Sonar

The State of Code

Volume 4: Languages

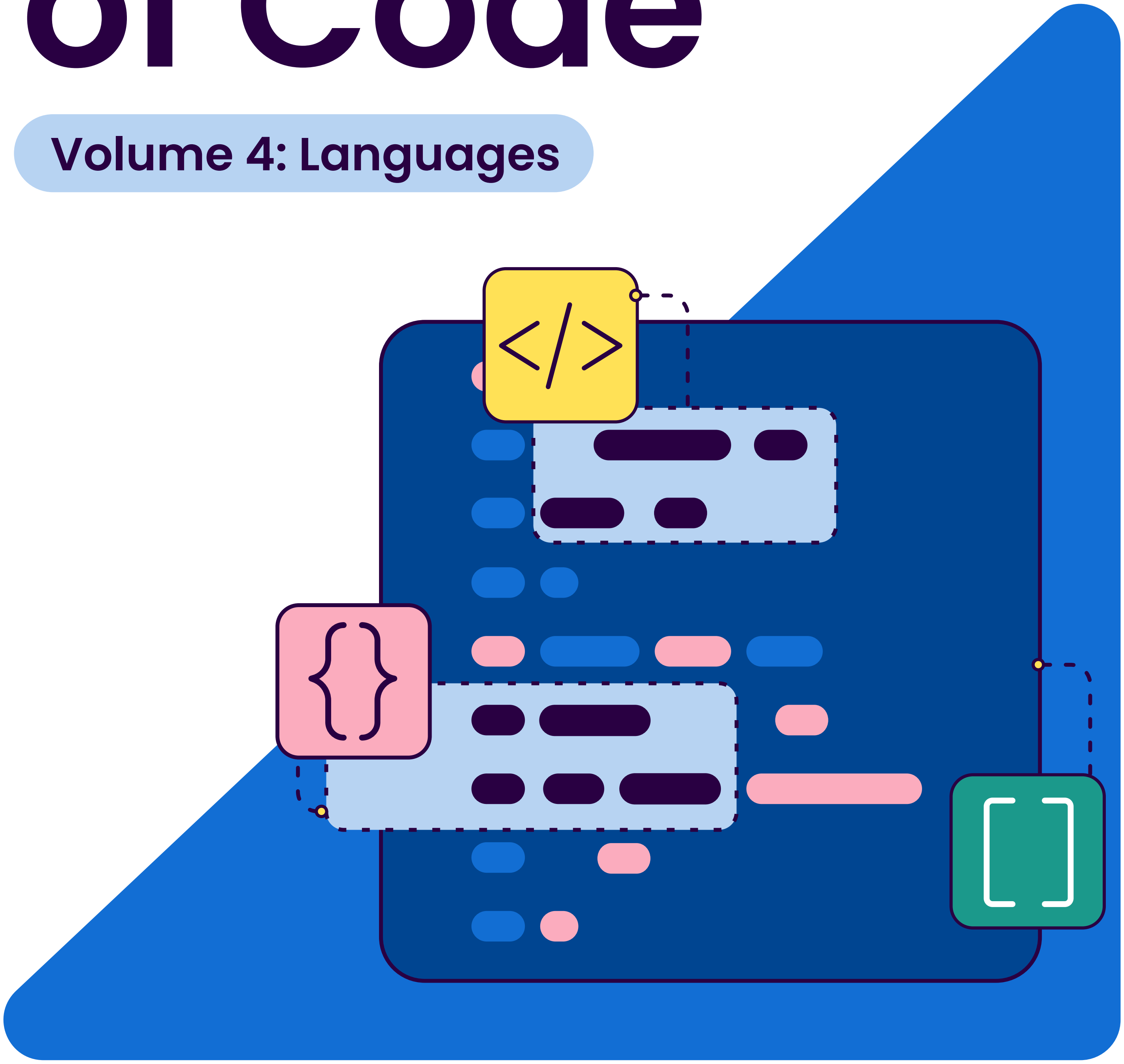


Table of Contents

Introduction 3

Overview and summary of key findings 4

Our methodology 5

Report findings 6

- Java 7
- JavaScript 10
- TypeScript 13
- Python 16
- C# 19
- C++ 22
- PHP 25

Conclusion 28

About Sonar 28

Introduction

Sonar's integrated solutions for code security and code quality, called SonarQube, examine 300 billion lines of code every single day in order to help development teams ship high-quality, secure code. Given our unique visibility into the code developers are creating today, we analyzed a subset of that code in order to extract the top issues that surfaced through our analysis. This is the fourth in a series of reports highlighting our findings.

In this report, we highlight the most common issues affecting code quality across the seven most popular languages used by software developers, according to our data. This context is essential for software developers, their leaders, and AppSec stakeholders who need to make (or justify) software engineering investment decisions around training, tooling, or technical debt and would benefit from knowing what issues may be lurking or unknown in their critical software. And as AI coding assistants generate more code, the quality of existing application source code becomes more important, as it is the main data used to train these AI tools.

For background, Sonar measures the impact of code issues in every project or codebase across three software qualities: reliability, security, and maintainability. These three areas are deeply interconnected in high-quality code: poorly maintained code typically develops reliability issues and security vulnerabilities over time. Taken together, reliability, security, and maintainability determine not just the initial success of software but its long-term value, adaptability, and total cost of ownership throughout its lifecycle. This is why every rule violation identified by Sonar is automatically assigned an impact quality for one of these three areas.

In this report, we'll surface the most common code issues we've seen (across reliability, security, and maintainability) in **Java**, **JavaScript**, **TypeScript**, **Python**, **C#**, **C++**, and **PHP**. Our prior reports, "The State of Code: Reliability," "The State of Code: Security," and "The State of Code: Maintainability" explore the most common issues across languages in each of the three areas of code quality, and are also worth a read.

