What the Fix? A Study of ASATs Rule Documentation

ICPC 2024

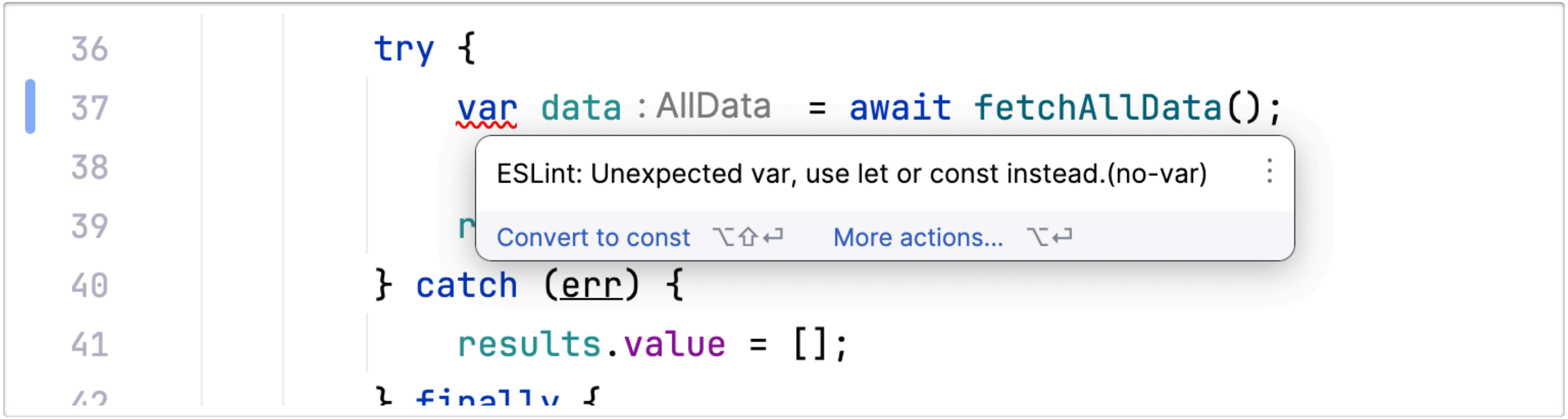
Corentin LATAPPY Thomas DEGUEULE Jean-Rémy FALLERI

Romain ROBBES Xavier BLANC Cédric TEYTON

Sophia de Mello Breyner Andresen Room

Tuesday, April 16, 2024

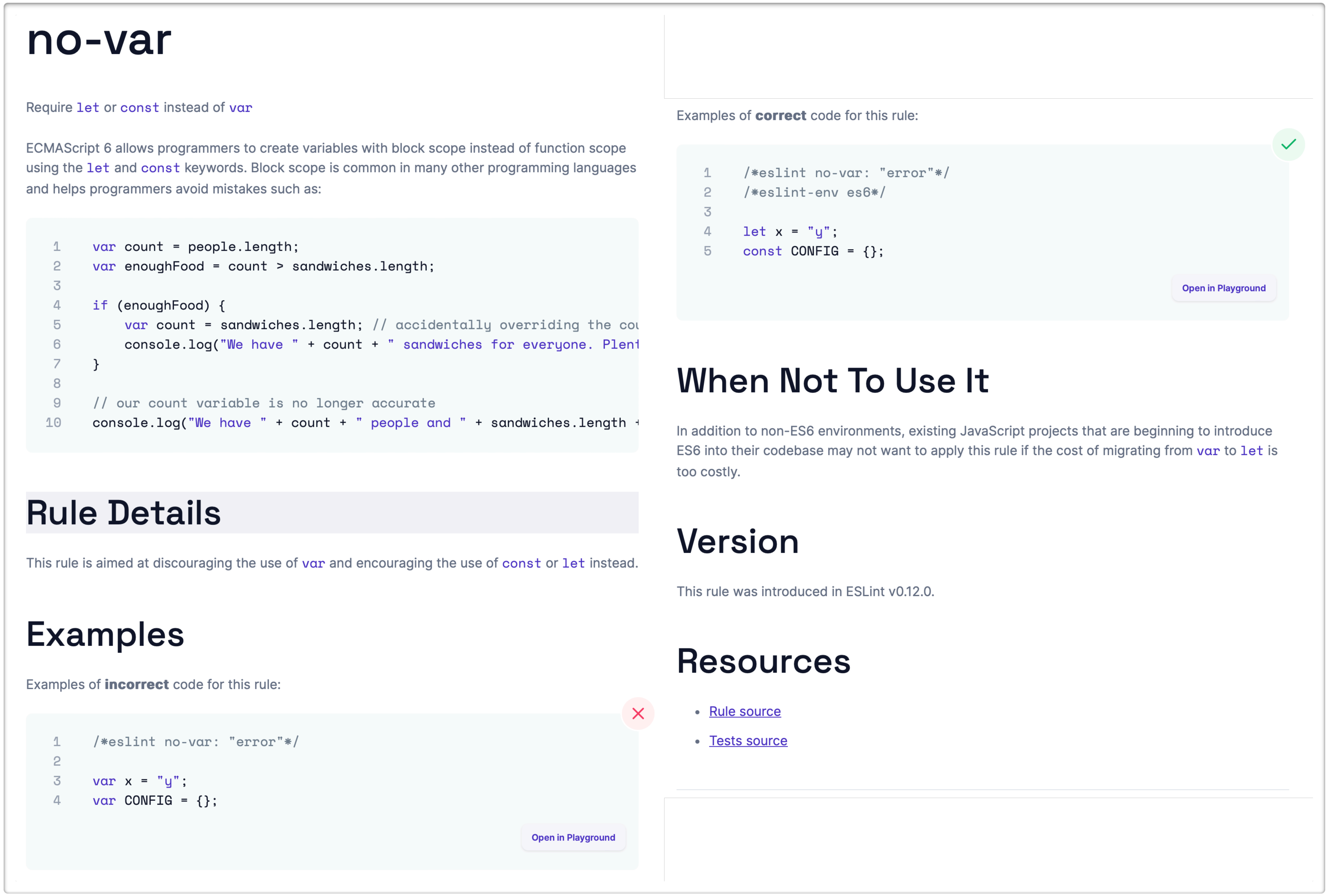
What’s an ASAT, aka Linter



```
    try {
        var data : AllData  = await fetchAllData();
        
        r
    } catch (err) {
        results.value = [];
    } finally {
```

ESLint: Unexpected var, use let or const instead.(no-var)

no-var rule from *ESLint* on a JS snippet



no-var

Require `let` or `const` instead of `var`

ECMAScript 6 allows programmers to create variables with block scope instead of function scope using the `let` and `const` keywords. Block scope is common in many other programming languages and helps programmers avoid mistakes such as:

```
var count = people.length;
var enoughFood = count > sandwiches.length;

if (enoughFood) {
    var count = sandwiches.length; // accidentally overriding the cou
    console.log("We have " + count + " sandwiches for everyone. Plent
}

// our count variable is no longer accurate
console.log("We have " + count + " people and " + sandwiches.length +
```

Rule Details

This rule is aimed at discouraging the use of `var` and encouraging the use of `const` or `let` instead.

Examples

Examples of **incorrect** code for this rule:

```
/*eslint no-var: "error"*/

var x = "y";
var CONFIG = {};
```

Open in Playground

Examples of **correct** code for this rule:

```
/*eslint no-var: "error"*/
/*eslint-env es6*/

let x = "y";
const CONFIG = {};
```

Open in Playground

When Not To Use It

In addition to non-ES6 environments, existing JavaScript projects that are beginning to introduce ES6 into their codebase may not want to apply this rule if the cost of migrating from `var` to `let` is too costly.

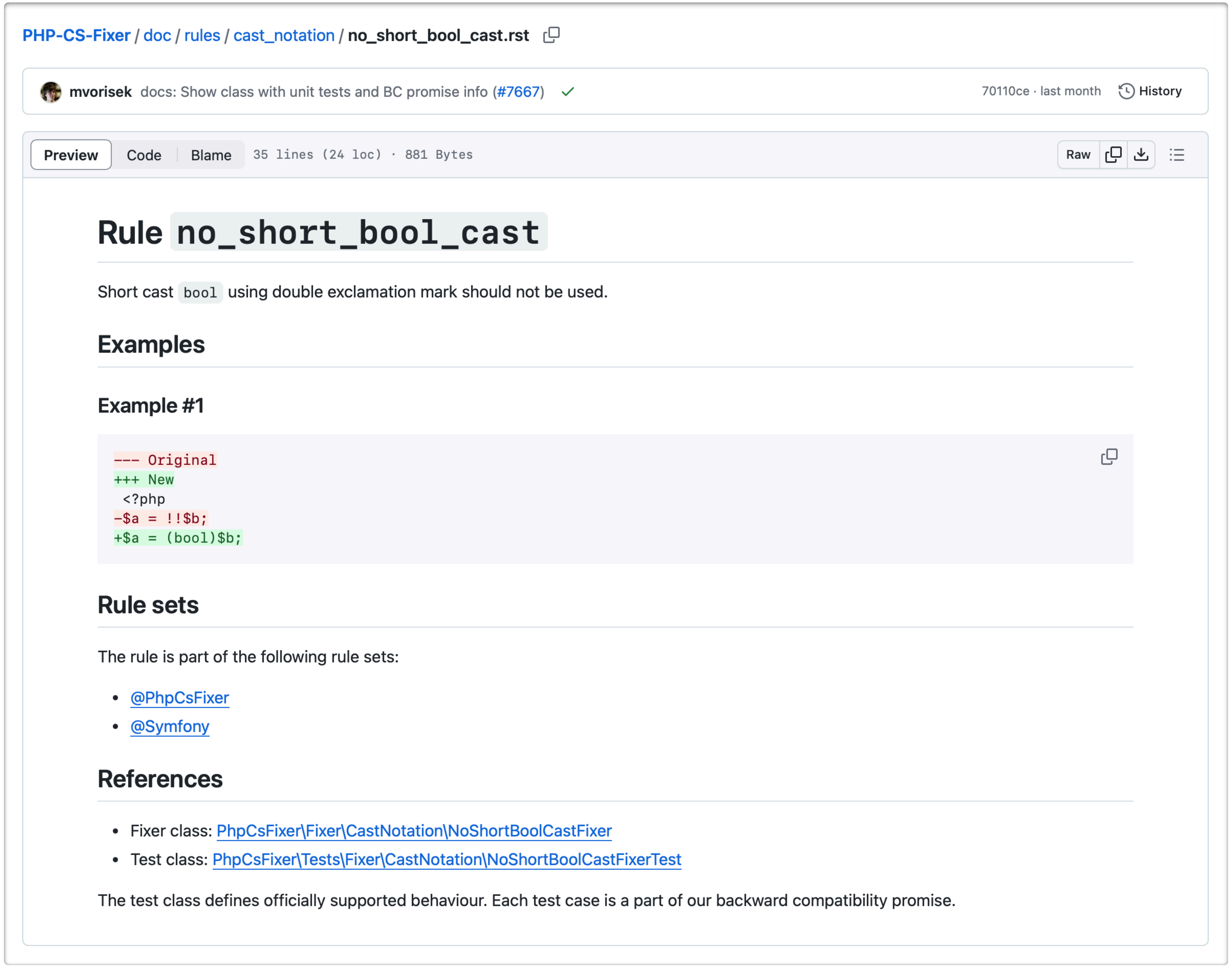
Version

This rule was introduced in ESLint v0.12.0.

Resources

- Rule source
- Tests source

Rule *no-var* from *ESLint*


PHP-CS-Fixer / doc / rules / cast_notation / no_short_bool_cast.rst

mvorisek docs: Show class with unit tests and BC promise info (#7667) ✓ 70110ce · last month History

Rule `no_short_bool_cast`

Short cast `bool` using double exclamation mark should not be used.

Examples

Example #1

```
--- Original
+++ New
 <?php
-$a = !!$b;
+$a = (bool)$b;
```

Rule sets

The rule is part of the following rule sets:

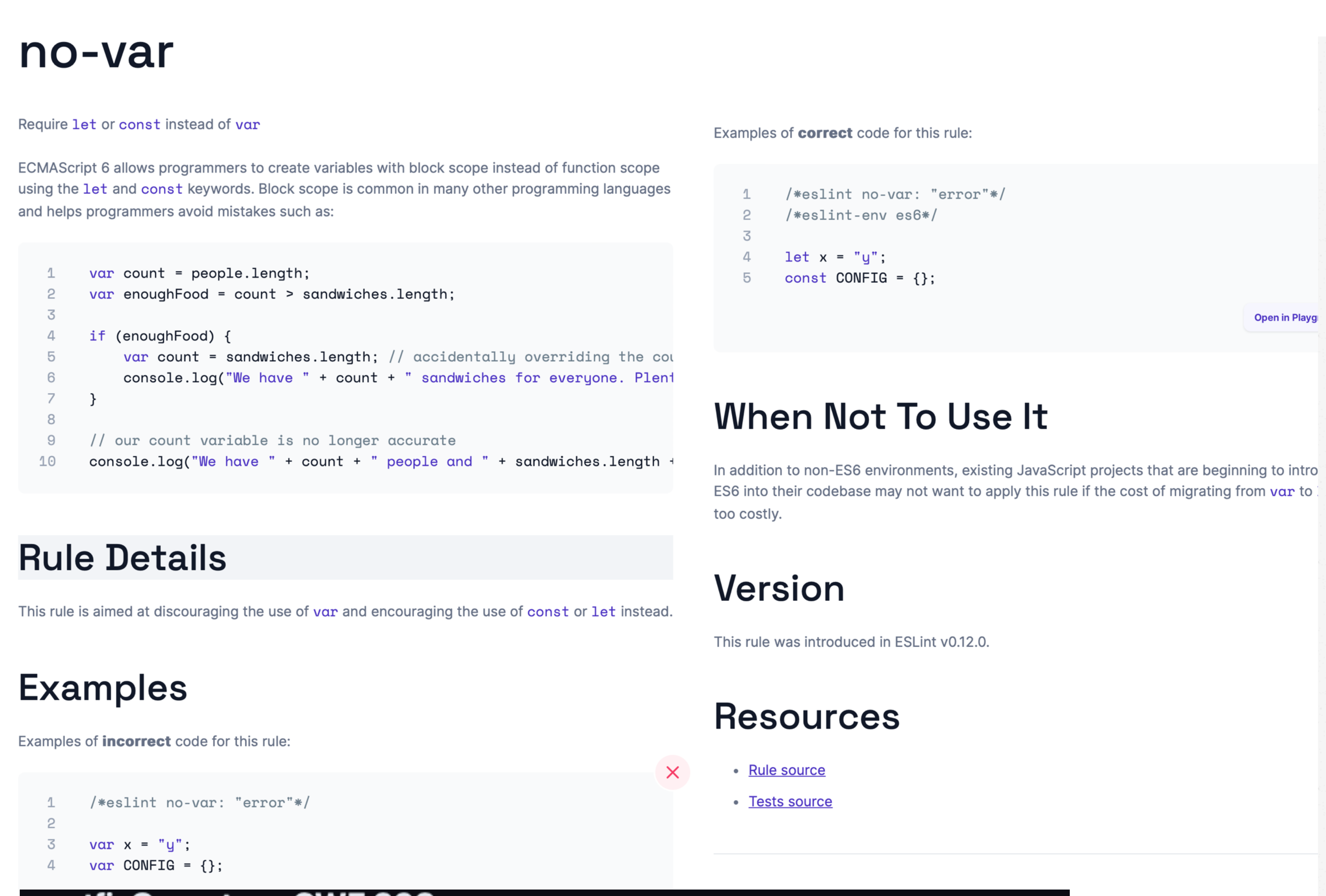
- @PhpCsFixer
- @Symfony

References

- Fixer class: PhpCsFixer\Fixer\CastNotation\NoShortBoolCastFixer
- Test class: PhpCsFixer\Tests\Fixer\CastNotation\NoShortBoolCastFixerTest

The test class defines officially supported behaviour. Each test case is a part of our backward compatibility promise.

Rule *no_short_bool_cast* from PHP CS Fixer


no-var

Require `let` or `const` instead of `var`

ECMAScript 6 allows programmers to create variables with block scope instead of function scope using the `let` and `const` keywords. Block scope is common in many other programming languages and helps programmers avoid mistakes such as:

```
var count = people.length;
var enoughFood = count > sandwiches.length;

if (enoughFood) {
    var count = sandwiches.length; // accidentally overriding the cou
    console.log("We have " + count + " sandwiches for everyone. Plent
}

// our count variable is no longer accurate
console.log("We have " + count + " people and " + sandwiches.length +
```

Rule Details

This rule is aimed at discouraging the use of `var` and encouraging the use of `const` or `let` instead.

