, organisations should determine **what evidence should be recorded** to demonstrate how the AI-assisted activity was performed, verified and accepted.

Where appropriate, evidence should be generated and retained to show what was changed, why it was changed, which checks were performed, what the results were, and who reviewed or approved of the changes.

Logs and review artefacts can also help to reconstruct how AI-assisted changes were produced, reviewed and accepted. Where appropriate, organisations can move towards more structured evidence generation, e.g. through SBOM updates, signed provenance records, and machine-processable attestations that can be integrated into CI/CD pipelines.

An assistant or agent can be explicitly instructed to help collect, summarise or format evidence. However, it should not replace the underlying checks or the human decision to accept the change.

Example: dependency selection and integration

For dependency selection and integration, evidence could include information like:

- the package name, version and source;
- the reason for adding or updating the package;
- vulnerability scan results;
- dependency manifest and lockfile changes;
- test results;
- an updated SBOM; and
- reviewer approval;

In general, AI-generated statements such as "the package is safe" or "all checks passed" should not be accepted unless they are linked to supporting evidence.

Annex A: Practical example: encoding secure by design expectations through reusable skills

This annex provides a practical example of how the actions set out in Section 4 can be applied in practice. It uses reusable skills as a means of specifying the relevant expectations for the assistant or agent. Established package-management guidance is used to identify applicable security requirements and translate them into skill content, including instructions, constraints, checks, approval points and evidence expectations.

A.1 What is a security skill?

A skill refers to a reusable set of instructions and supporting resources that guides an AI assistant or agent when performing a specific task.

In the context of secure AI-assisted software development, a security skill can be used to make secure by design expectations explicit before the assistant or agent generates or modifies outputs. Rather than relying only on generic ad hoc prompts from individual developers (e.g. 'make this output secure'), skills can provide a structured and consistent way to encode security requirements, approved practices, verification steps and evidence expectations.

It is important to note that a skill is only one type of reusable instruction mechanism and is not supported in the same form by every AI-assisted development system. Other systems may provide similar mechanisms, such as repository or project instruction files, rules, prompt templates and reusable workflows. Examples include AGENTS.md[25], CLAUDE.md[26], GEMINI.md[27] and GitHub Copilot repository instructions[28]. These mechanisms can also differ in their structure, scope and method of activation. For example, project instruction files typically provide persistent or scoped context, while skills are typically used to package task-specific instructions and supporting resources that can be loaded or invoked when needed.

Depending on the AI assistant or agent features and available configuration options, skills can usually be enabled or disabled to meet the needs of the use case. Moreover, skills can usually be referenced by the operator providing a clear request to use the mentioned skill during the interaction with the underlying model or just inform the underlying model that a skill is available allowing it to invoke it on the fly when its scope is found appropriate to the requested task.

[25] AGENTS.md, https://agents.md/, Accessed on 01/09/2026.
[26] Claude Code - How Claude remembers your project, https://code.claude.com/docs/en/memory, Accessed on 01/09/2026.
[27] Gemini CLI - Provide context with GEMINI.md files, https://github.com/google-gemini/gemini-cli/blob/main/docs/cli/gemini-md.md, Accessed on 01/09/2026.
[28] GitHub Copilot - Add custom instructions for Copilot, https://docs.github.com/en/copilot/how-tos/copilot-on-github/customize-copilot/add-custom-instructions, Accessed on 01/09/2026.

A.2 Skill structure

The exact structure of a skill depends on the assistant or agent framework. However, common skill formats generally use a folder that contains a required SKILL.md file together with optional supporting resources such as scripts, references or assets[29][30].

```
example-skill/
|- SKILL.md
|- references/     # Optional: additional documentation
|- scripts/        # Optional: executable code
|- assets/         # Optional: templates, images, data files
```