Overview and summary of key findings

Software plays a vital role in modern business, increasingly serving as a key differentiator and driver of customer experience. Consequently, the quality of internally developed software will have a growing impact on both customer satisfaction and competitive standing.

Poor software quality, therefore, represents a significant threat to every business. Recent projections estimate that the annual cost of poor software quality in the US has risen to over $2.41 trillion (Consortium for Information & Software Quality, 2022).

The staggering growth of new code that is generated or touched by AI tools dramatically increases the impact of this trend. AI coding tools are excellent mimics: they create based on what they learn from existing human code. It follows that code quality problems that commonly exist today will continue to pop up in AI generated code, and these issues must be addressed.

This report provides analysis of the most prevalent reliability, security, and maintainability issues across popular programming languages that software developers use with SonarQube.

Reliability issues or **bugs** are issues that could affect the software's capability to maintain its level of performance under promised conditions.

Security issues can be flagged as either **vulnerabilities** or security **hotspots**. A vulnerability indicates that the code could be open to attacks and requires immediate action, while security hotspots are security-sensitive pieces of code that need manual review to determine if they pose a threat.

Maintainability issues, often referred to as **code smells**, are problems in the code that affect its maintainability but not necessarily its functionality. They indicate weaknesses in the code design that can make the code hard to read, understand, and maintain, and they can increase the risk of bugs and errors over time.

Here are the most significant code quality issues* found in each language:

- Java: Delivering code in production with debug features activated (a security hotspot)
- JS: Code statements that don't have side effects, aka dead code (a bug)
- TS: Using hard-coded IP addresses (a security hotspot)
- Python: Using clear text protocols (a security hotspot)
- C#: Dereferencing null pointers (a bug)
- C++: Using unsafe string handlers like "strlen" or "wcslen" (a security hotspot)
- php: Omitted visibility declarations, defaulting methods to public (a bug)

* This list excludes code smells, which were naturally about 16 times more numerous than bugs, vulnerabilities, and hotspots combined.

Our methodology

Sonar operates the largest SaaS solution for integrated code security and code quality analysis, so we have a unique view into how code is being written across many languages, industries, organization sizes, and geographies. We looked at SonarQube analysis data for the last six months of 2024, across seven of the most common languages developers use. Unlike other reports that rely on surveys, this analysis is backed by concrete data that shows the real issues developers are encountering as they code. Moreover, each of these data points links back to an issue that was caught by SonarQube and surfaced to developers.

About our dataset

The dataset for this report comes from SonarQube Cloud, which is a SaaS code analysis solution designed to detect coding issues in over 35 languages, frameworks, and Infrastructure-as-Code (IaC) platforms. It integrates directly with continuous integration (CI/CD) pipelines and DevOps platforms (like GitHub, GitLab, Azure DevOps, and Bitbucket), and checks code against an extensive set of rules covering maintainability, reliability, and security issues on each merge/pull request or branch commit. Our analysis evaluates code health against thousands of criteria that have been created by developers, for developers.

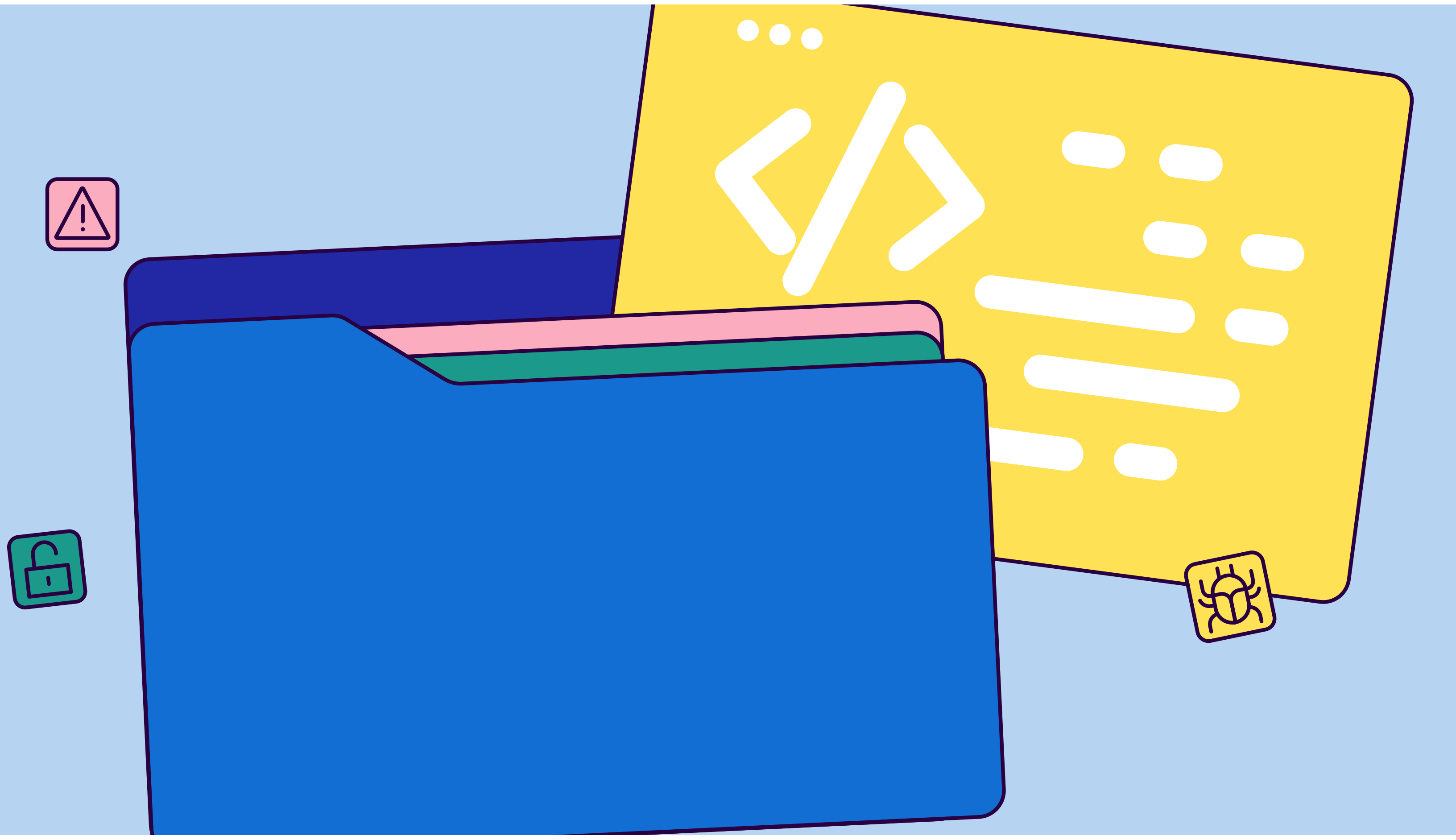
For this study, we've included every pull request and branch analysis in 2024 between July 1st and December 31st, where the code was written in **Java, JavaScript, TypeScript, Python, C#, C++, or PHP** (the top software development languages used with SonarQube).