Examples

Examples of **incorrect** code for this rule:

```
/*eslint no-var: "error"*/

var x = "y";
var CONFIG = {};
```

Examples of **correct** code for this rule:

```
/*eslint no-var: "error"*/
/*eslint-env es6*/

let x = "y";
const CONFIG = {};
```

Open in Playg

When Not To Use It

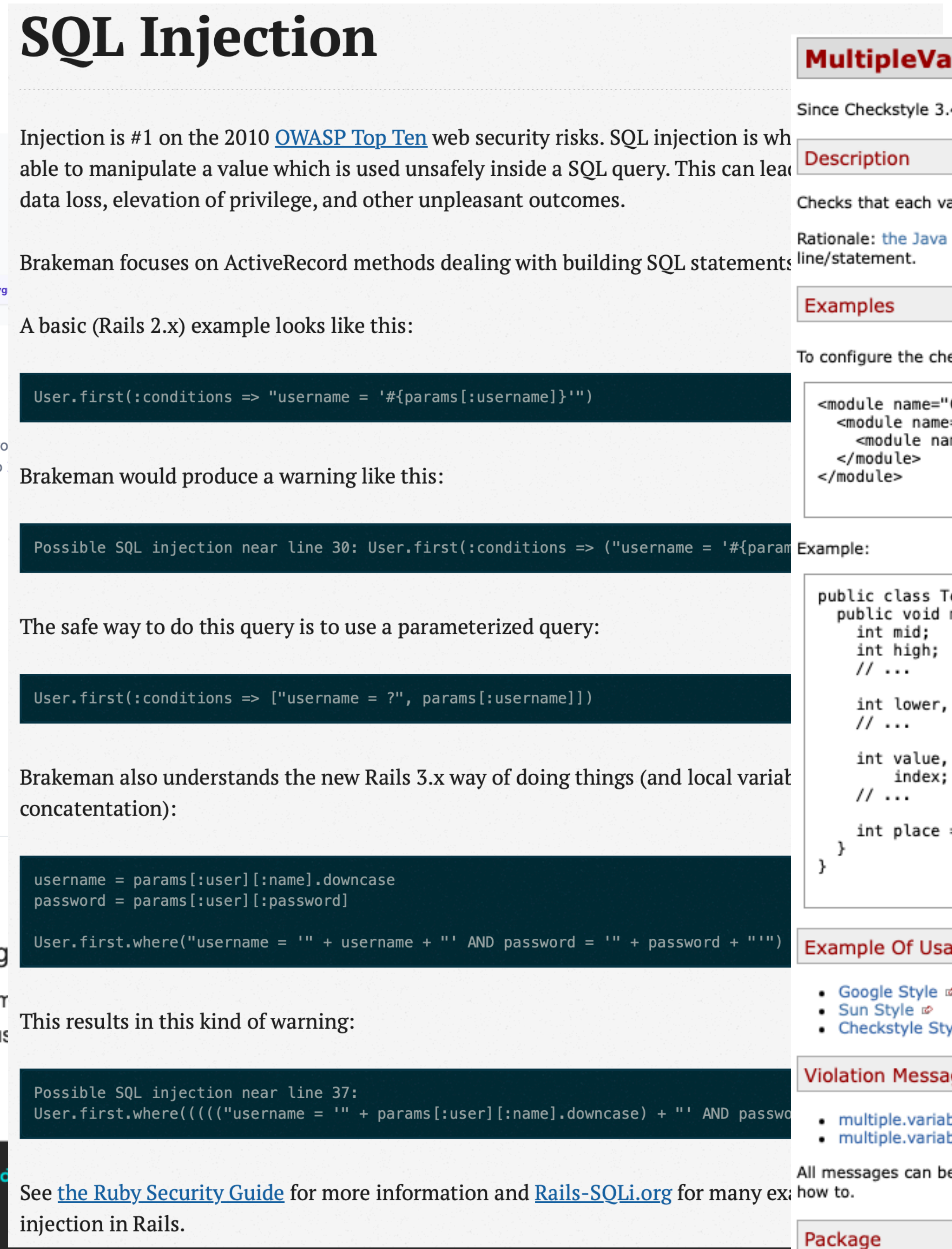
In addition to non-ES6 environments, existing JavaScript projects that are beginning to intro ES6 into their codebase may not want to apply this rule if the cost of migrating from `var` to too costly.

Version

This rule was introduced in ESLint v0.12.0.

Resources

- Rule source
- Tests source


SQL Injection

Injection is #1 on the 2010 OWASP Top Ten web security risks. SQL injection is wh able to manipulate a value which is used unsafely inside a SQL query. This can lead data loss, elevation of privilege, and other unpleasant outcomes.

Brakeman focuses on ActiveRecord methods dealing with building SQL statements

A basic (Rails 2.x) example looks like this:

```
User.first(:conditions => "username = '#{params[:username]}'")
```

Brakeman would produce a warning like this:

```
Possible SQL injection near line 30: User.first(:conditions => ("username = '#{param
```

The safe way to do this query is to use a parameterized query:

```
User.first(:conditions => ["username = ?", params[:username]])
```

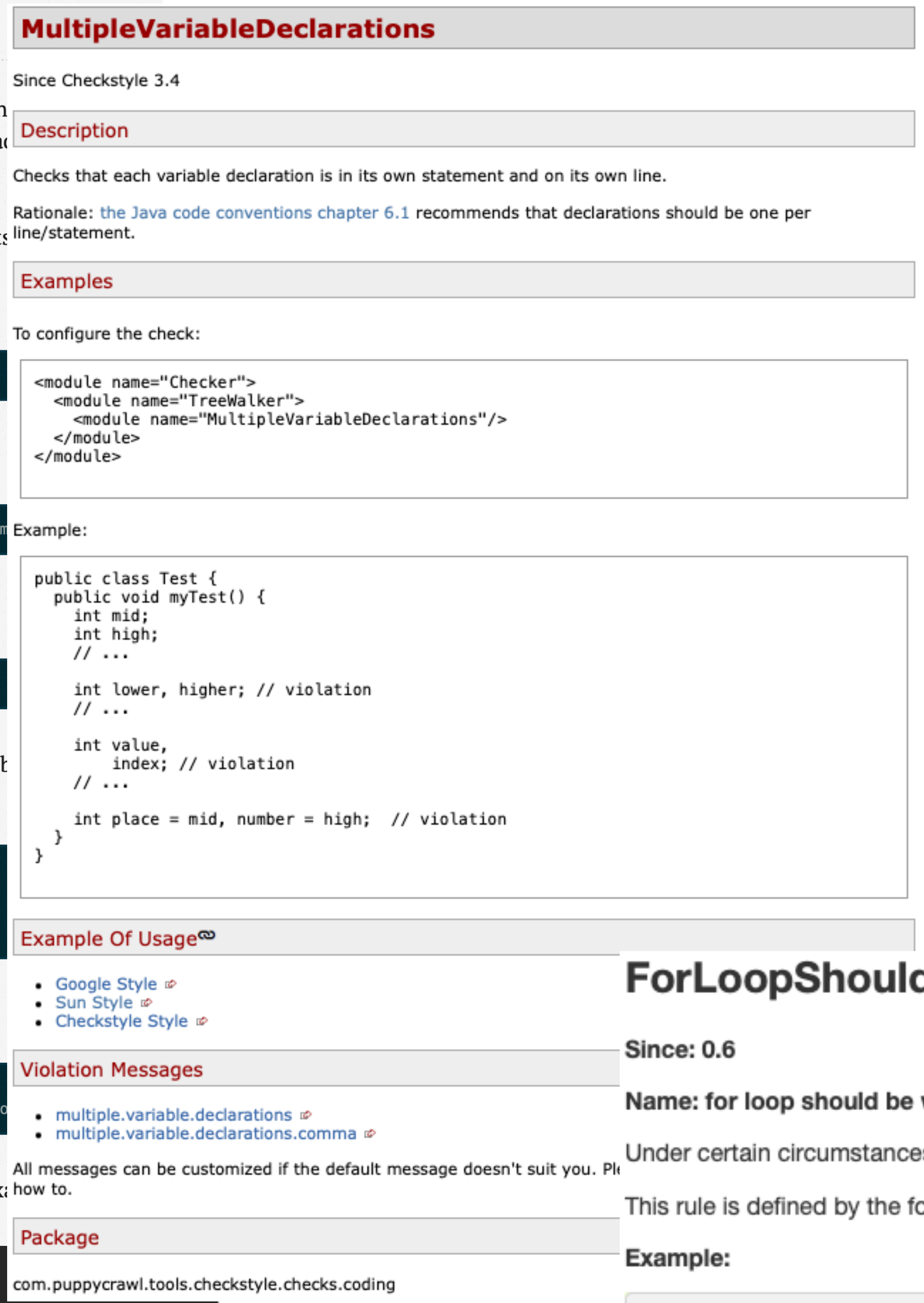
Brakeman also understands the new Rails 3.x way of doing things (and local variab concatentation):

```
username = params[:user][:name].downcase
password = params[:user][:password]

User.first.where("username = '" + username + "' AND password = '" + password + "'")
```

This results in this kind of warning:

```
Possible SQL injection near line 37:
User.first.where((((("username = '" + params[:user][:name].downcase) + "' AND passwo
```

See the Ruby Security Guide for more information and Rails-SQLi.org for many exa injection in Rails.


MultipleVariableDeclarations

Since Checkstyle 3.4

Description

Checks that each variable declaration is in its own statement and on its own line.

Rationale: the Java code conventions chapter 6.1 recommends that declarations should be one per line/statement.

Examples

To configure the check:

```
<module name="Checker">
  <module name="TreeWalker">
    <module name="MultipleVariableDeclarations"/>
  </module>
</module>
```

Example:

```
public class Test {
  public void myTest() {
    int mid;
    int high;
    // ...

    int lower, higher; // violation
    // ...

    int value,
        index; // violation
    // ...

    int place = mid, number = high;  // violation
  }
}
```

Example Of Usage

- Google Style
- Sun Style
- Checkstyle Style

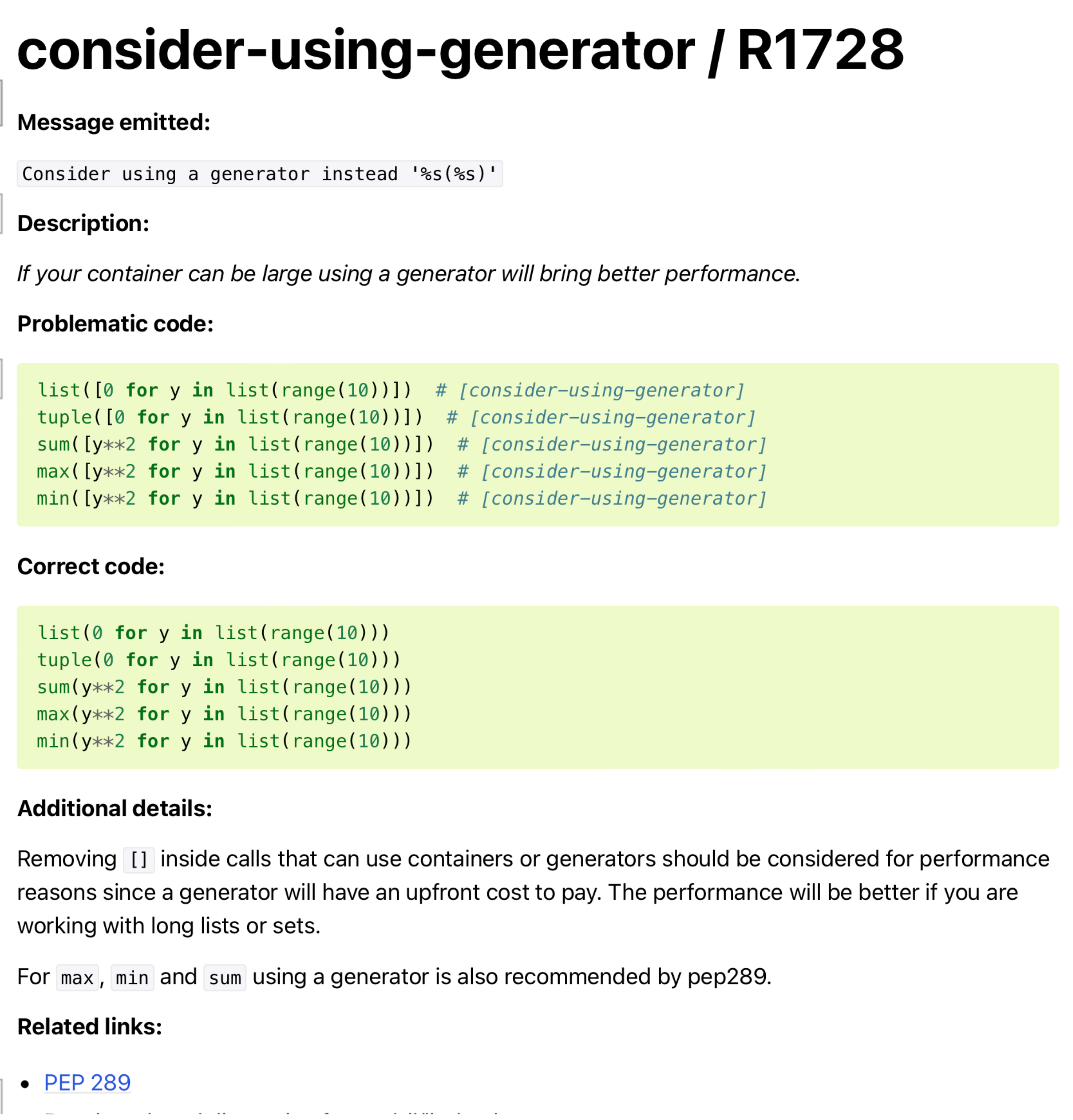
Violation Messages

- multiple.variable.declarations
- multiple.variable.declarations.comma

All messages can be customized if the default message doesn't suit you. Pl how to.

Package

com.puppycrawl.tools.checkstyle.checks.coding


consider-using-generator / R1728

Message emitted:

`Consider using a generator instead '%s(%s)'`

Description:

If your container can be large using a generator will bring better performance.

Problematic code:

```
list([0 for y in list(range(10))])  # [consider-using-generator]
tuple([0 for y in list(range(10))])  # [consider-using-generator]
sum([y**2 for y in list(range(10))])  # [consider-using-generator]
max([y**2 for y in list(range(10))])  # [consider-using-generator]
min([y**2 for y in list(range(10))])  # [consider-using-generator]
```

Correct code:

```
list(0 for y in list(range(10)))
tuple(0 for y in list(range(10)))
sum(y**2 for y in list(range(10)))
max(y**2 for y in list(range(10)))
min(y**2 for y in list(range(10)))
```

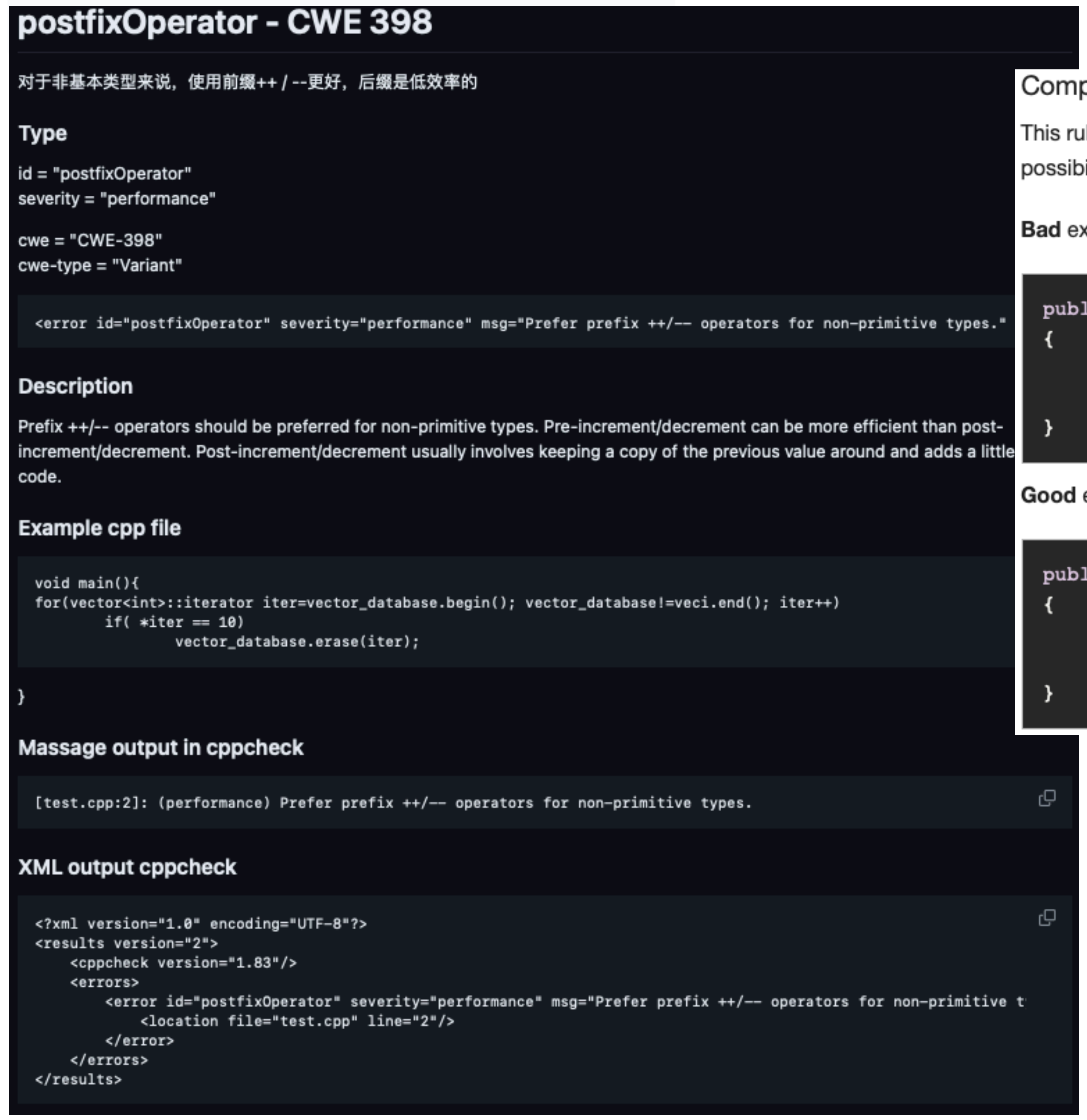
Additional details:

Removing `[]` inside calls that can use containers or generators should be considered for performance reasons since a generator will have an upfront cost to pay. The performance will be better if you are working with long lists or sets.