The SKILL.md file usually contains metadata, including the skill name and description, followed by Markdown instructions that describe when the skill should be used and what the assistant or agent should do. The following is an illustrative simple SKILL.md:

```
---
name: example-skill
description: Use this skill when the assistant or agent needs to perform a
specific repeatable task.
---

# Example Skill

## Purpose

Describe the task this skill supports and the outcome it should help produce.

## When to use this skill

Use this skill when the user request, repository context or development task
matches the scope described in the skill description.

## Instructions

Follow the steps below when applying this skill:

1. Identify the relevant input, files or context.
2. Apply the task-specific instructions.
3. Use any referenced scripts, templates or examples where appropriate and
permitted.
4. Produce the expected output in the requested format.

## Expected output

Produce the output required by the task, such as a code change, explanation,
summary, configuration change, pull request text or evidence record.
```

A.3 Encoding secure by design requirements in skills

Encoding secure by design requirements in skills can help to move security expectations earlier in the AI-assisted workflow. The assistant or agent is not only asked to produce a functional output, but also to consider relevant security requirements, constraints, approved practices and verification

[29] Claude - Creating custom skills, https://claude.com/docs/skills/how-to, Accessed on 01/09/2026.
[30] Agent Skills Specification, https://agentskills.io/specification, Accessed on 01/09/2026.

expectations. This does not replace later review or testing, but it can reduce the likelihood that insecure or incomplete outputs are proposed in the first place.

For example, instead of asking an assistant to “find a package that can do X”, a security skill could instruct it to “identify whether an existing approved dependency can do X; if a new package is needed, use trusted sources, check maintenance status, assess known vulnerabilities and transitive dependencies, and explain the rationale for the selection.”

Therefore, the objective moves from simply finding a package that is functionally suitable to finding a suitable package that also considers security, maintenance, provenance, necessity and evidence before a change is proposed. The following subsections illustrate how this can be done by translating package manager guidance into concrete skill content.

A.4 Translating package manager guidance into a dependency-management skill

The ENISA technical advisory on secure use of package managers[31] presents practices for selecting, integrating and monitoring packages, as well as addressing vulnerabilities found in dependencies. These recommendations can be translated into skill content by defining aspects like when the skill should be used, which actions should be restricted or require approval and what evidence should be produced.

The table below provides an example of such a translation by focusing on the selection recommendations provided in the technical advisory, including the pre-selection recommendation to assess whether the dependency is needed in the first place. The wording is illustrative and should be adapted to the organisation’s approved package sources, tools and risk criteria.

Table 5: recommendation-to-instruction

Technical advisory recommendation	Possible Skill section	Example skill content
Assess whether the dependency is needed	## Instructions	Check whether the requirement can be met using platform functionality, an existing project dependency or an approved internal library. Do not introduce a new dependency unless there is a clear need.
Trusted source	## Instructions / ## Constraints	Use only official and verifiable package registries, or the organisation’s approved package proxy. Prefer packages published through secure workflows that provide provenance metadata. Do not suggest packages from unofficial mirrors, unknown forks or unverified sources unless explicitly approved.
Existing known vulnerabilities	## Instructions / ## Checks	Check the selected package and version for known vulnerabilities using available vulnerability databases or scanning tools before recommending or using it. Flag unresolved high or critical findings for human review.
Package signing and integrity verification	## Checks	Where available, verify package signing, integrity or provenance metadata. Do not bypass package-manager integrity checks or recommend installation methods that weaken integrity verification.
Maintainer reputation	## Instructions / ## Checks	Review whether the package is maintained by a reputable or verified organisation, publisher or maintainer. Flag unclear ownership, newly created maintainers or suspicious maintainer changes for review.

[31] ENISA Technical Advisory – Package Managers, https://www.enisa.europa.eu/sites/default/files/2026-03/ENISA%20Technical%20Advisory%20-%20Package_Managers_Final.pdf, Accessed on 01/09/2026.

Popularity and maintenance	## Instructions / ## Checks	Consider adoption and maintenance signals, such as recent releases, changelog activity, downloads, commits and open issues. Do not rely on popularity metrics alone, as they may be misleading or inflated.
Usage of secure practices	## Instructions / ## Checks / ## Constraints	Review whether the package uses secure build and installation practices. Flag unnecessary or risky installation behaviour, such as unusual install scripts, post-install hooks or excessive dependency chains.