This scope yields a vast dataset encompassing:

- More than **7.9 billion lines of code**.
- Work from over **970,000 developers** across over **40,000 organizations** globally.
- Roughly **445 million code issues** across **5,300 unique quality and security rules**.

Report findings

The following sections explore prevalent coding issues across the most popular programming languages software developers use with SonarQube (Java, JavaScript, TypeScript, Python, C#, C++, and PHP). Each language has its own dedicated subsection featuring a list of the top reliability and security issues Sonar discovered, followed by a list of the top discovered maintainability issues. We list maintainability issues separately because they are about 16 times more numerous than bugs, vulnerabilities, and hotspots combined.



Java

Sonar’s dataset for Java contained analysis of roughly 1.5 billion lines of code. We found about 106 million issues across those scans (about 69,000 issues per million lines of Java code).

The most common reliability and security issues

Delivering code in production with debug features activated is security-sensitive

Type: Security hotspot **Volume:** □□□□□□□□□□□

Debugging features in development tools and frameworks should never be enabled in production. They can leak sensitive system information and pose a security risk. Developers should ensure these features are never enabled in production environments or applications distributed to end users.

Null pointers should not be dereferenced

Type: Bug **Volume:** □□□□□□

Accessing a null value can cause an exception, or even program termination that could potentially expose sensitive information. To prevent this, developers should ensure that the variable has a value, or check that it is not null before accessing it.

Logging should not be vulnerable to injection attacks

Type: Vulnerability **Volume:** □□□□

Log injection happens when an application doesn't sanitize untrusted data used for logging. Attackers can forge log content to hide malicious activities, compromising the integrity of the log. Developers should ensure data used for logging is content-restricted and typed by validating or sanitizing the content.

Exceptions should not be thrown from servlet methods

Type: Vulnerability **Volume:** □□□

Servlets, Java web development components that process HTTP requests and generate responses, use exceptions to manage unexpected errors. Unfortunately, servlet method signatures don't require exception handling, so surrounding method calls with try/catch blocks is crucial.

Optional value should only be accessed after calling isPresent()

Type: Bug **Volume:** □□□

Optional objects can hold either a value or not, and trying to directly access its value will throw a NoSuchElementException if there is no value present. To avoid the exception, developers should call the isPresent() or isEmpty() method before any call to get().

Why you should turn off debug features in production

Enabling debugging features in production environments exposes sensitive system information like error messages, stack traces, and internal architecture details that attackers can exploit to identify vulnerabilities. These debugging features may also provide interactive capabilities that could grant unauthorized access to application internals or database queries. This would significantly expand the attack surface and compromise security principles that, in production settings, should prioritize protection over transparency. Since it is a security hotspot, developers should review debug features to ensure that only those needed in production are turned on.

The most common maintainability issues

Local variable and method parameter names should comply with a naming convention

Volume: □□□□□□□□□

Adhering to naming conventions for code elements improves readability and maintainability. Inconsistent naming can lead to errors and collaboration difficulties. To fix this, developers should understand the project's naming convention and update the names accordingly.

"@Deprecated" code should not be used

Volume: □□□□□

Deprecated code should not be used because it will eventually be removed. Developers should check the relevant documentation for more information and recommended alternatives.

Unnecessary imports should be removed

Volume: □□□□

Unnecessary imports don't affect runtime behavior, but they do affect readability, maintainability, build times, and auto-completion relevance. They can be removed manually or automatically using code editors.

Sections of code should not be commented out

Volume: □□□

Commented-out code creates unnecessary noise that can be distracting and complicate code maintenance. It's never executed, so it can quickly become outdated and invalid. Developers should remove code that is commented out.

String literals should not be duplicated

Volume: □□□

Duplicated string literals make the process of refactoring complex and error-prone, as any change would need to be propagated on all occurrences. Developers should use constants to replace the duplicated string literals, since they can be updated once and referenced from many places.

Why you should follow naming conventions

Inconsistent naming conventions in Java code significantly reduce readability and maintainability. When developers use different naming patterns across variables, methods, and classes, code becomes difficult to understand, increasing cognitive load and leading to misinterpretations that introduce bugs. This inconsistency hampers collaboration, makes onboarding new team members harder, and diminishes the effectiveness of automated tools and IDEs. What appears to be a minor stylistic issue ultimately impacts code quality, team productivity, and long-term project maintenance.