For `max`, `min` and `sum` using a generator is also recommended by pep289.

Related links:

- PEP 289


postfixOperator - CWE 398

对于非基本类型来说，使用前缀++ / --更好，后缀是低效率的

Type

id = "postfixOperator"
severity = "performance"

cwe = "CWE-398"
cwe-type = "Variant"

```
<error id="postfixOperator" severity="performance" msg="Prefer prefix ++/-- operators for non-primitive types."
```

Description

Prefix ++/-- operators should be preferred for non-primitive types. Pre-increment/decrement can be more efficient than post-increment/decrement. Post-increment/decrement usually involves keeping a copy of the previous value around and adds a little code.

Example cpp file

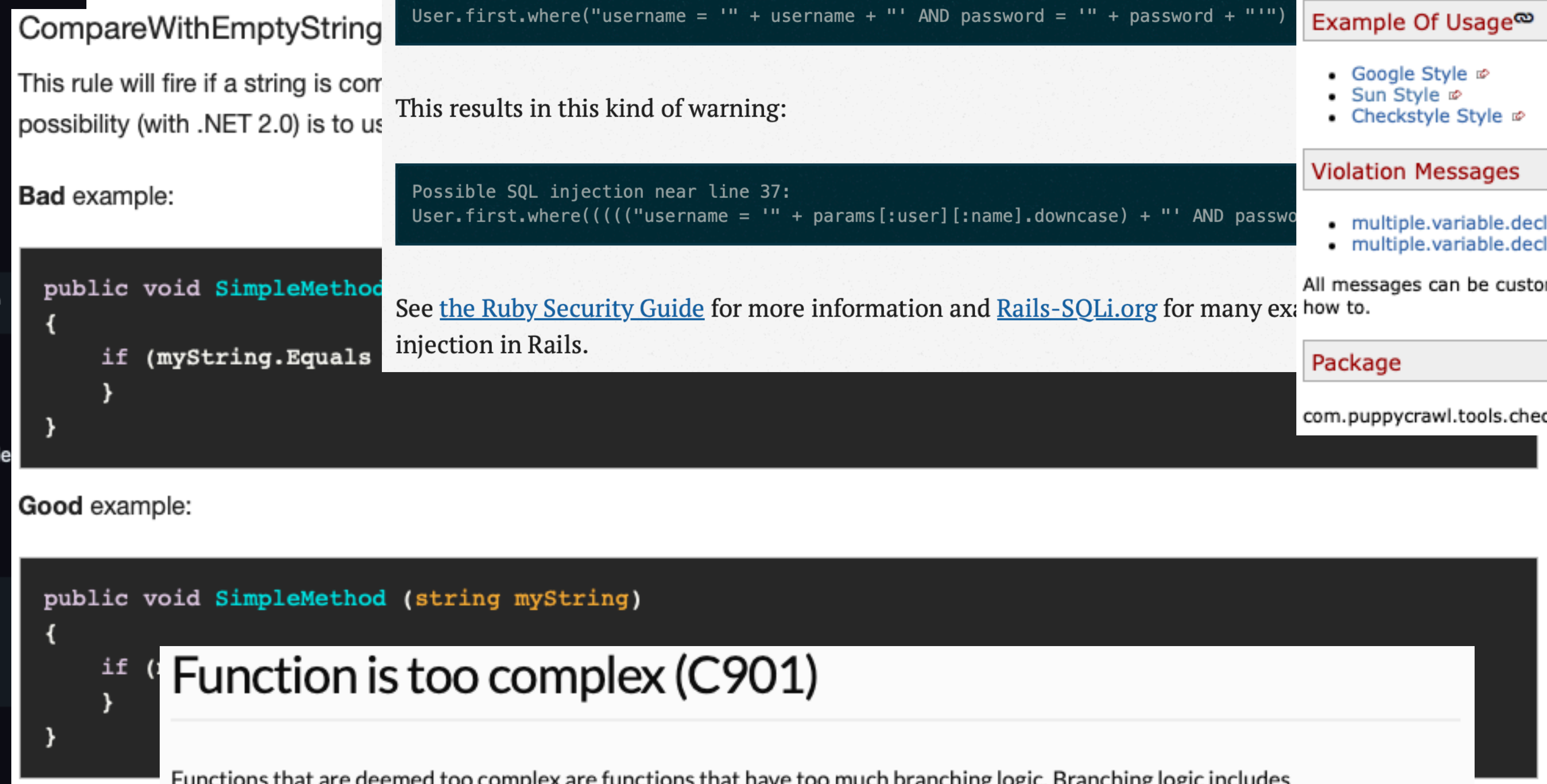
```
void main(){
for(vector<int>::iterator iter=vector_database.begin(); vector_database!=veci.end(); iter++)
        if( *iter == 10)
                vector_database.erase(iter);

}
```

Massage output in cppcheck

```
[test.cpp:2]: (performance) Prefer prefix ++/-- operators for non-primitive types.
```

XML output cppcheck

```
<?xml version="1.0" encoding="UTF-8"?>
<results version="2">
    <cppcheck version="1.83"/>
    <errors>
        <error id="postfixOperator" severity="performance" msg="Prefer prefix ++/-- operators for non-primitive t
            <location file="test.cpp" line="2"/>
        </error>
    </errors>
</results>
```


CompareWithEmptyString

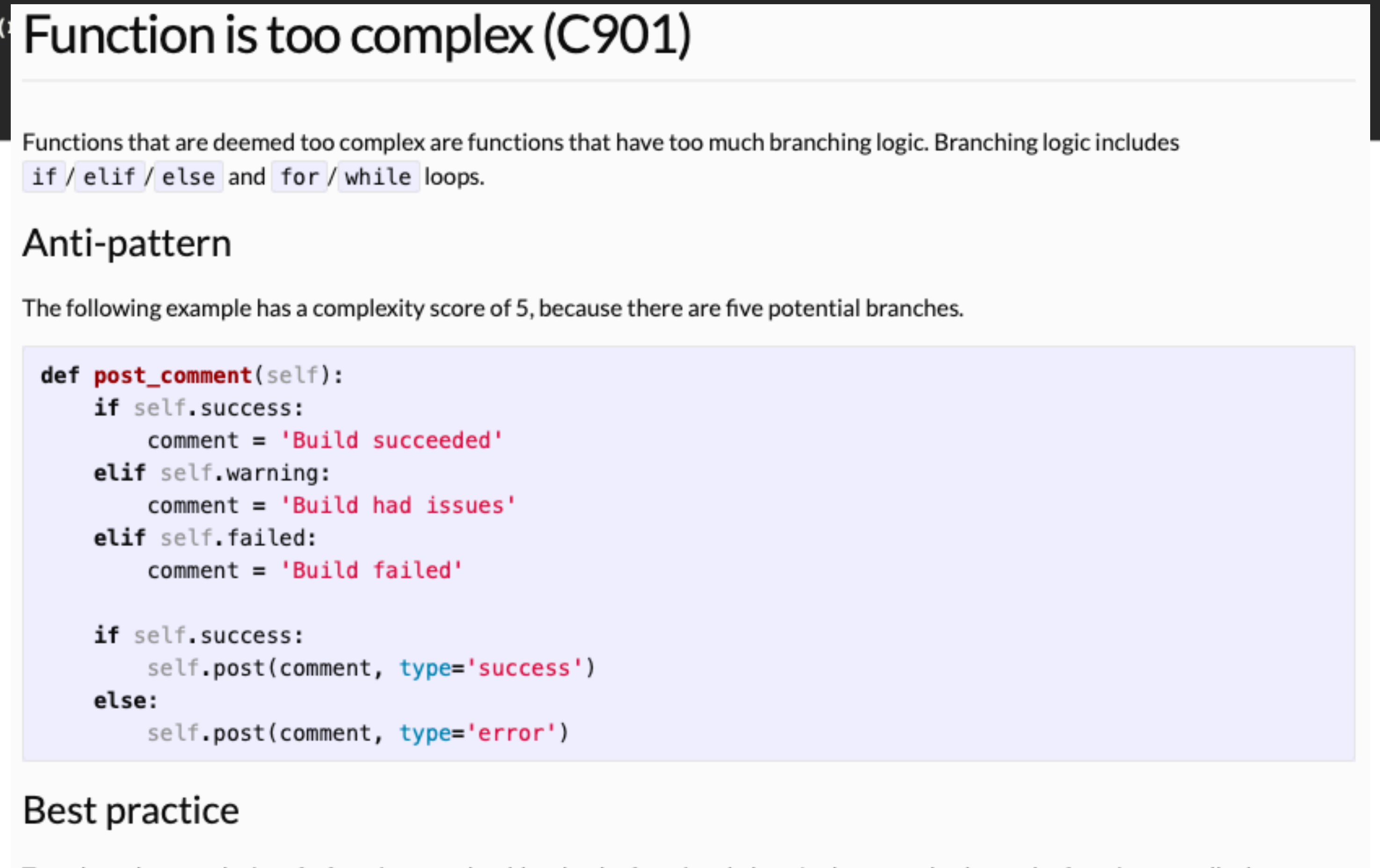
This rule will fire if a string is com possibility (with .NET 2.0) is to us

Bad example:

```
public void SimpleMethod
{
    if (myString.Equals
    }
}
```

Good example:

```
public void SimpleMethod (string myString)
{
    if (
    }
}
```


Function is too complex (C901)

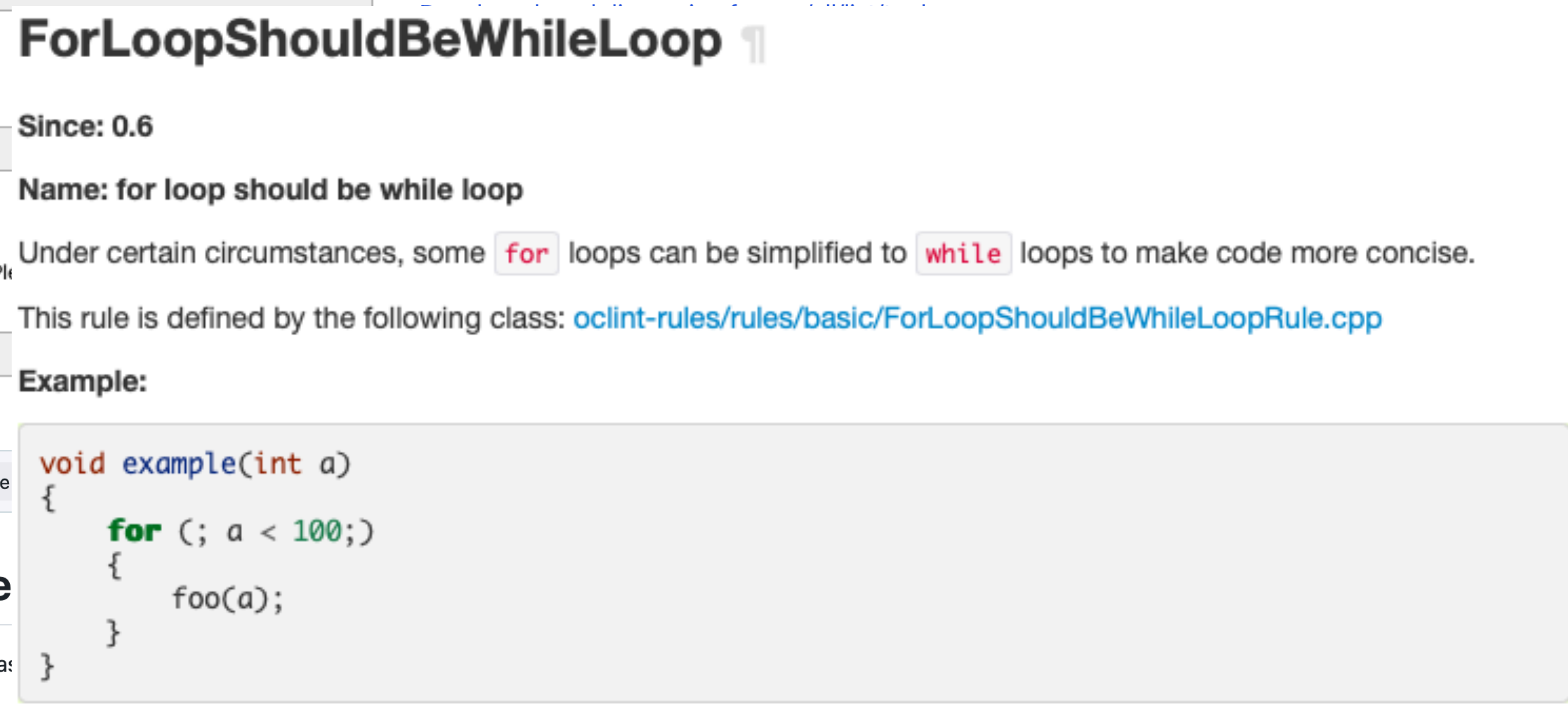
Functions that are deemed too complex are functions that have too much branching logic. Branching logic includes `if` / `elif` / `else` and `for` / `while` loops.

Anti-pattern

The following example has a complexity score of 5, because there are five potential branches.

```
def post_comment(self):
    if self.success:
        comment = 'Build succeeded'
    elif self.warning:
        comment = 'Build had issues'
    elif self.failed:
        comment = 'Build failed'

    if self.success:
        self.post(comment, type='success')
    else:
        self.post(comment, type='error')
```

Best practice


ForLoopShouldBeWhileLoop ¶

Since: 0.6

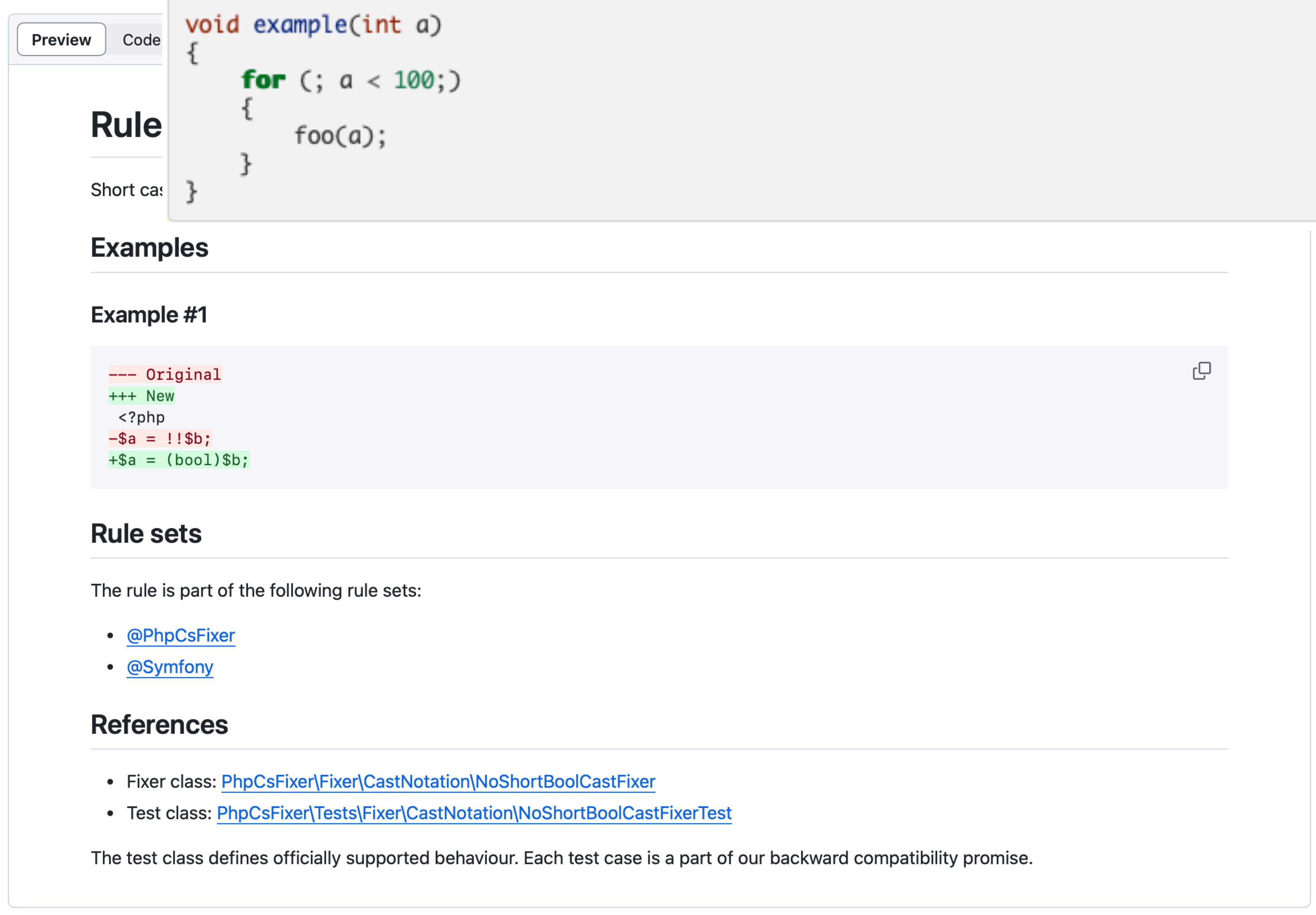
Name: for loop should be while loop

Under certain circumstances, some `for` loops can be simplified to `while` loops to make code more concise.