A possible repository structure could be:

```
secure-dependency-selection/
|-- SKILL.md
|-- references/
|   |-- package-selection-mapping.md
|   |-- approved-package-sources.md
|-- templates/
|   |-- dependency-selection-evidence.md
```

The following excerpt shows an illustrative example of what such a dependency-selection skill could look like. It is intended to show how security expectations can be expressed as assistant or agent instructions, not to provide a complete package-management security policy:

```
# Secure Dependency Selection
## Purpose
Guide dependency decisions so that third-party packages are selected only when
necessary and assessed for source trust, known vulnerabilities, integrity, provenance,
maintainer reputation, maintenance status and secure practices.

## When to use this skill
Use this skill when the task may involve:

- suggesting a new third-party package or library;
- adding, updating or replacing a dependency;
- reviewing dependency-related changes;
- modifying dependency manifests or lockfiles.

## Instructions

Before suggesting, adding, updating or replacing a dependency:

1. Check whether the requirement can be met using platform functionality, an existing
project dependency or an approved internal library.
<!-- Additional instructions from Table 5. -->

## Checks

Where tooling and context are available, perform or request the following checks before
recommending or using the dependency:

- Check the selected package and version for known vulnerabilities using available
vulnerability databases or dependency scanning tools.
<!-- Additional instructions from Table 5. -->

## Constraints

Do not suggest packages from unofficial mirrors, unknown forks, direct URLs or
unverified sources unless explicitly approved.
<!-- Additional instructions from Table 5. -->

## Supporting resources

Use the following supporting resources when available and relevant:
```

```
- [Package-selection mapping](references/package-selection-mapping.md)
- [Approved package sources](references/approved-package-sources.md)
- [Dependency-selection evidence template](templates/dependency-selection-evidence.md)

If organisation-specific approved package sources are defined in `references/approved-
package-sources.md`, treat them as authoritative.

Where evidence is requested, use `templates/dependency-selection-evidence.md`.

## Expected output

Provide:

- package name and version;
- package source or registry;
- reason for adding, updating, replacing or rejecting the package;
- whether platform functionality, an existing dependency or an approved internal
library could meet the need;
- checks performed and results;
- residual concerns or required human approvals.

Do not state that a package is "safe" or that "all checks passed" unless the statement
is linked to supporting evidence.
```

A more detailed implementation of a dependency-management skill, focusing on more recommendations from the technical advisory, is available in the ENISA skill repository[32].

A.5 Limitations of skills

Skills can help make security expectations more explicit, consistent and reusable, but it should be noted that they do not guarantee secure outputs. Their effectiveness can depend on factors like the quality of the requirements, instructions, tools and verification steps available to the assistant or agent. If a skill is incomplete too generic or applied in the wrong context, the assistant or agent may still produce insecure, unsuitable or unsupported outputs.

Skills should therefore be considered as a possible control within a broader secure development process. They do not replace secure engineering judgement, human review, vulnerability management or accountability for accepted changes. Skills also need to be maintained over time. They should be reviewed and updated as security requirements, development tools and threats evolve. It is also important to keep in mind that skills and their supporting resources should themselves be treated as security-relevant inputs and obtained, reviewed and maintained accordingly.

[32] ENISA's Skills for AI Agents, https://github.com/enisaeu/agentic-skills, Accessed on 16/09/2026.

ABOUT ENISA

The European Union Agency for Cybersecurity, ENISA, is the Union's agency dedicated to achieving a high common level of cybersecurity across Europe. Established in 2004 and strengthened by the EU Cybersecurity Act, the European Union Agency for Cybersecurity contributes to EU cyber policy, enhances the trustworthiness of ICT products, services and processes with cybersecurity certification schemes, cooperates with Member States and EU bodies, and helps Europe prepare for the cyber challenges of tomorrow. Through knowledge sharing, capacity building and awareness raising, the Agency works together with its key stakeholders to strengthen trust in the connected economy, to boost resilience of the Union's infrastructure, and, ultimately, to keep Europe's society and citizens digitally secure. More information about ENISA and its work can be found here: www.enisa.europa.eu.

ENISA
European Union Agency for Cybersecurity

Athens Office
Agamemnonos 14
Chalandri 15231, Attiki, Greece

Brussels Office
Rue de la Loi 107
1049 Brussels, Belgium

enisa.europa.eu

Publications Office
of the European Union