JavaScript

Sonar's dataset for JavaScript contained analysis of roughly 1.3B lines of code. We found about 116 million issues across those scans (about 93,000 issues per million lines of JavaScript code).

The most common reliability and security issues

Non-empty statements should change control flow or have at least one side-effect

Type: Bug **Volume:** □□□□□□□□□□□□□

Statements that have no side effects or do not change control flow can indicate a programming error or be redundant. They might not behave as intended due to errors or be residual from refactoring. Developers should verify that these statements do not contain bugs or redundancies.

Using slow regular expressions is security-sensitive

Type: Security Hotspot **Volume:** □□

Regular expressions can cause performance issues due to backtracking. To avoid this, ensure your regex doesn't have problematic repetitions, consecutive repetitions that can match the same content, or unbounded repetitions in partial matches.

Mouse events should have corresponding keyboard events

Type: Bug **Volume:** □

Developers should support both mouse and keyboard navigation to ensure compatibility with assistive technology. For elements using onClick, add onKeyUp, onKeyDown, or onKeyPress. For elements using onmouseover/onmouseout, add onfocus/onblur.

The return value of void functions should not be used

Type: Bug **Volume:** □

JavaScript functions without a return statement, or with a return statement but no value, implicitly return "undefined". Developers should avoid using their return values because they are meaningless and may cause errors.

Using clear-text protocols is security-sensitive

Type: Security Hotspot **Volume:** □

Clear-text protocols like FTP, telnet, and HTTP lack encryption and authentication, exposing applications to data leaks, malicious redirects, malware, and code execution. All transport channels should be secured, even on local networks, to prevent compromise.

Why you should eliminate dead code

When writing code, it is important to ensure that each statement serves a purpose and contributes to the overall functionality of the program. Statements without side effects or control flow impact are problematic because they typically represent programming errors or redundancy rather than intentional design. Potentially, they can mask actual bugs when developers assume the code works correctly, when in reality critical operations may be incorrectly implemented. At minimum, these "do-nothing" statements often indicate incomplete implementations, logical mistakes, or remnants from refactoring that weren't properly removed. This increases performance overhead, maintenance time, and costs as teams investigate code that accomplishes nothing.

The most common maintainability issues

Variables should be declared with "let" or "const"

Volume: □□□□□□□□□

Variables declared with \`var\` have function scope and can be accessed within the entire function. \`let\` and \`const\` have block scope and are limited to the block of code in which they are defined. \`const\` variables are also immutable. \`let\` and \`const\` are preferred due to their more precise variable types.

React components should validate prop types

Volume: □□□□

In JavaScript, props can be passed as plain objects, which can lead to errors. By defining types for component props, developers can enforce type safety and provide clear documentation, which helps catch errors and improves code maintainability.

Comma operator should not be used

Volume: □□□

The use of the comma operator to combine two expressions is generally detrimental to the readability and reliability of code, and the same effect can be achieved by other means. Developers should write each expression on its own line to improve readability and prevent misunderstandings.

"===" and "!==" should be used instead of "==" and "!="

Volume: □□

JavaScript has strict and non-strict comparison operators. Strict operators (===, !==) compare both value and type, while non-strict operators (==, !=) compare only value. Developers should use strict operators to avoid unexpected results due to type coercion.

Functions should not be nested too deeply

Volume: □

Nested functions are common in JavaScript, but deep nesting can impact readability and maintainability due to increased complexity. Developers should refactor deeply nested functions into smaller, more manageable ones, or consider alternative design patterns.

Why you should make sure variables are scoped correctly

Variables declared using \`var\` exist throughout their entire function, not just within the specific block of code where they were defined. This means that variables can be accessed before they're properly set up (giving mysterious undefined values) and can leak outside of loops or if-statements. \`const\` prevents variables from being changed after creation, making it harder to accidentally reassign values. These issues might seem small, but they lead to bugs that are hard to find, especially in larger programs. That's why modern JavaScript introduced let and const—they create clearer boundaries around where variables exist and whether they can be changed.

TypeScript

Sonar's dataset for TypeScript contained analysis of roughly 1.7 billion lines of code. We found about 62 million issues across those scans (about 37,000 issues per million lines of TypeScript code).

The most common reliability and security issues

Using hardcoded IP addresses is security sensitive

Type: Security Hotspot **Volume:** □□□□□□□

Hardcoding IP addresses in source code can cause issues in product development, delivery, deployment, and security. Developers should use environment variables or configuration files for IP addresses instead, or use a domain name if confidentiality is not required.

Mouse events should have corresponding keyboard events

Type: Bug **Volume:** □□□□□□

Developers should support both mouse and keyboard navigation to ensure compatibility with assistive technology. For elements using onClick, add onKeyUp, onKeyDown, or onKeyPress. For elements using onmouseover/onmouseout, add onfocus/onblur.

Using pseudorandom number generators (PRNGs) is security sensitive

Type: Security Hotspot **Volume:** □□□□□

PRNGs are insufficiently random for security applications. Developers should only use random number generators recommended by trusted organizations (e.g. OWASP), use the generated values only once, and refrain from exposing them.

Return values from functions without side effects should not be ignored

Type: Bug **Volume:** □□□□

Functions return values that can be used in the calling code, and disregarding them may indicate errors or poor coding practices. Developers should ensure that the return values are either assigned to variables or used directly in expressions.

Promises should not be misused

Type: Bug **Volume:** □□□□

Promises are used for asynchronous operations. If a promise is not awaited or resolved, it returns a promise object instead of the expected value, which can lead to bugs. To ensure that the expected value is returned, developers should always await promises or resolve them.