This rule is defined by the following class: oclint-rules/rules/basic/ForLoopShouldBeWhileLoopRule.cpp

Example:

```
void example(int a)
{
    for (; a < 100;)
    {
        foo(a);
    }
}
```


Preview Code

Rule

Short cas

Examples

Example #1

```
--- Original
+++ New
 <?php
-$a = !!$b;
+$a = (bool)$b;
```

Rule sets

The rule is part of the following rule sets:

- @PhpCsFixer
- @Symfony

References

- Fixer class: PhpCsFixer\Fixer\CastNotation\NoShortBoolCastFixer
- Test class: PhpCsFixer\Tests\Fixer\CastNotation\NoShortBoolCastFixerTest

The test class defines officially supported behaviour. Each test case is a part of our backward compatibility promise.

What makes a good rule documentation?

no-var

Require `let` or `const` instead of `var`

ECMAScript 6 allows programmers to create variables with block scope instead of function scope using the `let` and `const` keywords. Block scope is common in many other programming languages and helps programmers avoid mistakes such as:

```
var count = people.length;
var enoughFood = count > sandwiches.length;

if (enoughFood) {
    var count = sandwiches.length; // accidentally overriding the co
    console.log("We have " + count + " sandwiches for everyone. Plent
}

// our count variable is no longer accurate
console.log("We have " + count + " people and " + sandwiches.length +
```

Rule Details

This rule is aimed at discouraging the use of `var` and encouraging the use of `const` or `let` instead.

Examples

Examples of **incorrect** code for this rule:

```
/*eslint no-var: "error"*/

var x = "y";
var CONFIG = {};
```

Examples of **correct** code for this rule:

```
/*eslint no-var: "error"*/
/*eslint-env es6*/

let x = "y";
const CONFIG = {};
```

Open in Playg

When Not To Use It

In addition to non-ES6 environments, existing JavaScript projects that are beginning to intro ES6 into their codebase may not want to apply this rule if the cost of migrating from `var` to too costly.

Version

This rule was introduced in ESLint v0.12.0.

Resources

- Rule source
- Tests source

SQL Injection

Injection is #1 on the 2010 OWASP Top Ten web security risks. SQL injection is wh able to manipulate a value which is used unsafely inside a SQL query. This can lea data loss, elevation of privilege, and other unpleasant outcomes.

Brakeman focuses on ActiveRecord methods dealing with building SQL statements

A basic (Rails 2.x) example looks like this:

```
User.first(:conditions => "username = '#{params[:username]}'")
```

Brakeman would produce a warning like this:

```
Possible SQL injection near line 30: User.first(:conditions => ("username = '#{para
```

The safe way to do this query is to use a parameterized query:

```
User.first(:conditions => ["username = ?", params[:username]])
```

Brakeman also understands the new Rails 3.x way of doing things (and local variab concatenation):

```
username = params[:user][:name].downcase
password = params[:user][:password]

User.first.where("username = '" + username + "' AND password = '" + password + "'")
```

See the Ruby Security Guide for more information and Rails-SQLi.org for many exa injection in Rails.

MultipleVariableDeclarations

Since Checkstyle 3.4

Description

Checks that each variable declaration is in its own statement and on its own line.

Rationale: the Java code conventions chapter 6.1 recommends that declarations should be one per line/statement.

Examples

To configure the check:

```
<module name="Checker">
  <module name="TreeWalker">
    <module name="MultipleVariableDeclarations"/>
  </module>
</module>
```

Example:

```
public class Test {
  public void myTest() {
    int mid;
    int high;
    // ...

    int lower, higher; // violation
    // ...

    int value,
        index; // violation
    // ...

    int place = mid, number = high;  // violation
  }
}
```

Example Of Usage

- Google Style

All messages can be customized if the default message doesn't suit you. Pl how to.

Package

com.puppycrawl.tools.checkstyle.checks.coding

consider-using-generator / R1728

Message emitted:

`Consider using a generator instead '%s(%s)'`

Description:

If your container can be large using a generator will bring better performance.

Problematic code:

```
list([0 for y in list(range(10))])  # [consider-using-generator]
tuple([0 for y in list(range(10))])  # [consider-using-generator]
sum([y**2 for y in list(range(10))])  # [consider-using-generator]
max([y**2 for y in list(range(10))])  # [consider-using-generator]
min([y**2 for y in list(range(10))])  # [consider-using-generator]
```

Correct code:

```
list(0 for y in list(range(10)))
tuple(0 for y in list(range(10)))
sum(y**2 for y in list(range(10)))
max(y**2 for y in list(range(10)))
min(y**2 for y in list(range(10)))
```

Additional details:

Removing `[]` inside calls that can use containers or generators should be considered for performance reasons since a generator will have an upfront cost to pay. The performance will be better if you are working with long lists or sets.

For `max`, `min` and `sum` using a generator is also recommended by pep289.

Related links:

- PEP 289

postfixOperator - CWE 398

对于非基本类型来说，使用前缀++ / --更好，后缀是低效率的

Type

id = "postfixOperator"
severity = "performance"

cwe = "CWE-398"
cwe-type = "Variant"

```
<error id="postfixOperator" severity="performance" msg="Prefer prefix ++/-- operators for non-primitive types."
```

Description

Prefix ++/-- operators should be preferred for non-primitive types. Pre-increment/decrement can be more efficient than post-increment/decrement. Post-increment/decrement usually involves keeping a copy of the previous value around and adds a little code.

Example cpp file

```
void main(){
for(vector<int>::iterator iter=vector_database.begin(); vector_database!=veci.end(); iter++)
        if( *iter == 10)
                vector_database.erase(iter);

}
```

Massage output in cppcheck

```
[test.cpp:2]: (performance) Prefer prefix ++/-- operators for non-primitive types.
```

XML output cppcheck

```
<?xml version="1.0" encoding="UTF-8"?>
<results version="2">
    <cppcheck version="1.83"/>
    <errors>
        <error id="postfixOperator" severity="performance" msg="Prefer prefix ++/-- operators for non-primitive t
            <location file="test.cpp" line="2"/>
        </error>
    </errors>
</results>
```

CompareWithEmptyString

This rule will fire if a string is com

```
public void SimpleMetho
{
    if (myString.Equals
    }
}
```

Good example:

```
public void SimpleMethod (string myString)
{
    if (
    }
}
```

Function is too complex (C901)

Functions that are deemed too complex are functions that have too much branching logic. Branching logic includes `if` / `elif` / `else` and `for` / `while` loops.

Anti-pattern

The following example has a complexity score of 5, because there are five potential branches.

```
def post_comment(self):
    if self.success:
        comment = 'Build succeeded'
    elif self.warning:
        comment = 'Build had issues'
    elif self.failed:
        comment = 'Build failed'

    if self.success:
        self.post(comment, type='success')
    else:
        self.post(comment, type='error')
```

Best practice

ForLoopShouldBeWhileLoop

lified to `while` loops to make code more concise.

This rule is defined by the following class: oclint-rules/rules/basic/ForLoopShouldBeWhileLoopRule.cpp

Example:

```
void example(int a)
{
    for (; a < 100;)
    {
        foo(a);
    }
}
```

Preview Code

Rule

Short ca

Examples

Example #1

```
--- Original
+++ New
 <?php
-$a = !!$b;
+$a = (bool)$b;
```

Rule sets

The rule is part of the following rule sets:

- @PhpCsFixer
- @Symfony

References

- Fixer class: PhpCsFixer\Fixer\CastNotation\NoShortBoolCastFixer
- Test class: PhpCsFixer\Tests\Fixer\CastNotation\NoShortBoolCastFixerTest

The test class defines officially supported behaviour. Each test case is a part of our backward compatibility promise.

Documentation Analysis

Documentation Analysis

7 languages

Documentation Analysis

7 languages	14 tools + 2 multi-language

Documentation Analysis

7 languages | 14 tools + 2 multi-language

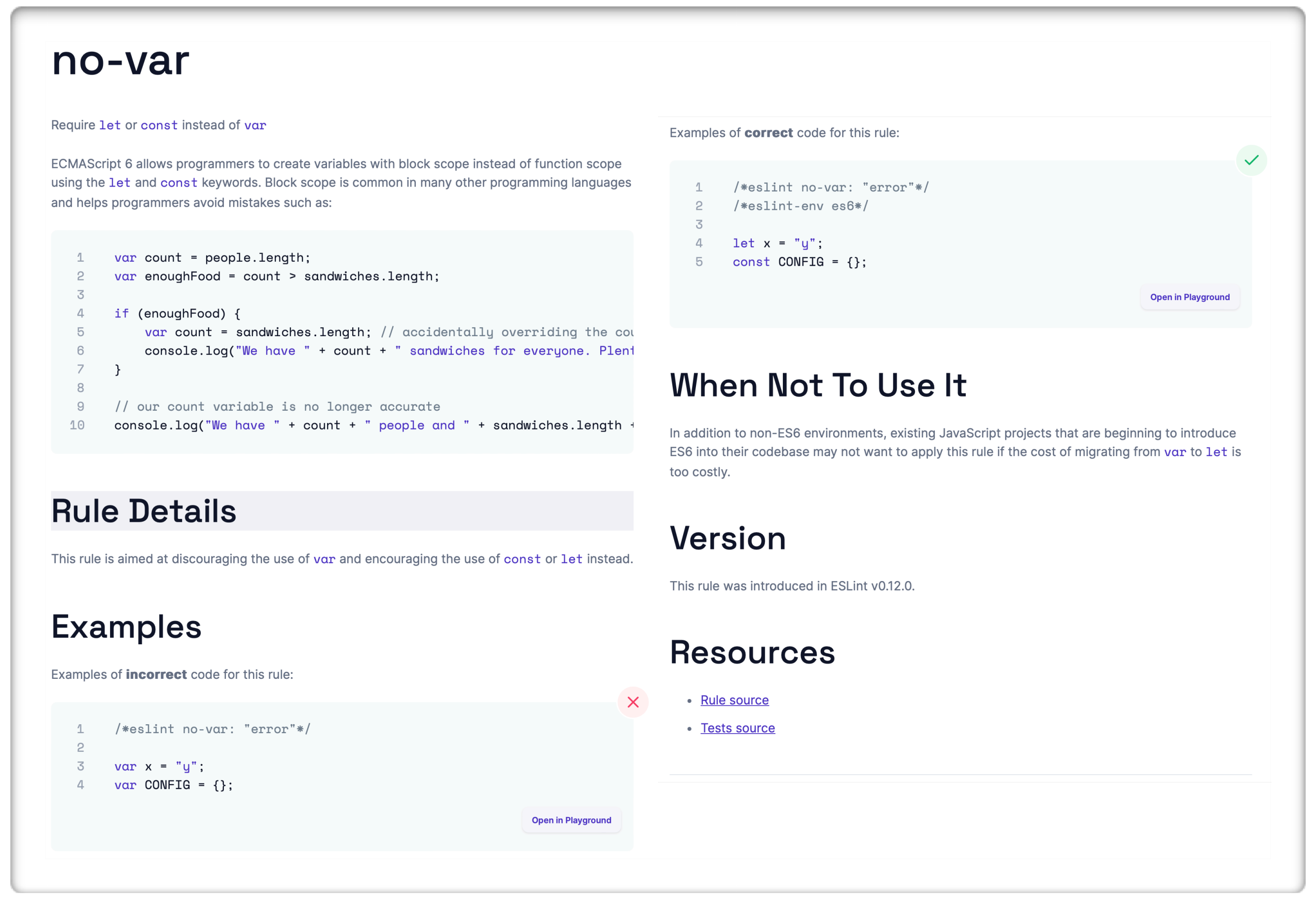
Pick 1 rule from tool documentation

Documentation Analysis

7 languages | 14 tools + 2 multi-language

Pick 1 rule from tool documentation

ESLint no-var



no-var

Require `let` or `const` instead of `var`

ECMAScript 6 allows programmers to create variables with block scope instead of function scope using the `let` and `const` keywords. Block scope is common in many other programming languages and helps programmers avoid mistakes such as:

```
var count = people.length;
var enoughFood = count > sandwiches.length;

if (enoughFood) {
    var count = sandwiches.length; // accidentally overriding the cou
    console.log("We have " + count + " sandwiches for everyone. Plent
}

// our count variable is no longer accurate
console.log("We have " + count + " people and " + sandwiches.length +
```

Rule Details

This rule is aimed at discouraging the use of `var` and encouraging the use of `const` or `let` instead.

Examples

Examples of **incorrect** code for this rule:

```
/*eslint no-var: "error"*/

var x = "y";
var CONFIG = {};
```

Open in Playground

Examples of **correct** code for this rule:

```
/*eslint no-var: "error"*/
/*eslint-env es6*/

let x = "y";
const CONFIG = {};
```

Open in Playground

When Not To Use It

In addition to non-ES6 environments, existing JavaScript projects that are beginning to introduce ES6 into their codebase may not want to apply this rule if the cost of migrating from `var` to `let` is too costly.

Version

This rule was introduced in ESLint v0.12.0.

Resources

- Rule source
- Tests source

Documentation Analysis

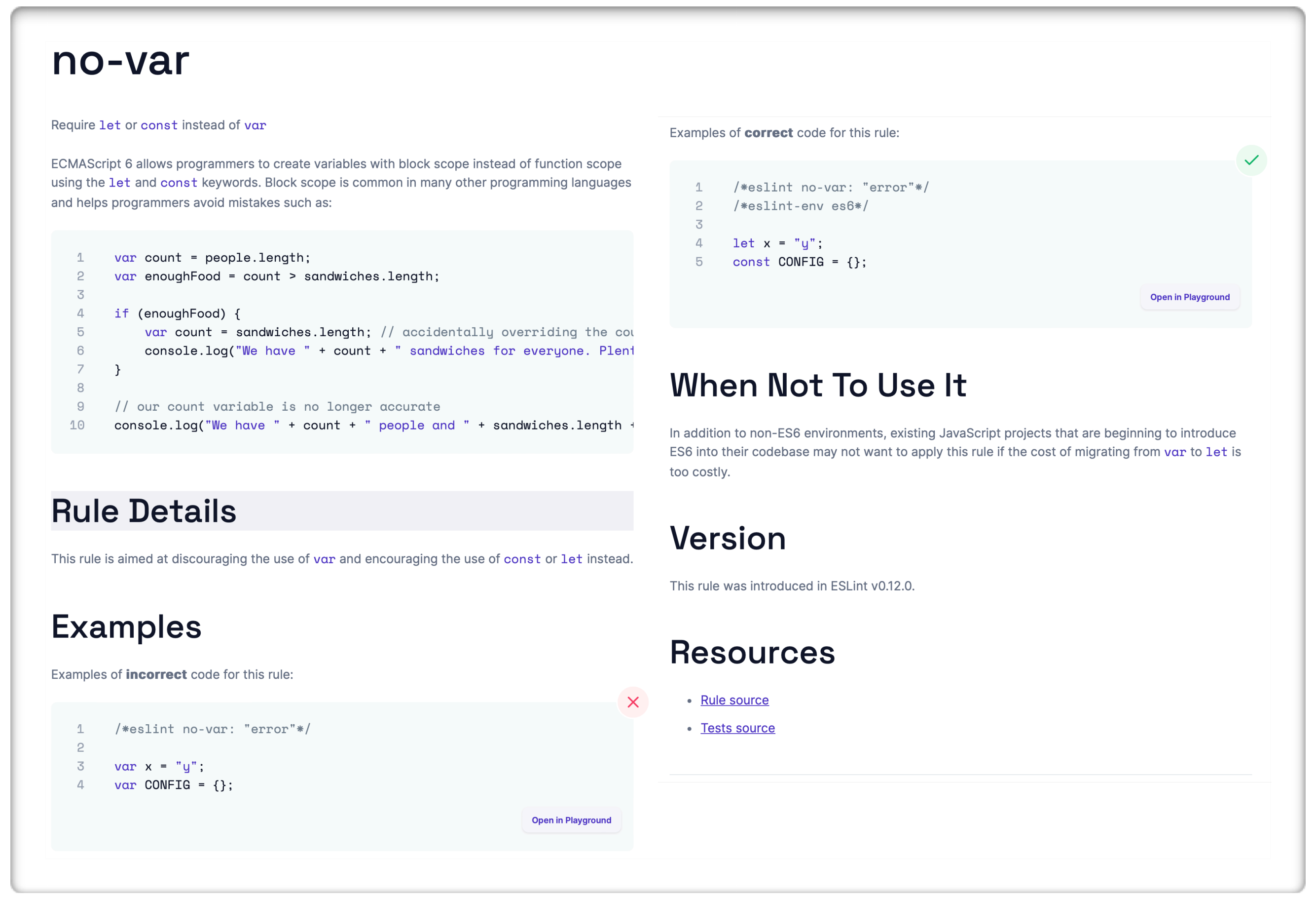
7 languages | 14 tools + 2 multi-language

Pick 1 rule from tool documentation

ESLint | no-var

Extract concepts from rule documentation

Name	Description	Example	Link	…
no-var	X	X		
…				



no-var

Require `let` or `const` instead of `var`

ECMAScript 6 allows programmers to create variables with block scope instead of function scope using the `let` and `const` keywords. Block scope is common in many other programming languages and helps programmers avoid mistakes such as:

```
var count = people.length;
var enoughFood = count > sandwiches.length;

if (enoughFood) {
    var count = sandwiches.length; // accidentally overriding the cou
    console.log("We have " + count + " sandwiches for everyone. Plent
}

// our count variable is no longer accurate
console.log("We have " + count + " people and " + sandwiches.length +
```

Rule Details

This rule is aimed at discouraging the use of `var` and encouraging the use of `const` or `let` instead.

Examples

Examples of **incorrect** code for this rule:

```
/*eslint no-var: "error"*/

var x = "y";
var CONFIG = {};
```

Examples of **correct** code for this rule:

```
/*eslint no-var: "error"*/
/*eslint-env es6*/

let x = "y";
const CONFIG = {};
```

When Not To Use It

In addition to non-ES6 environments, existing JavaScript projects that are beginning to introduce ES6 into their codebase may not want to apply this rule if the cost of migrating from `var` to `let` is too costly.

Version

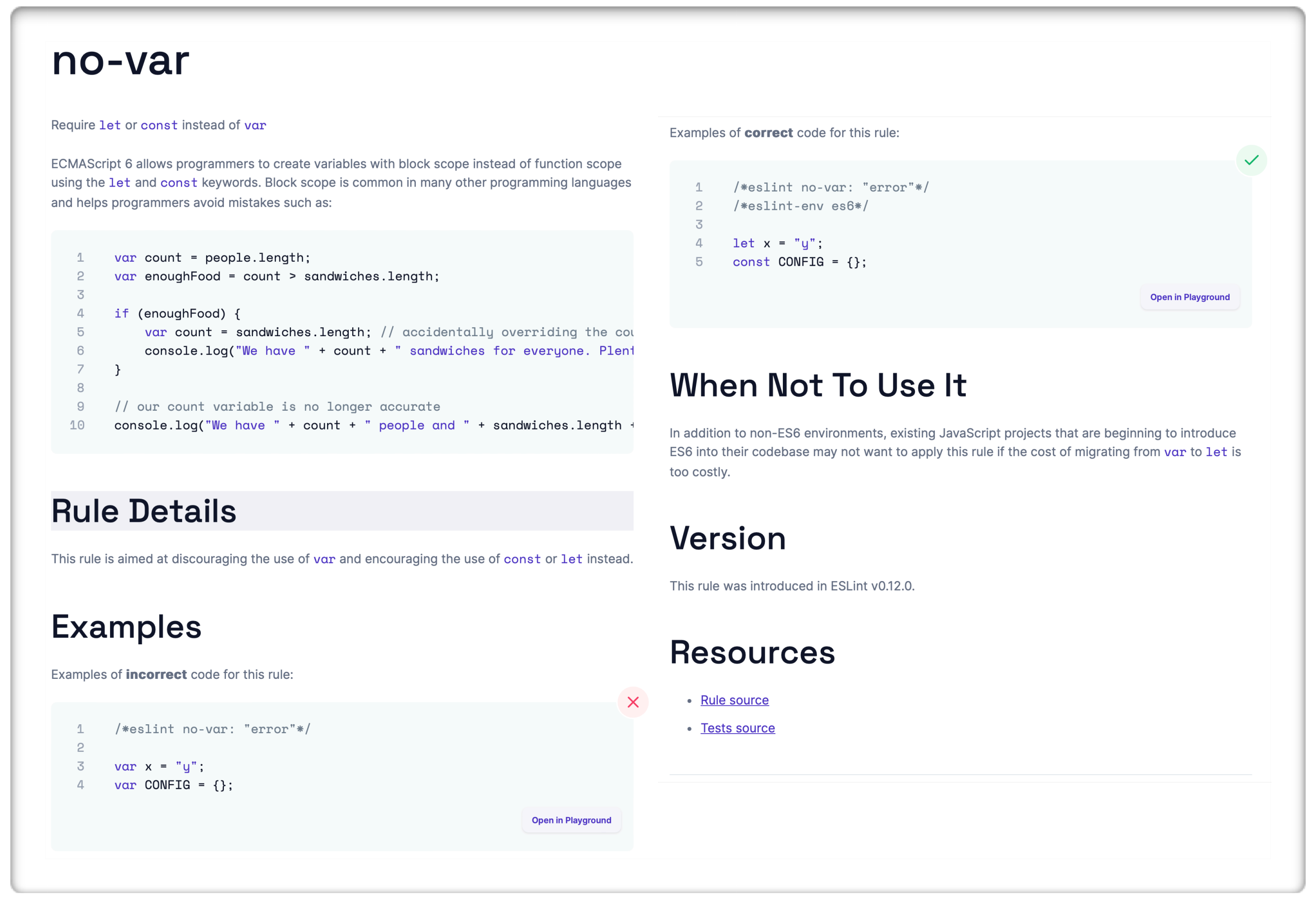
This rule was introduced in ESLint v0.12.0.

Resources

- Rule source
- Tests source

Documentation Analysis

7 languages | 14 tools + 2 multi-language



no-var

Require `let` or `const` instead of `var`

ECMAScript 6 allows programmers to create variables with block scope instead of function scope using the `let` and `const` keywords. Block scope is common in many other programming languages and helps programmers avoid mistakes such as:

```
var count = people.length;
var enoughFood = count > sandwiches.length;

if (enoughFood) {
    var count = sandwiches.length; // accidentally overriding the cou
    console.log("We have " + count + " sandwiches for everyone. Plent
}

// our count variable is no longer accurate
console.log("We have " + count + " people and " + sandwiches.length +
```

Rule Details

This rule is aimed at discouraging the use of `var` and encouraging the use of `const` or `let` instead.

Examples

Examples of **incorrect** code for this rule:

```
/*eslint no-var: "error"*/

var x = "y";
var CONFIG = {};
```

Open in Playground

Examples of **correct** code for this rule:

```
/*eslint no-var: "error"*/
/*eslint-env es6*/

let x = "y";
const CONFIG = {};
```

Open in Playground

When Not To Use It

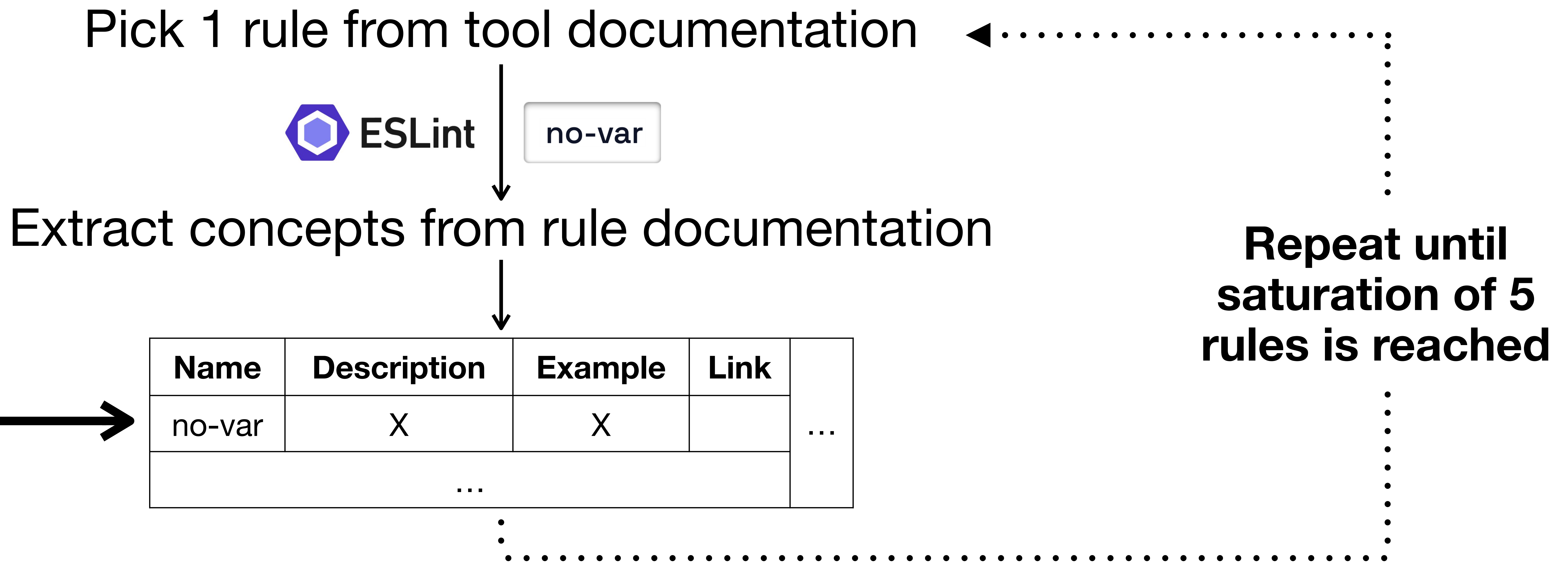
In addition to non-ES6 environments, existing JavaScript projects that are beginning to introduce ES6 into their codebase may not want to apply this rule if the cost of migrating from `var` to `let` is too costly.

Version

This rule was introduced in ESLint v0.12.0.

Resources

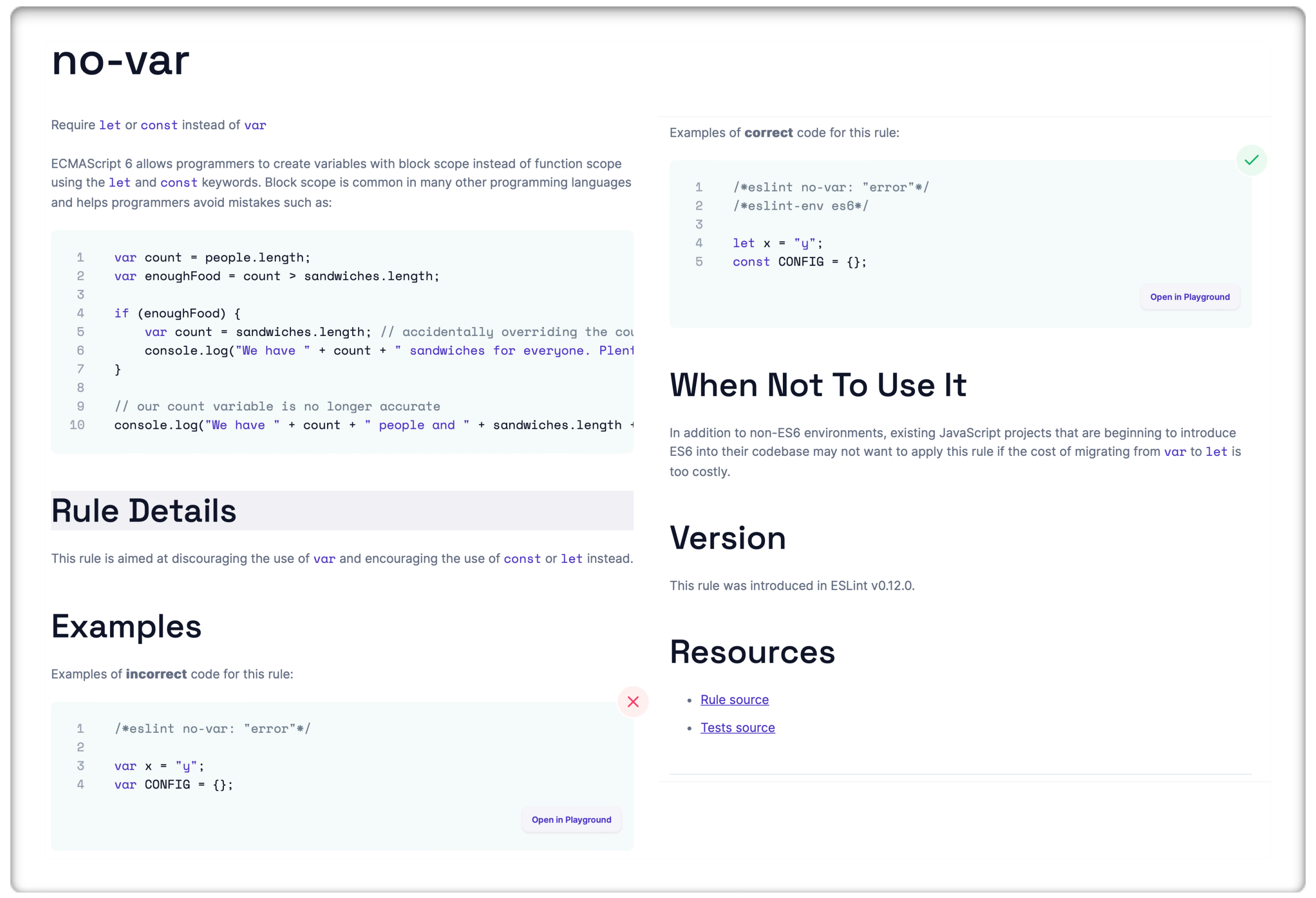
- Rule source
- Tests source


Pick 1 rule from tool documentation
ESLint
no-var
Extract concepts from rule documentation
Repeat until saturation of 5 rules is reached

Name	Description	Example	Link	…
no-var	X	X		
…				

Documentation Analysis

7 languages | 14 tools + 2 multi-language



no-var

Require `let` or `const` instead of `var`

ECMAScript 6 allows programmers to create variables with block scope instead of function scope using the `let` and `const` keywords. Block scope is common in many other programming languages and helps programmers avoid mistakes such as:

```
var count = people.length;
var enoughFood = count > sandwiches.length;

if (enoughFood) {
    var count = sandwiches.length; // accidentally overriding the cou
    console.log("We have " + count + " sandwiches for everyone. Plent
}

// our count variable is no longer accurate
console.log("We have " + count + " people and " + sandwiches.length +
```

Rule Details

This rule is aimed at discouraging the use of `var` and encouraging the use of `const` or `let` instead.

Examples

Examples of **incorrect** code for this rule:

```
/*eslint no-var: "error"*/

var x = "y";
var CONFIG = {};
```

Open in Playground

Examples of **correct** code for this rule:

```
/*eslint no-var: "error"*/
/*eslint-env es6*/

let x = "y";
const CONFIG = {};
```

Open in Playground

When Not To Use It

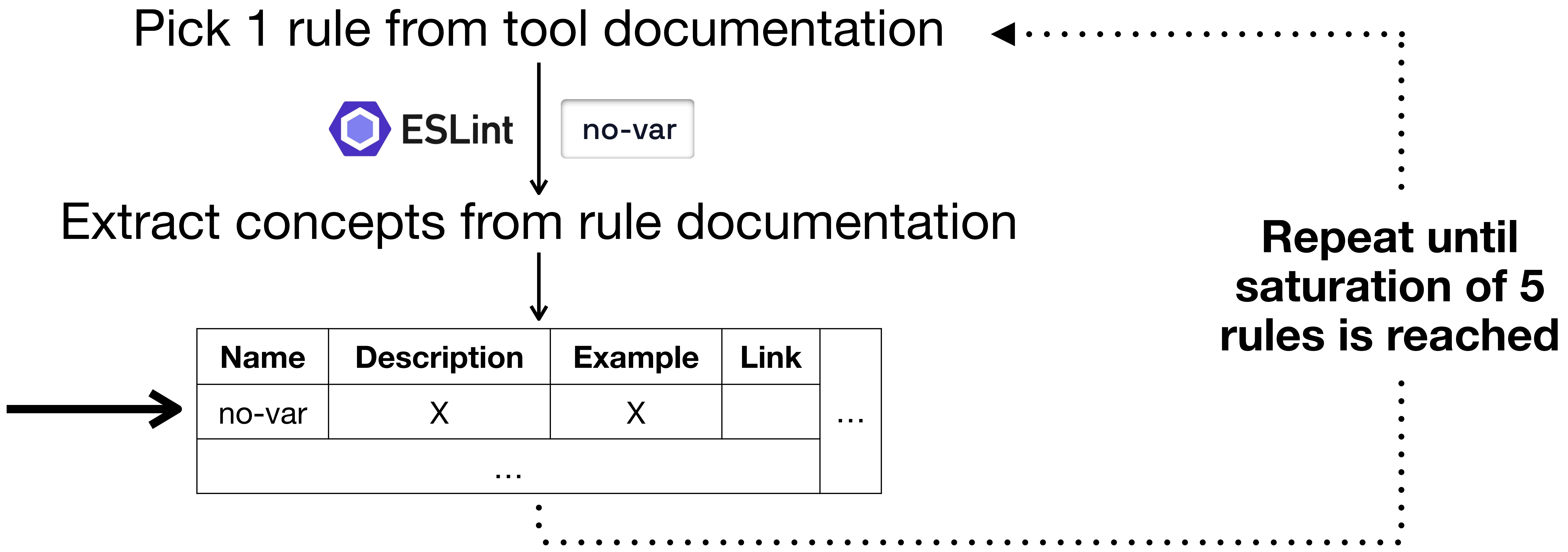
In addition to non-ES6 environments, existing JavaScript projects that are beginning to introduce ES6 into their codebase may not want to apply this rule if the cost of migrating from `var` to `let` is too costly.