Why you shouldn’t hardcode IP addresses

Hardcoding IP addresses in source code creates inflexible applications that break whenever network environments change. This approach forces code modifications for simple infrastructure changes that should only require configuration updates, causing deployment delays and increasing error risk. From a security perspective, embedded IPs can expose internal network topology to potential attackers while bypassing proper access controls. By violating the principle of separating configuration from code, hardcoded IPs make applications less portable, less secure, and significantly harder to maintain across different environments.

The most common maintainability issues

Fields that are only assigned in the constructor should be "readonly"

Volume: □□□□□□□□□□

Fields that are only assigned a value within a class constructor, but are not marked as readonly, can lead to confusion about their intended use. To prevent this ambiguity and avoid unintentional modifications by future maintainers, developers should explicitly mark these fields as readonly.

Unnecessary imports should be removed

Volume: □□□□

Unnecessary imports should be removed. They don't affect runtime behavior, but they do affect readability, maintainability, build times, and auto-completion relevance. They can be removed manually or automatically using code editors.

Deprecated APIs should not be used

Volume: □□□

Deprecated code should not be used because it will eventually be removed. Developers should check the relevant documentation for more information and recommended alternatives.

Optional chaining should be preferred

Volume: □□

The optional chaining operator '?.' provides a concise, safe way to access nested properties or methods of an object without having to check if each intermediate property exists. Use it to replace expressions that check for null/undefined values before accessing properties, elements, or calling functions.

Sections of code should not be commented out

Volume: □□

Commented-out code creates unnecessary noise that can be distracting and complicate code maintenance. It's never executed, so it can quickly become outdated and invalid. Developers should remove code that is commented out.

Why you should make fields written only in constructors read only

Fields in classes that are only assigned in constructors but lack the readonly modifier create dangerous ambiguity about mutability intentions. Without explicit readonly declarations, developers must examine the entire codebase to determine if fields should remain constant after initialization. This absence permits accidental reassignments by team members unfamiliar with the original design intent, potentially introducing subtle bugs. Missing readonly modifiers also eliminate valuable compiler safeguards that would catch unintended modifications, undermining TypeScript's type system benefits and generating technical debt that compounds as the codebase evolves and team composition changes.

Python

Sonar's dataset for Python contained analysis of almost a billion lines of code. We found about 18 million issues across those scans (about 20,000 issues per million lines of Python code).

The most common reliability and security issues

Using clear-text protocols is security-sensitive

Type: Security Hotspot **Volume:** □□□□□□□□

Clear-text protocols like FTP, telnet, and HTTP lack encryption and authentication, exposing applications to data leaks, malicious redirects, malware, and code execution. All transport channels should be secured, even on local networks, to prevent compromise.

Using hardcoded IP addresses is security-sensitive

Type: Security Hotspot **Volume:** □□□□□□□□

Hardcoding IP addresses in source code can cause issues in product development, delivery, deployment, and security. Developers should use environment variables or configuration files for IP addresses instead, or use a domain name if confidentiality is not required.

Floating point numbers should not be tested for equality

Type: Bug **Volume:** □□□□

Floating point numbers are imprecise due to their binary approximation and non-associative arithmetic, so equality checks can be unreliable. Developers should use a tolerance value (epsilon) for comparisons instead.

Using pseudorandom number generators (PRNGs) is security-sensitive

Type: Security Hotspot **Volume:** □□□

PRNGs are insufficiently random for security applications. Developers should only use random number generators recommended by trusted organizations (e.g. OWASP), use the generated values only once, and refrain from exposing them.

Non-empty statements should change control flow or have at least one side-effect

Type: Bug **Volume:** □□

Statements that have no side effects or do not change control flow can indicate a programming error or be redundant. They might not behave as intended due to errors or be residual from refactoring. Developers should verify that these statements do not contain bugs or redundancies.

Why you should avoid clear-text protocols

Clear-text protocols like FTP, telnet, and HTTP could expose systems to significant security risks. Without encryption, sensitive data including credentials and personal information can be easily intercepted during transmission. These protocols lack robust authentication, enabling attackers to perform man-in-the-middle attacks, redirect users to malicious sites, and inject malware. Even on local networks, these vulnerabilities create dangerous entry points for attackers, potentially compromising the entire system. The convenience of these protocols comes at the cost of leaving applications perpetually exposed to data theft, unauthorized access, and code execution vulnerabilities.

The most common maintainability issues

Local variable and method parameter names should comply with a naming convention

Volume: □□□□□□□□

Adhering to naming conventions for code elements improves readability and maintainability. Inconsistent naming can lead to errors and collaboration difficulties. To fix this, developers should understand the project's naming convention and update the names accordingly.

String literals should not be duplicated

Volume: □□□□□□□

Duplicated string literals make the process of refactoring complex and error-prone, as any change would need to be propagated on all occurrences. Developers should use constants to replace the duplicated string literals, since they can be updated once and referenced from many places.

Sections of code should not be commented out

Volume: □□□□

Commented-out code creates unnecessary noise that can be distracting and complicate code maintenance. It's never executed, so it can quickly become outdated and invalid. Developers should remove code that is commented out.

Cognitive Complexity of functions should not be too high

Volume: □□□□

Cognitive complexity measures how hard it is to understand a piece of code. Complexity increases with breaks in linear flow (loops, conditionals, etc.) and nesting levels. To reduce complexity, extract conditions into functions, break down large functions, and avoid deep nesting.

Unused local variables should be removed

Volume: □□□□

Unused local variables are declared but not used within the code block. They decrease readability, cause misunderstanding, and can lead to bugs, maintenance issues, and inefficient memory usage. Developers should remove them to maintain clarity and efficiency.

Why you should follow naming conventions

Inconsistent naming conventions create significant challenges for development teams. When variables, functions, and classes follow different naming patterns (for example, mixing camelCase, snake_case, and PascalCase), code becomes difficult to understand. It forces developers to constantly context switch between naming styles, increasing cognitive load and the likelihood of misinterpreting code behavior. Collaboration suffers as team members struggle to integrate their work with code that uses unpredictable naming patterns. Additionally, Python-specific tools and linters become less effective when naming conventions aren't followed. What initially seems like a stylistic preference ultimately undermines code quality, readability, and the long-term health of the project.

C#

Sonar's dataset for C# contained analysis of roughly 1.6 billion lines of code. We found about 103 million issues across those scans (about 65,000 issues per million lines of C# code).

The most common reliability and security issues

Null pointers should not be dereferenced

Type: Bug **Volume:** □□□□□□□□□□

Accessing a null value can cause an exception, or even program termination that could potentially expose sensitive information. To prevent this, developers should ensure that the variable has a value, or check that it is not null before accessing it.

Hard-coded credentials are security-sensitive

Type: Security Hotspot **Volume:** □□□□

Credential leaks often occur when a sensitive piece of authentication data is stored with the source code of an application. These credentials should be revoked immediately, and a secret vault should be used to generate and store a replacement.

Not specifying a timeout for regular expressions is security-sensitive

Type: Security Hotspot **Volume:** □□

Regular expressions without timeouts, especially those processing untrusted input with patterns vulnerable to catastrophic backtracking, can lead to Denial-of-Service attacks. Developers should always specify a matchTimeout and review the patterns to ensure they are not vulnerable.

Conditionally executed code should be reachable

Type: Bug **Volume:** □□

Code can become unreachable due to conditional expressions that are always true or false. Developers should review these conditional expressions and either update or remove them.

Floating point numbers should not be tested for equality

Type: Bug **Volume:** □□

Floating point numbers are imprecise due to their binary approximation and non-associative arithmetic, so equality checks can be unreliable. Developers should use a tolerance value (epsilon) for comparisons instead.

Why you should avoid dereferencing null pointers

Even though C# is memory safe, null reference errors are dangerous because they cause unpredictable application crashes and can potentially expose sensitive information through stack traces. These failures often occur only under specific production conditions that weren't caught during testing. What makes these bugs particularly troublesome is that they frequently appear far from their root cause—a null value might originate in one component but only trigger visible failures when used by another component much later in the execution flow. This separation between cause and effect makes null reference bugs notoriously difficult to diagnose, but simple to fix in advance with the right tooling in place to detect them early.

The most common maintainability issues

String literals should not be duplicated

Volume: ▢▢▢▢▢▢▢▢▢▢▢

Duplicated string literals make the process of refactoring complex and error-prone, as any change would need to be propagated on all occurrences. Developers should use constants to replace the duplicated string literals, since they can be updated once and referenced from many places.

Sections of code should not be commented out

Volume: ▢▢▢▢▢▢▢▢▢

Commented-out code creates unnecessary noise that can be distracting and complicate code maintenance. It’s never executed, so it can quickly become outdated and invalid. Developers should remove code that is commented out.

Methods and properties that don't access instance data should be static

Volume: ▢▢▢▢▢▢▢▢

Methods and properties that don't access instance data should be marked as static. This improves clarity, performance, memory usage, and testability.

Cognitive Complexity of functions should not be too high

Volume: ▢▢▢▢▢▢▢▢

Cognitive complexity measures how hard it is to understand a piece of code. Complexity increases with breaks in linear flow (loops, conditionals, etc.) and nesting levels. To reduce complexity, extract conditions into functions, break down large functions, and avoid deep nesting.

Boolean literals should not be redundant

Volume: ▢▢▢▢▢▢

Boolean literals (true, false) can be combined with logical operators to form logical expressions. However, comparing a boolean literal to a boolean variable or expression is redundant and makes code less readable. Developers should remove redundant boolean literals to improve code readability.

Why you should avoid duplicating string literals

Duplicated string literals can significantly impair code maintainability. When identical strings appear throughout a codebase, updates require finding and changing every occurrence, inevitably missing some instances and creating inconsistencies. These inconsistencies lead to bugs when string values represent critical elements like API endpoints or configuration keys. Using constants instead centralizes definitions, enabling single-point updates while providing meaningful names that document the string's purpose, making code both safer and more readable.

C++

Sonar's dataset for C++ contained analysis of roughly 350 million lines of code. We found about 20 million issues across those scans (about 58,000 issues per million lines of C++ code).

The most common reliability and security issues

Using "strlen" or "wcslen" is security-sensitive

Type: Security Hotspot **Volume:** □□□□□□□□□

The standard C library functions strlen and wcslen measure the length of a string, but they can lead to undefined behavior if the string is not null-terminated. This can create security vulnerabilities. To mitigate this, use safer functions like strlen_s and wcslen_s (C11 annex K), or std::string (C++).

"sprintf" should not be used

Type: Security Hotspot **Volume:** □□□□□□

sprintf can cause buffer overflows, which can lead to program crashes or malicious code execution. Developers should use snprintf or C++ alternatives like std::string, std::ostringstream, or std::format instead.

Using "strcpy" or "wcscpy" is security-sensitive

Type: Security Hotspot **Volume:** □□□□□□

C strings require careful management to avoid security vulnerabilities. strcpy and wcscpy can be unsafe if misused, and strncpy and wcsncpy are not always safer. Developers should consider using safer alternatives like strcpy_s, wcscpy_s, strlcpy, or std::string.

Using "strncpy" or "wcsncpy" is security-sensitive

Type: Bug **Volume:** □□□□

The functions strncpy and wcsncpy, designed for fixed-length strings, may result in non-null-terminated strings and have security risks. Safer alternatives, strncpy_s and wcsncpy_s, are available, but introduce overhead and require error handling. Use std::string for simpler and less error-prone string manipulation.