Version

This rule was introduced in ESLint v0.12.0.

Resources

- Rule source
- Tests source


Pick 1 rule from tool documentation
ESLint
no-var
Extract concepts from rule documentation
Repeat until saturation of 5 rules is reached

Name	Description	Example	Link	...
no-var	X	X		
...				

119 rules analyzed

State of Rules Documentation

Concepts	**% of appearance**
Description	92
Code Example	87
Severity	34
Further Information	30
Since	29
Rule Definition	25
Configuration	22
Error Output	19
Auto Fix	15
Rule Set	11
Related Rules	8
When Not To Use It	6
Usage Example	5
Compatibility	3
IDE Fix	2

15 concepts

State of Rules Documentation

Themes	Concepts	% of appearance
Comprehension	Description	92
	Code Example	87
	Further Information	30
	When Not To Use It	6
Usage	Since	29
	Configuration	22
	Error Output	19
	Auto Fix	15
	Usage Example	5
	Compatibility	3
	IDE Fix	2
Metadata	Severity	34
	Rule Definition	25
	Rule Set	11
	Related Rules	8

15 concepts

3 themes

State of Rules Documentation

Themes	Concepts	% of appearance
Comprehension	Description	92
	Code Example	87
	Further Information	30
	When Not To Use It	6
Usage	Since	29
	Configuration	22
	Error Output	19
	Auto Fix	15
	Usage Example	5
	Compatibility	3
	IDE Fix	2
Metadata	Severity	34
	Rule Definition	25
	Rule Set	11
	Related Rules	8

15 concepts 3 themes

Explore the comprehension

State of Rules Documentation

Themes	Concepts	% of appearance
Comprehension	Description	92
	Code Example	87
	Further Information	30
	When Not To Use It	6
Usage	Since	29
	Configuration	22
	Error Output	19
	Auto Fix	15
	Usage Example	5
	Compatibility	3
	IDE Fix	2
Metadata	Severity	34
	Rule Definition	25
	Rule Set	11
	Related Rules	8

15 concepts

3 themes

Explore the comprehension

↓

🔨 Taxonomy

🔎 Developers expectations

Taxonomy

Taxonomy

Open card-sorting

Taxonomy

Open card-sorting $\longrightarrow$

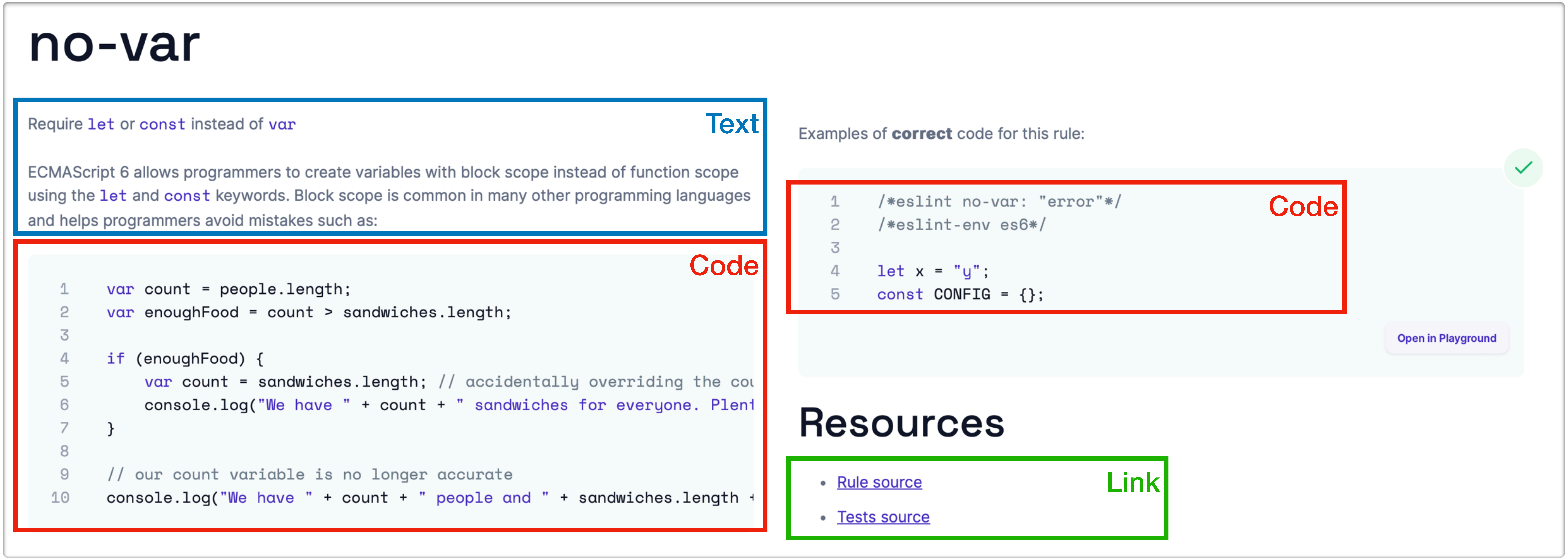
<u>Types of Content</u>

- <u>Text</u>
- <u>Code</u>
- <u>Link</u>

Taxonomy

Open card-sorting → **Types of Content**

- Text
- Code
- Link



no-var

Require `let` or `const` instead of `var`

ECMAScript 6 allows programmers to create variables with block scope instead of function scope using the `let` and `const` keywords. Block scope is common in many other programming languages and helps programmers avoid mistakes such as:

```
var count = people.length;
var enoughFood = count > sandwiches.length;

if (enoughFood) {
    var count = sandwiches.length; // accidentally overriding the cou
    console.log("We have " + count + " sandwiches for everyone. Plent
}

// our count variable is no longer accurate
console.log("We have " + count + " people and " + sandwiches.length +
```

Examples of **correct** code for this rule:

```
/*eslint no-var: "error"*/
/*eslint-env es6*/

let x = "y";
const CONFIG = {};
```

Resources

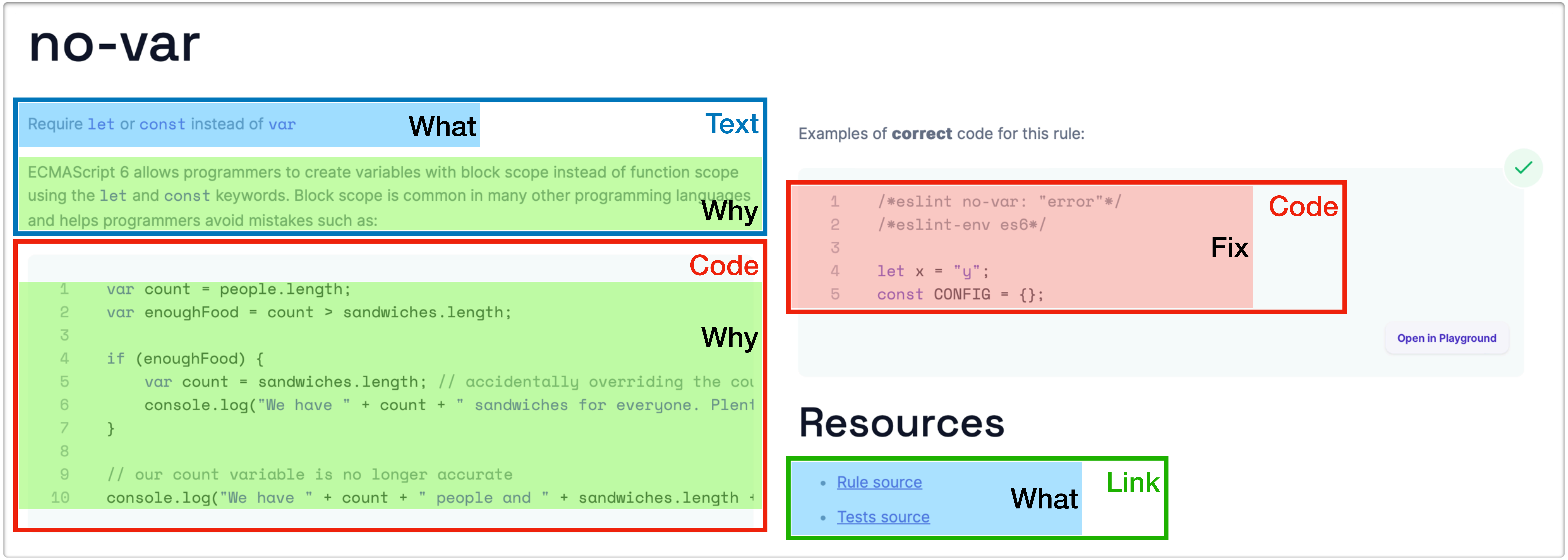
- Rule source
- Tests source

Taxonomy

Open card-sorting →

Types of Content
- Text
- Code
- Link

Purposes
- **What** (100%)
- **Why** (50%)
- **Fix** (77%)



no-var

Require let or const instead of var

ECMAScript 6 allows programmers to create variables with block scope instead of function scope using the let and const keywords. Block scope is common in many other programming languages and helps programmers avoid mistakes such as:

```
var count = people.length;
var enoughFood = count > sandwiches.length;

if (enoughFood) {
    var count = sandwiches.length; // accidentally overriding the cou
    console.log("We have " + count + " sandwiches for everyone. Plent
}

// our count variable is no longer accurate
console.log("We have " + count + " people and " + sandwiches.length +
```

Examples of **correct** code for this rule:

```
/*eslint no-var: "error"*/
/*eslint-env es6*/

let x = "y";
const CONFIG = {};
```

Resources

- Rule source
- Tests source

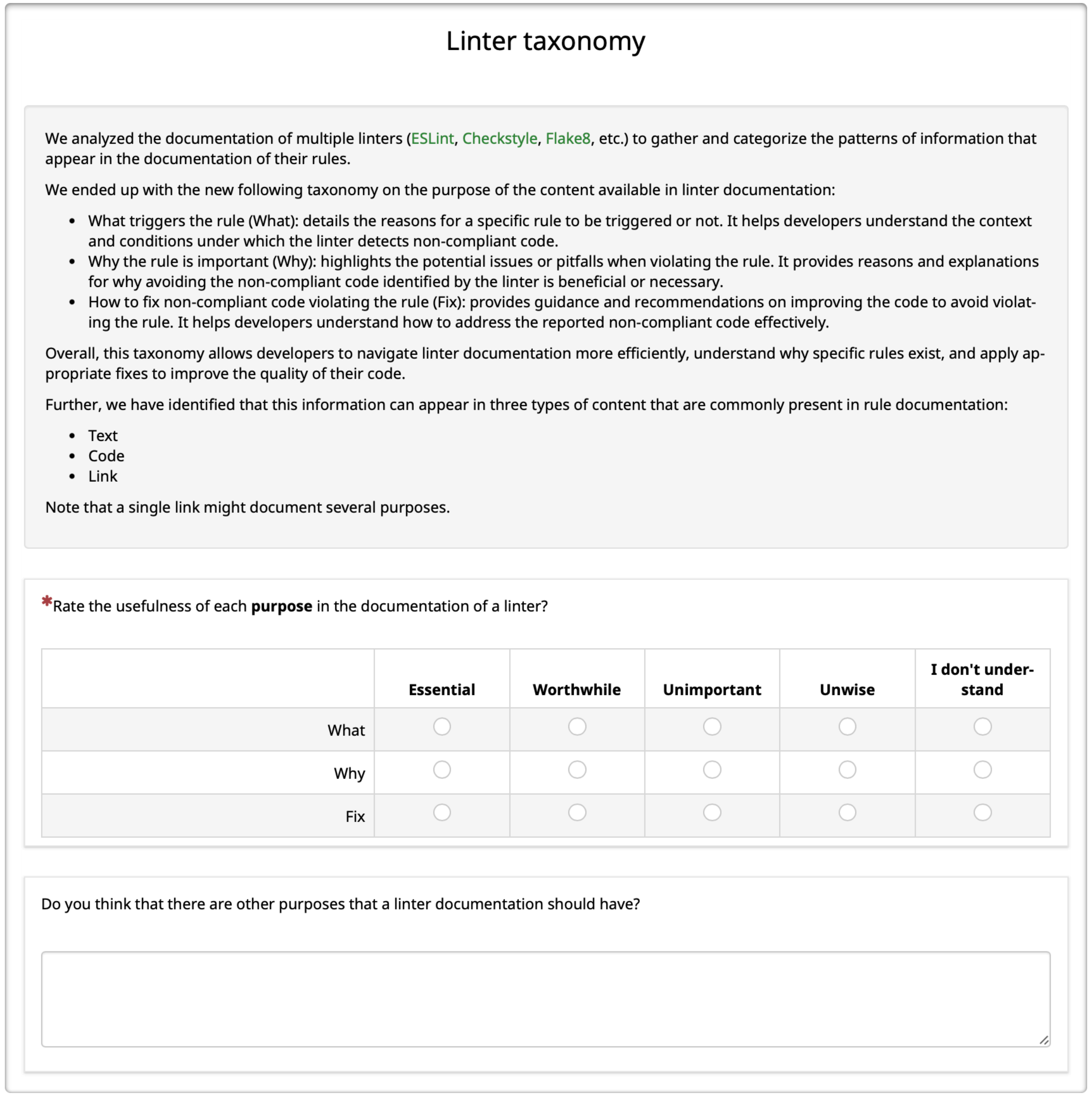
Survey

3 steps:

Survey

3 steps:

- Taxonomy evaluation



Linter taxonomy

We analyzed the documentation of multiple linters (ESLint, Checkstyle, Flake8, etc.) to gather and categorize the patterns of information that appear in the documentation of their rules.

We ended up with the new following taxonomy on the purpose of the content available in linter documentation:

- What triggers the rule (What): details the reasons for a specific rule to be triggered or not. It helps developers understand the context and conditions under which the linter detects non-compliant code.
- Why the rule is important (Why): highlights the potential issues or pitfalls when violating the rule. It provides reasons and explanations for why avoiding the non-compliant code identified by the linter is beneficial or necessary.
- How to fix non-compliant code violating the rule (Fix): provides guidance and recommendations on improving the code to avoid violating the rule. It helps developers understand how to address the reported non-compliant code effectively.

Overall, this taxonomy allows developers to navigate linter documentation more efficiently, understand why specific rules exist, and apply appropriate fixes to improve the quality of their code.

Further, we have identified that this information can appear in three types of content that are commonly present in rule documentation:

- Text
- Code
- Link

Note that a single link might document several purposes.

*Rate the usefulness of each **purpose** in the documentation of a linter?