Non-standard characters should not occur in header file names in "#include" directives

Type: Bug **Volume:** □□□

The behavior of using ', \, ", //, or /* characters within the file names enclosed by double quotes or angle brackets in an #include directive is implementation dependent.

Why you should avoid using unsafe string handling functions

Using ‘strlen’ and ‘wcslen’ in C++ creates serious security vulnerabilities when strings lack proper null-termination. These functions blindly scan memory until finding a null terminator, potentially accessing invalid memory and triggering undefined behavior. This leads to buffer overflows and information leaks that attackers can exploit to access sensitive data or execute malicious code. Such vulnerabilities often remain undetected during testing but become attack vectors in production. Safer alternatives like ‘strlen_s’ or ‘std::string’ provide boundary checking that prevents these dangerous memory access issues by design.

The most common maintainability issues

Macros should not be used to define constants

Volume: □□□□□

Macros do not respect type or scoping rules. Developers should replace them with constants (constexpr or const) or enums.

"nullptr" should be used to denote the null pointer

Volume: □□□□

Denoting a null pointer with the integer literal 0 or the NULL macro creates ambiguity. C++11 introduced the keyword nullptr to unambiguously refer to the null pointer, and it should be used systematically.

C-style array should not be used

Volume: □□□

C-style arrays are inconvenient due to their fixed size and potential for memory mismanagement. The C++ standard library offers better alternatives: std::array for fixed-size arrays, std::vector for variable-size arrays, and std::string for character strings.

A variable which is not modified shall be const qualified

Volume: □□

A variable that is not modified after its initialization should be declared as const. This practice enhances code readability and safety by explicitly stating that the variable's value will not change, preventing unintended modifications and allowing the compiler to perform optimizations.

Implicit casts should not lower precision

Severity: MEDUIM

Narrowing conversions happen when converting to a type that can't hold all the original values, potentially losing information (for example, floats converted to ints). Developers should choose a destination type that can fit all source values, or by using an explicit conversion.

Why you shouldn’t use macros to define constants

Macro constants in C++ bypass critical language safeguards through simple text substitution. Unlike ‘constexpr’ or ‘const’ variables, macros lack type checking, allowing invalid operations that compilers cannot detect. They also ignore scope boundaries, causing name collisions across the codebase. This makes debugging extremely difficult since error messages reference expanded code rather than the original definition. Additionally, macros aren't visible in debuggers and can't participate in type-dependent operations. Modern C++ alternatives provide the same compile-time benefits while maintaining proper type safety, scope respect, and meaningful error messages.

PHP

Sonar’s dataset for PHP contained analysis of roughly half a billion lines of code. We found about 21 million issues across those scans (about 37,000 issues per million lines of PHP code).

The most common reliability and security issues

Method visibility should be explicitly declared

Type: Bug **Volume:** □□□□□□□□

Explicitly declaring method visibility improves code readability and maintainability, and promotes encapsulation. Access modifiers include private (access within the same class), protected (access to the class and its child classes), and public (unfettered access by all, default).

"require_once" and "include_once" should be used instead of "require" and "include"

Type: Bug **Volume:** □□□□

The PHP functions require, require_once, include, and include_once all include one file within another, but require_once and include_once ensure that a given file is only included once. Since including the same file multiple times could have unpredictable results, the _once versions are preferred.

Variables should be initialized before use

Type: Bug **Volume:** □□□

PHP doesn't require variable initialization, but it's a bad practice due to potential confusion and interpreter warnings.

The output of functions that don't return anything should not be used

Type: Bug **Volume:** □□

Assigning the output of a function that doesn't return a value to a variable, or passing it to another function, is likely a bug. This is because the function was probably not intended to be used in this way, given its lack of an output.

The number of arguments passed to a function should match the number of parameters

Type: Bug **Volume:** □□

Calling a function or a method with fewer or more arguments than expected will raise a TypeError. This is usually a bug and should be fixed. Developers should add default values or missing arguments if there are too few, and add parameters or remove arguments if there are too many.

Why you should explicitly declare method visibility

Omitting visibility declarations in PHP methods undermines code quality and security. When visibility modifiers aren't explicitly specified, methods default to public access—potentially exposing internal implementation details that should remain private. This breaks encapsulation, forcing developers to guess the intended access patterns rather than having them clearly documented in code. Without clear visibility boundaries, maintenance becomes harder as it's difficult to distinguish between intentionally public APIs and implementation details. This ambiguity increases the risk of security vulnerabilities and makes refactoring more error-prone, as developers may inadvertently rely on methods never intended for external use.

The most common maintainability issues

Lines should not end with trailing whitespaces

Volume: □□□□□□□

Trailing whitespaces bring no information, may generate noise when comparing different versions of the same file, and they can create bugs when they appear after a \ marking a line continuation. Developers should systematically remove them.

String literals should not be duplicated

Volume: □□□

Duplicated string literals make the process of refactoring complex and error-prone, as any change would need to be propagated on all occurrences. Developers should use constants to replace the duplicated string literals, since they can be updated once and referenced from many places.

Function names should comply with a naming convention

Volume: □□□

Shared naming conventions allow teams to collaborate efficiently. This rule raises an issue when a function name does not match a provided regular expression.

Control structures should use curly braces

Volume: □□

Omitting curly braces, while technically correct, can be misleading and error-prone during maintenance. Developers should add curly braces to improve readability and robustness.

Field names should comply with a naming convention

Volume: □□

Shared naming conventions allow teams to collaborate efficiently. This rule raises an issue when a field name does not match a provided regular expression.

Why you should avoid trailing whitespaces

Trailing whitespace in PHP code creates subtle but potentially significant problems. These invisible characters pollute version control diffs, generating noise that obscures meaningful changes during code reviews. More critically, whitespace appearing after line continuation characters (\) breaks line joining functionality, causing syntax errors or logic bugs that are extremely difficult to spot. These seemingly minor characters can lead to production failures with complex debugging paths. Additionally, inconsistent trailing whitespace affects automated formatting tools, triggering unnecessary changes across files and complicating collaboration. Systematic removal of trailing whitespace improves code clarity and streamlines the development process.

Conclusion

This report is intended to be a first step towards increased transparency around some of the most common security issues that can be found in the source code being actively written and maintained today. It underscores the need for solutions that facilitate automated code review, like SonarQube, to intercept critical issues so they don't escape into production environments.

As our community moves increasingly towards using AI to augment software development and dramatically increase the rate at which new code is created, we think this assessment of the state of code they are analyzing with Sonar will help to highlight frequently-occurring potential failure points and help developers strengthen the value of the code they create.

About Sonar

Sonar helps developers accelerate productivity, improves code security and code quality, and supports organizations in meeting compliance requirements while embracing AI technologies. The SonarQube platform, used by 7M+ developers worldwide, analyzes all code – developer-written, AI-generated, and third-party open source code – supercharging developers to build better applications, faster.

Sonar provides code review and assurance, inherently applies secure-by-design principles, fixes issues in code before they become a problem, and enforces policy standards – all while improving the developer experience. Sonar is trusted by the world's most innovative companies and is considered the industry standard for integrated code quality and code security. Today, Sonar is used by 400K organizations, including the DoD, Microsoft, NASA, Mastercard, Siemens, and T-Mobile.

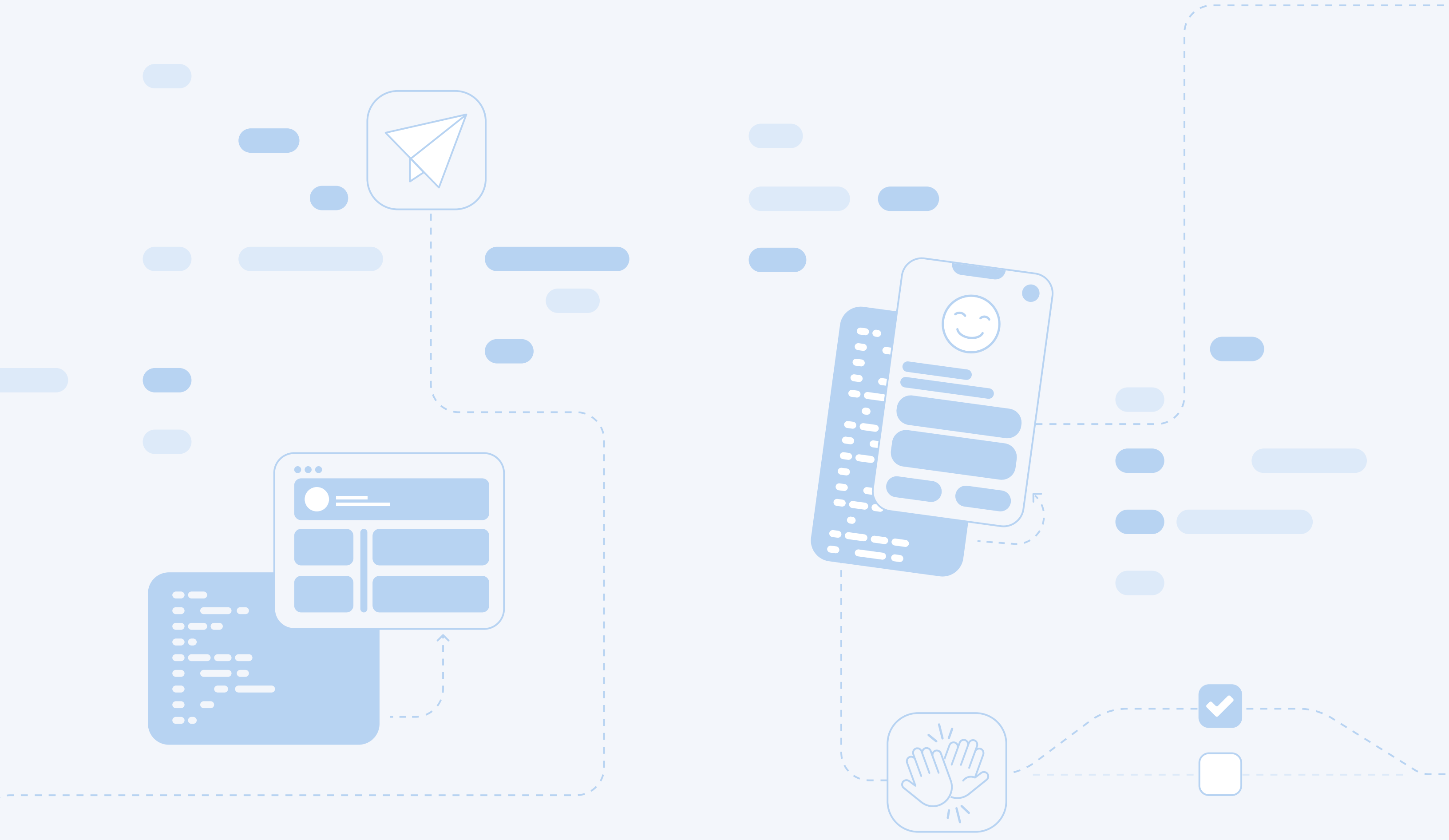
Trusted by over 7M developers and 400K organizations

Microsoft NASA NVIDIA Pfizer Mercedes-Benz

AIRFRANCE BARCLAYS COSTCO WHOLESALE Johnson&Johnson Santander

Learn more at sonar.com

Sonar



sonar.com