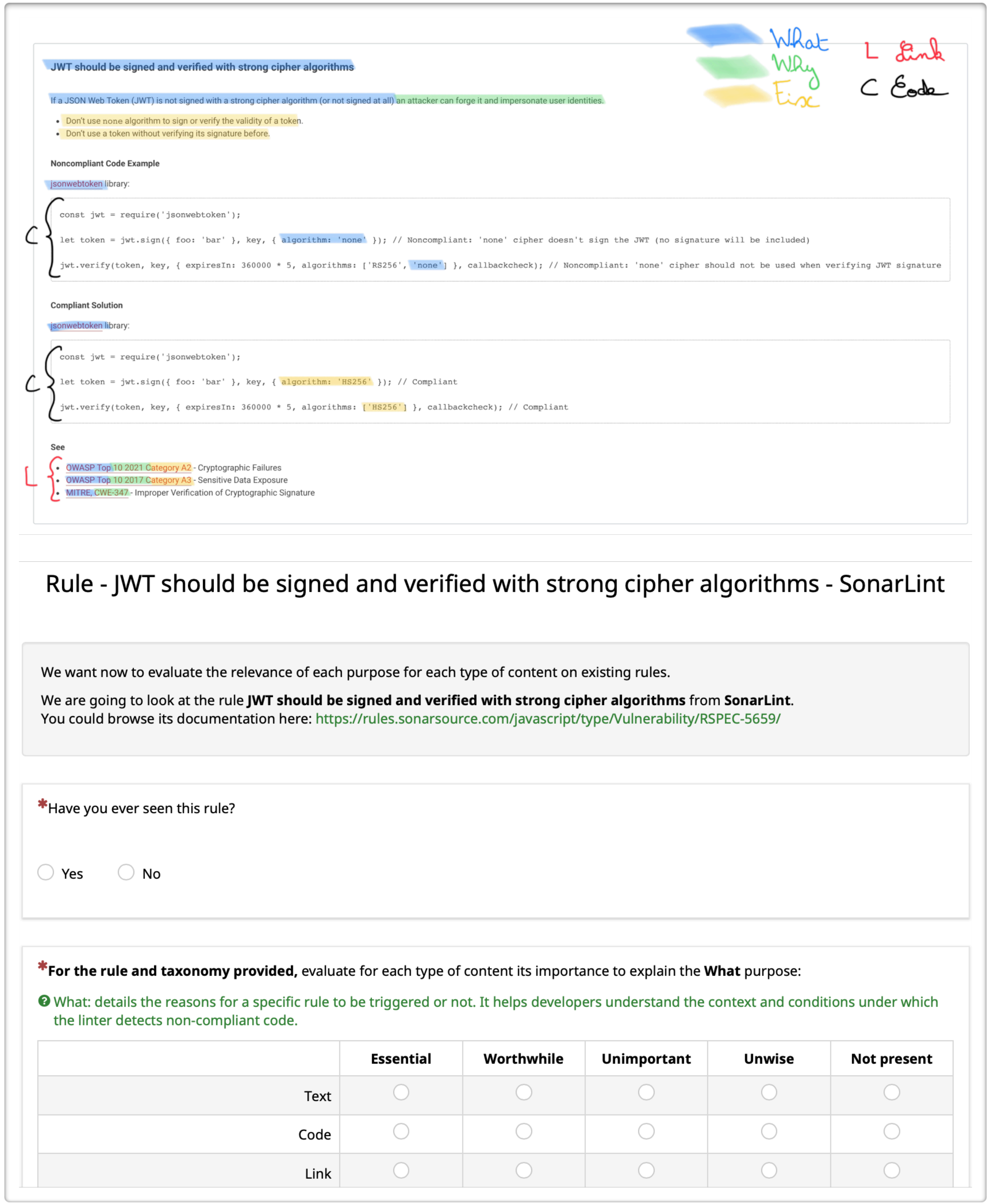
	Essential	Worthwhile	Unimportant	Unwise	I don't understand
What	○	○	○	○	○
Why	○	○	○	○	○
Fix	○	○	○	○	○

Do you think that there are other purposes that a linter documentation should have?

Survey

3 steps:

- Taxonomy evaluation
- Rules analysis



JWT should be signed and verified with strong cipher algorithms

If a JSON Web Token (JWT) is not signed with a strong cipher algorithm (or not signed at all) an attacker can forge it and impersonate user identities.

- Don't use none algorithm to sign or verify the validity of a token.
- Don't use a token without verifying its signature before.

Noncompliant Code Example

jsonwebtoken library:

```
const jwt = require('jsonwebtoken');

let token = jwt.sign({ foo: 'bar' }, key, { algorithm: 'none' }); // Noncompliant: 'none' cipher doesn't sign the JWT (no signature will be included)

jwt.verify(token, key, { expiresIn: 360000 * 5, algorithms: ['RS256', 'none'] }, callbackcheck); // Noncompliant: 'none' cipher should not be used when verifying JWT signature
```

Compliant Solution

jsonwebtoken library:

```
const jwt = require('jsonwebtoken');

let token = jwt.sign({ foo: 'bar' }, key, { algorithm: 'HS256' }); // Compliant

jwt.verify(token, key, { expiresIn: 360000 * 5, algorithms: ['HS256'] }, callbackcheck); // Compliant
```

See

- OWASP Top 10 2021 Category A2 - Cryptographic Failures
- OWASP Top 10 2017 Category A3 - Sensitive Data Exposure
- MITRE, CWE-347 - Improper Verification of Cryptographic Signature

Rule - JWT should be signed and verified with strong cipher algorithms - SonarLint

We want now to evaluate the relevance of each purpose for each type of content on existing rules.

We are going to look at the rule **JWT should be signed and verified with strong cipher algorithms** from **SonarLint**.
You could browse its documentation here: https://rules.sonarsource.com/javascript/type/Vulnerability/RSPEC-5659/

*Have you ever seen this rule?

○ Yes ○ No

***For the rule and taxonomy provided,** evaluate for each type of content its importance to explain the **What** purpose:

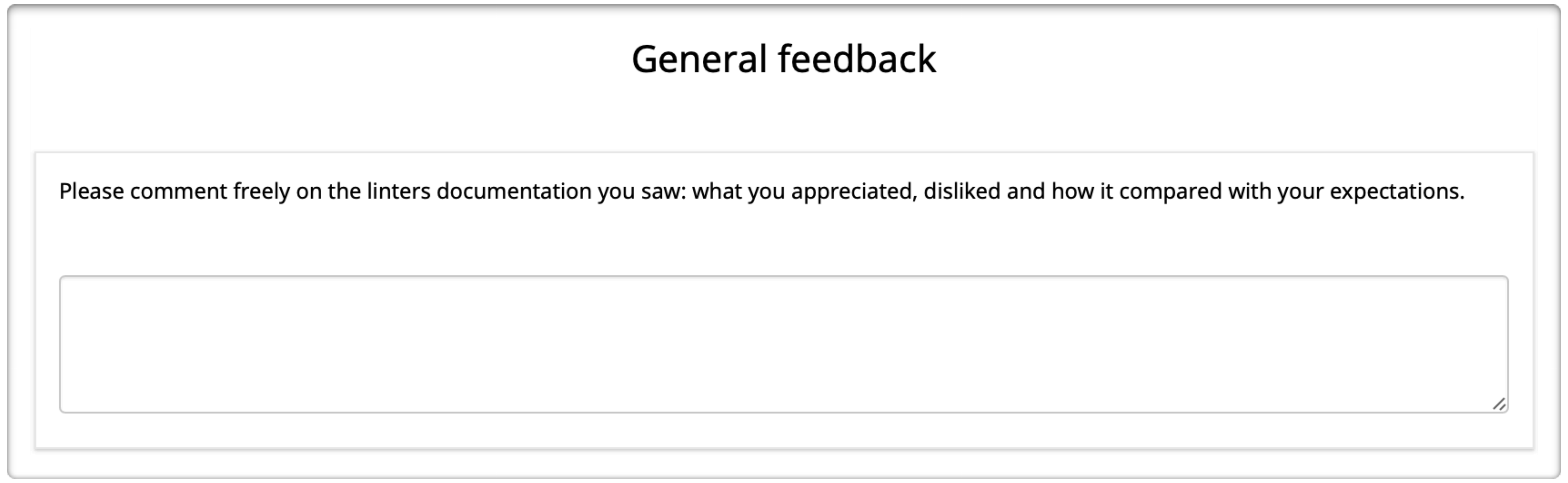
What: details the reasons for a specific rule to be triggered or not. It helps developers understand the context and conditions under which the linter detects non-compliant code.

	Essential	Worthwhile	Unimportant	Unwise	Not present
Text	○	○	○	○	○
Code	○	○	○	○	○
Link	○	○	○	○	○

Survey

3 steps:

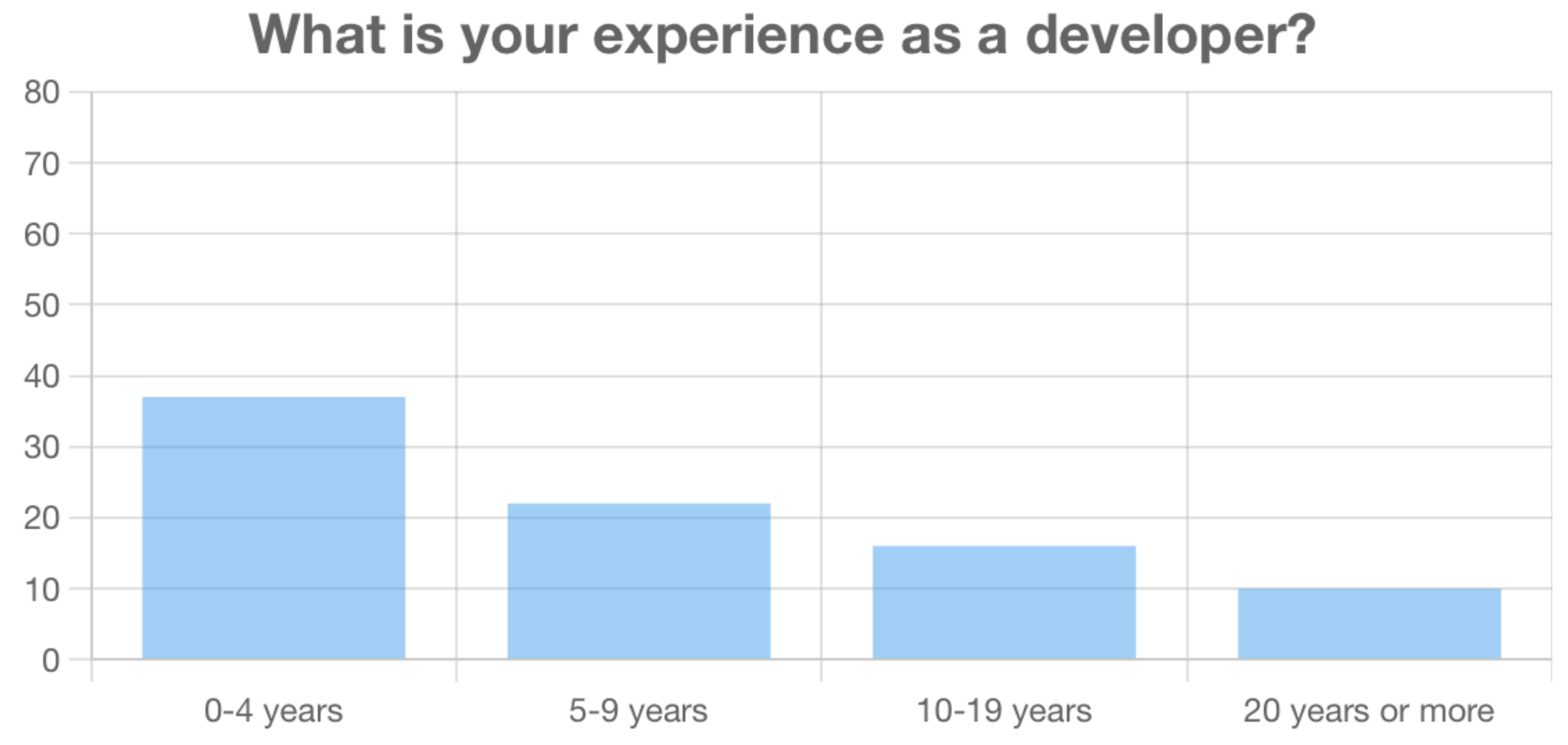
- Taxonomy evaluation
- Rules analysis
- General feedback

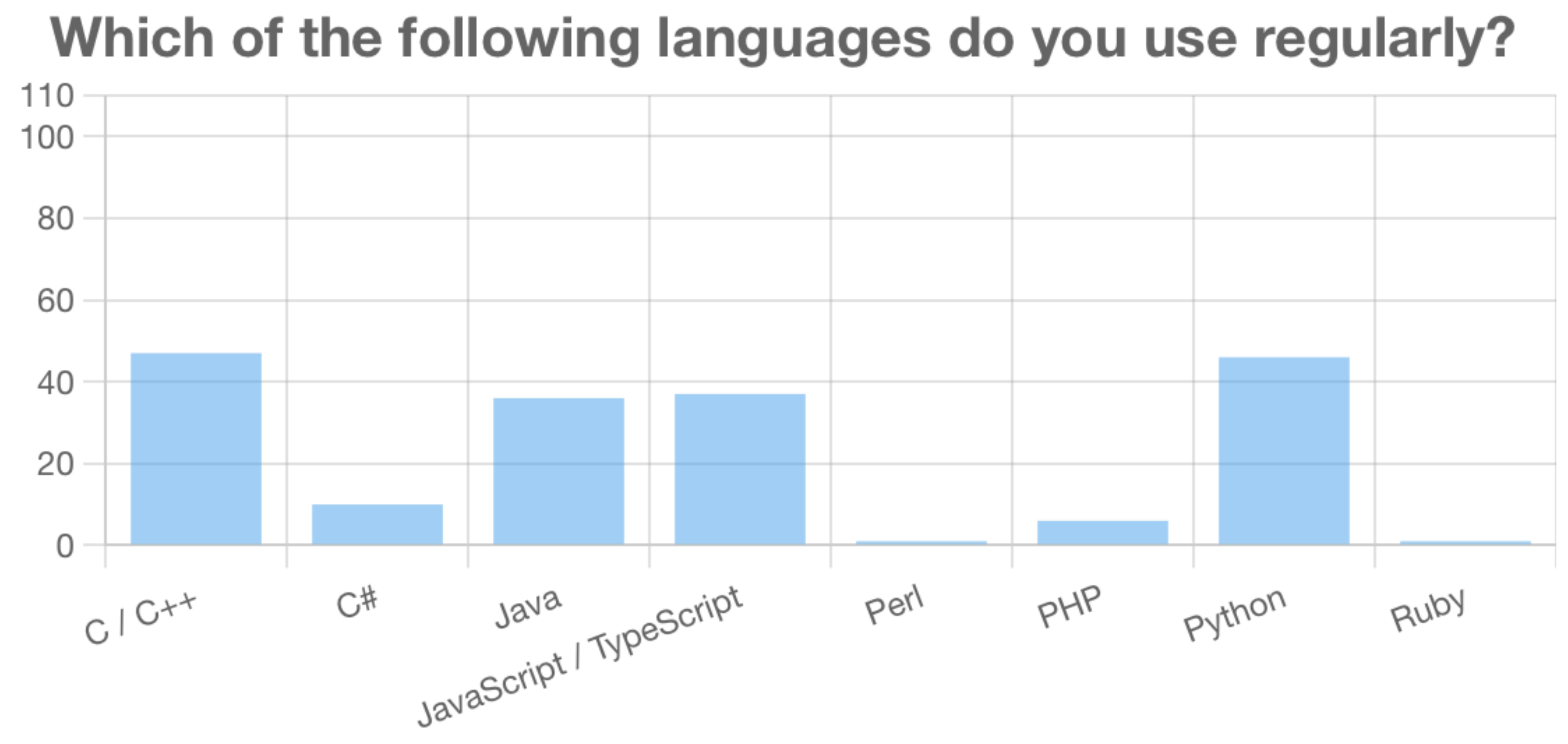


General feedback

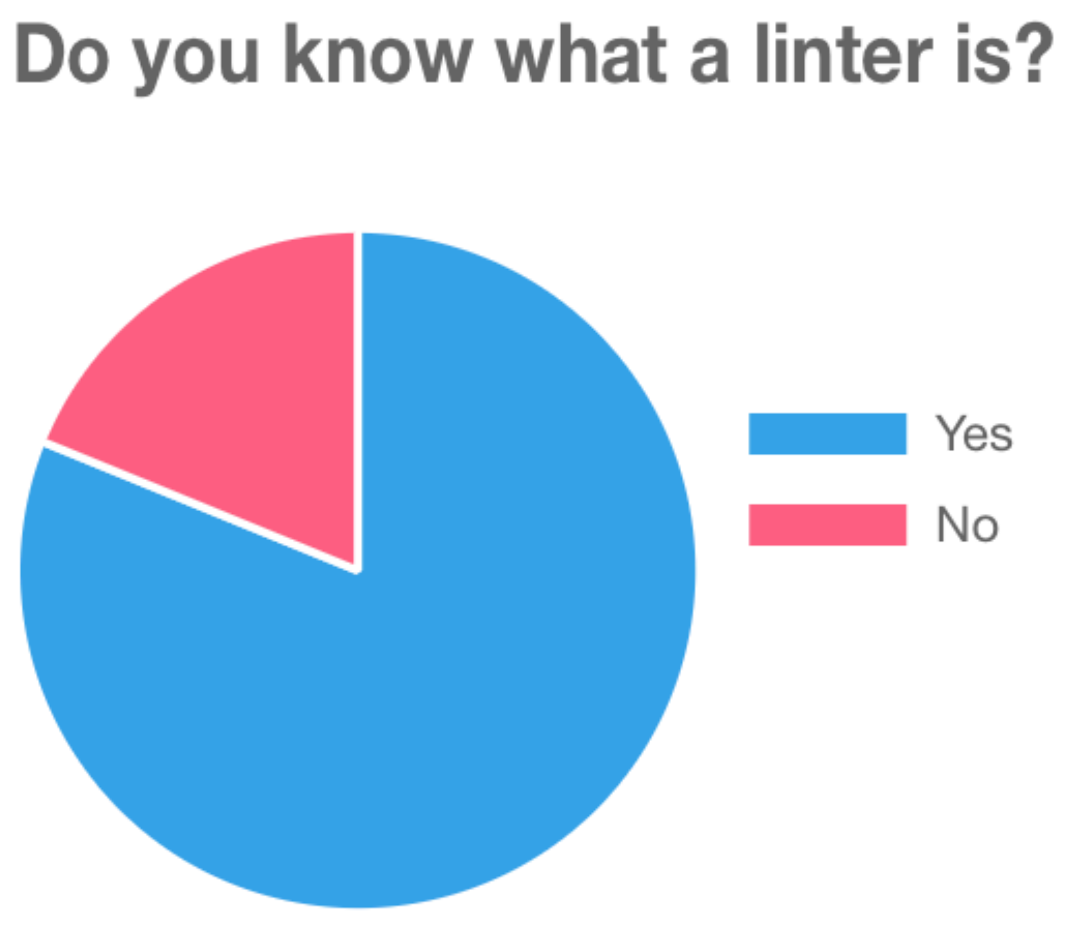
Please comment freely on the linters documentation you saw: what you appreciated, disliked and how it compared with your expectations.

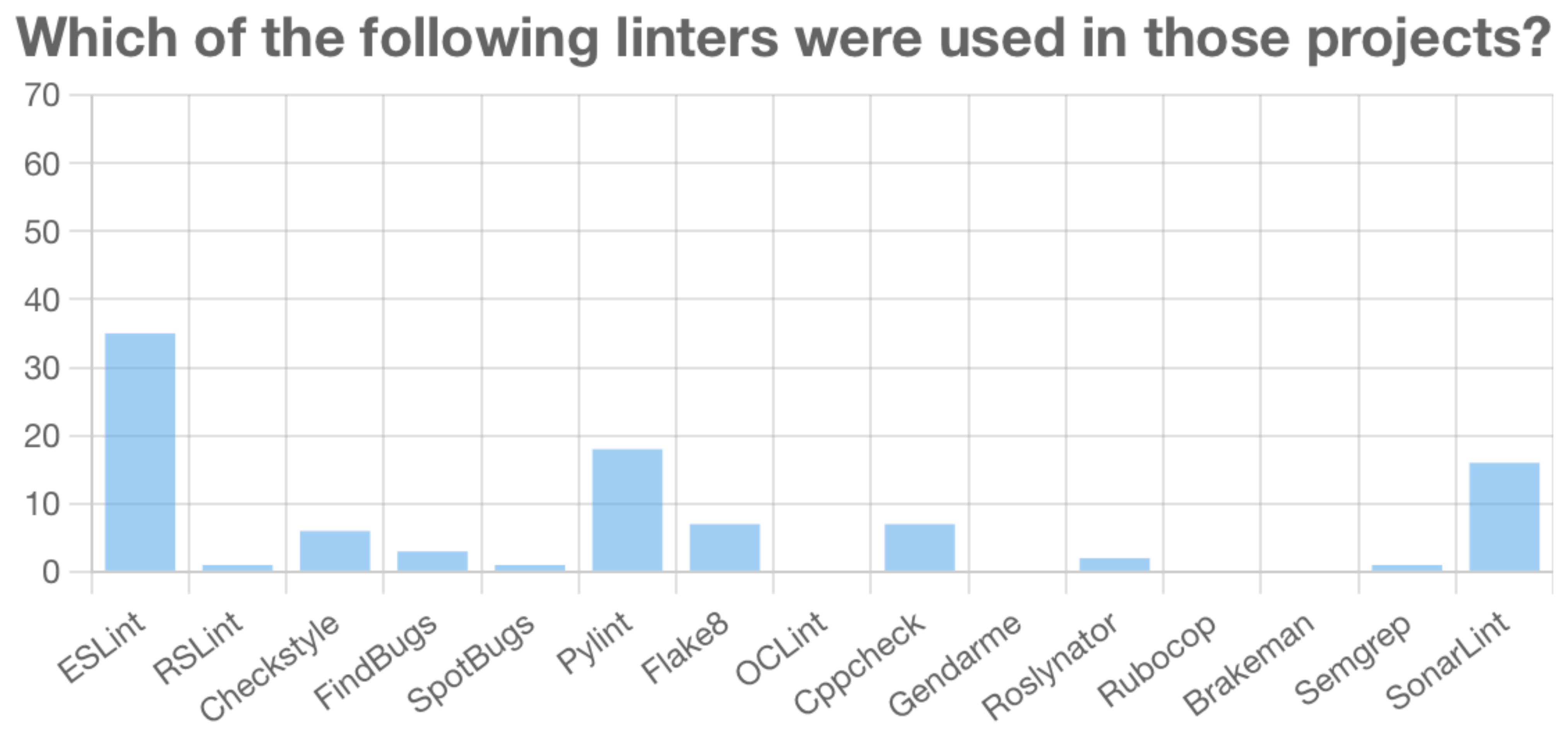
Survey Participants



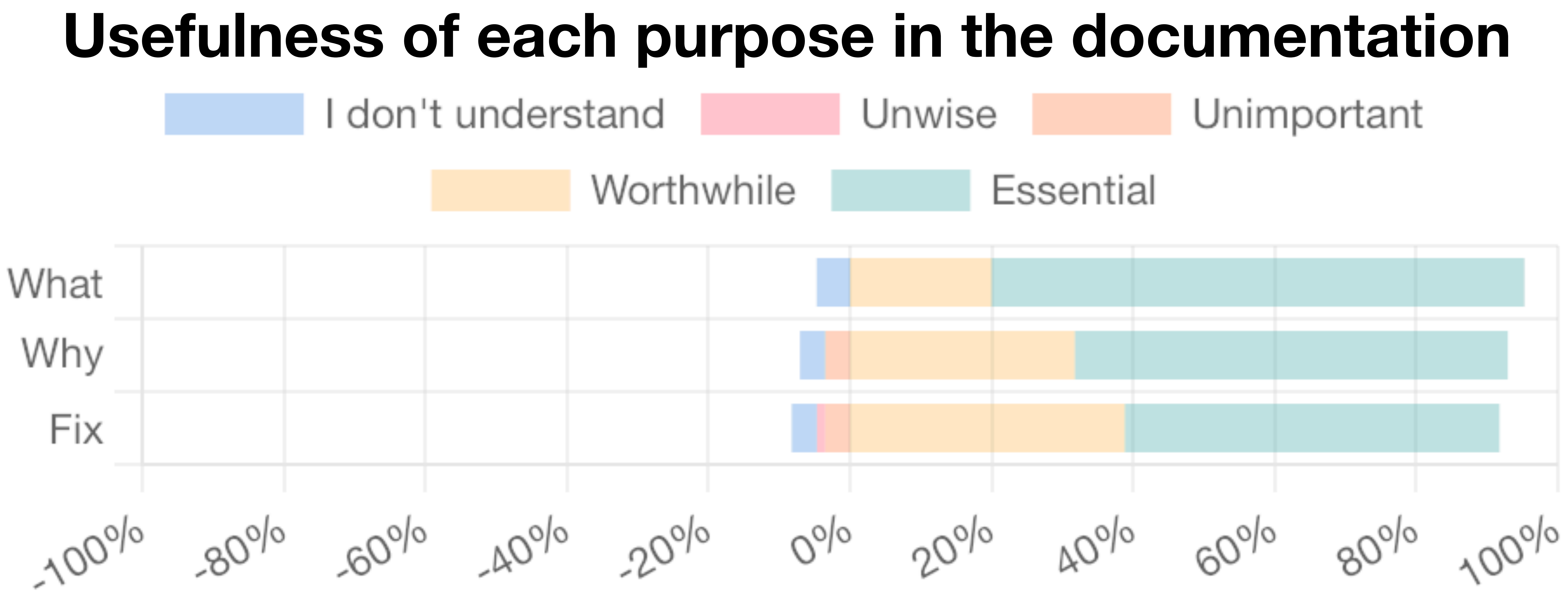


85 Participants

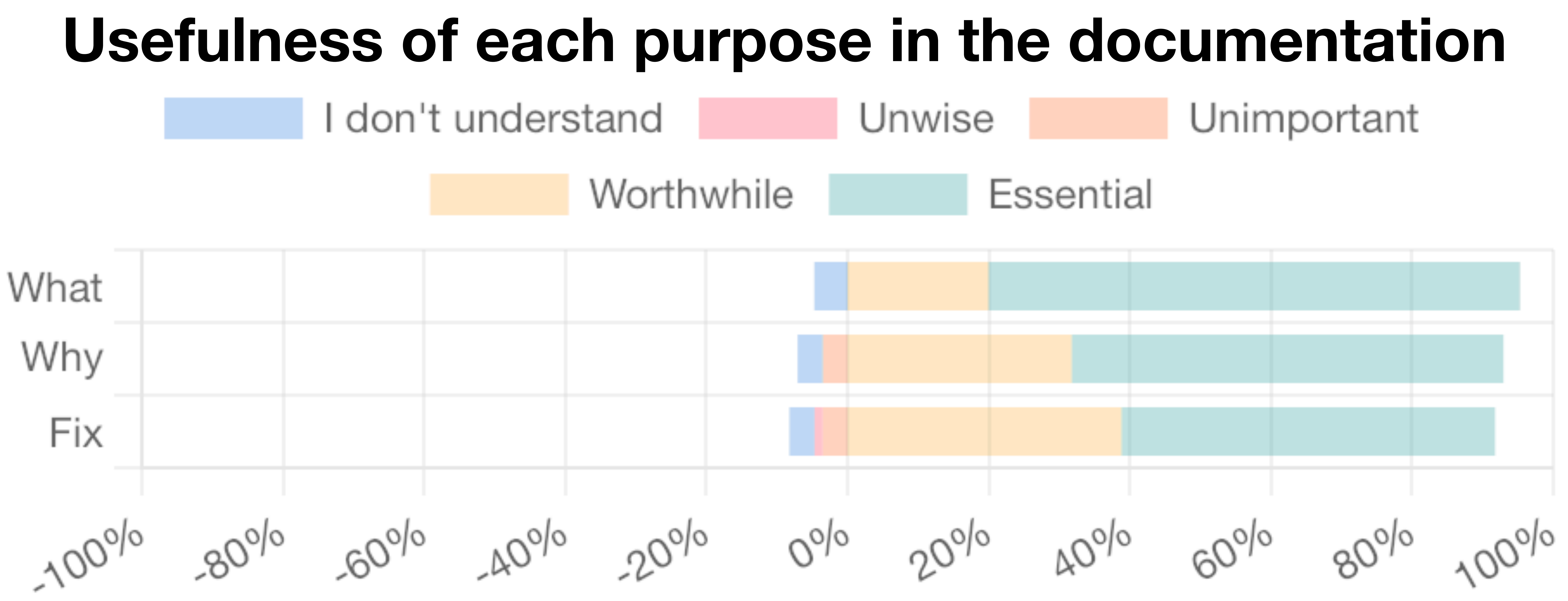




Quantitative Results



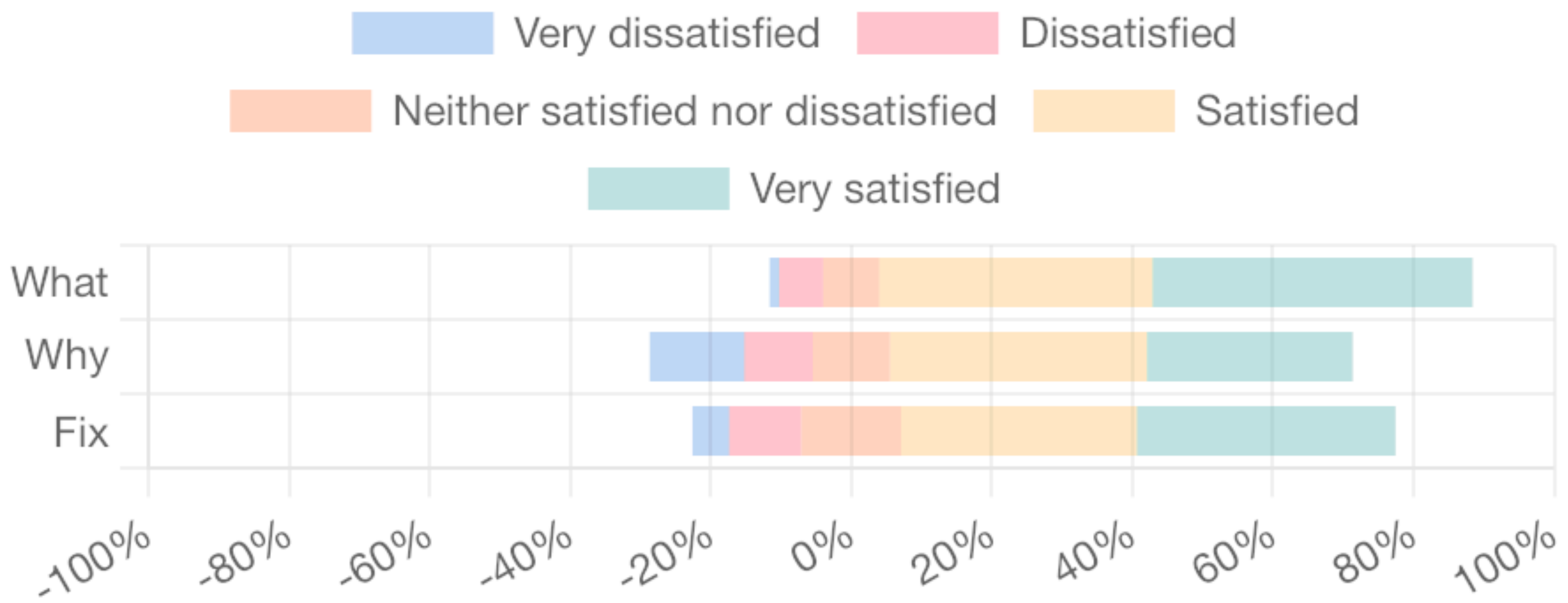
Quantitative Results



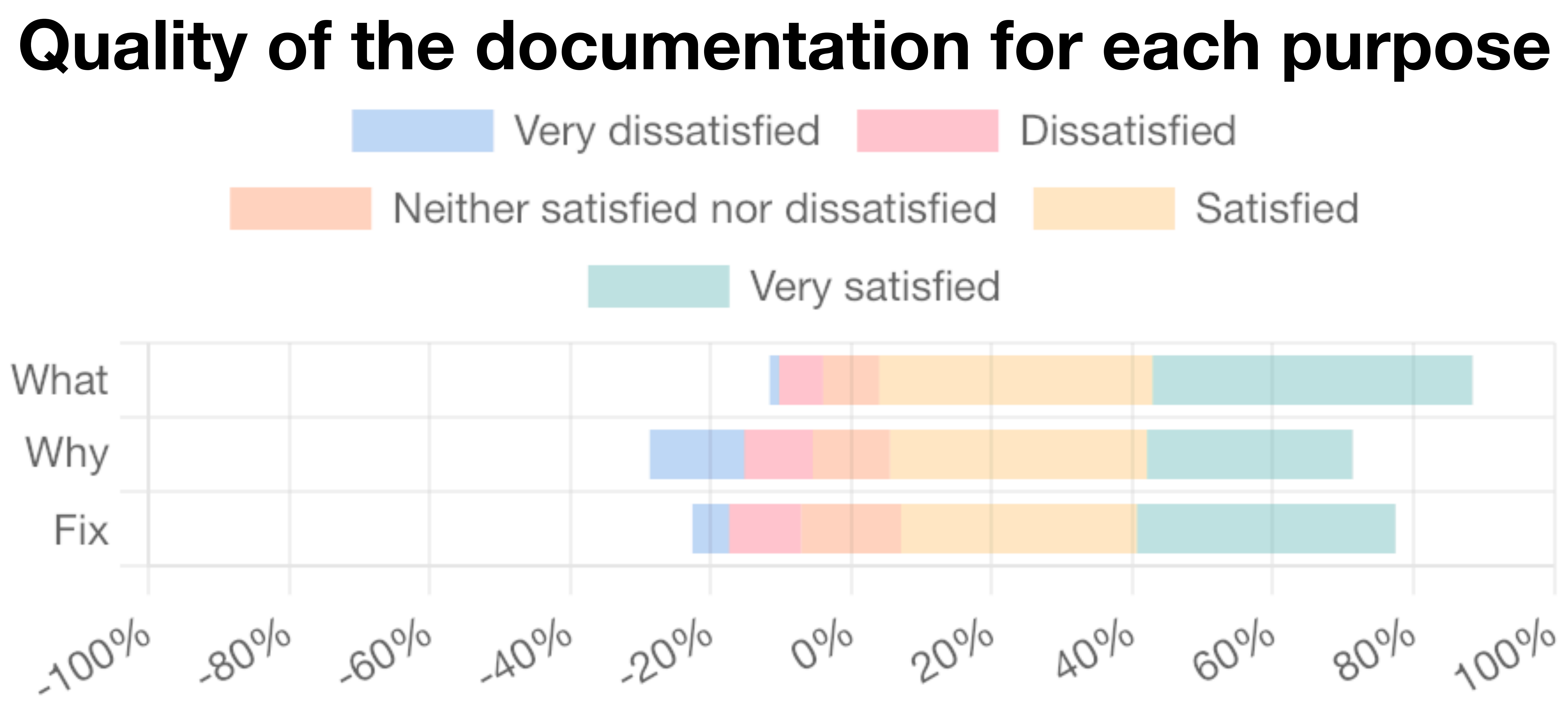
Each purpose has to be documented

Quantitative Results

Quality of the documentation for each purpose



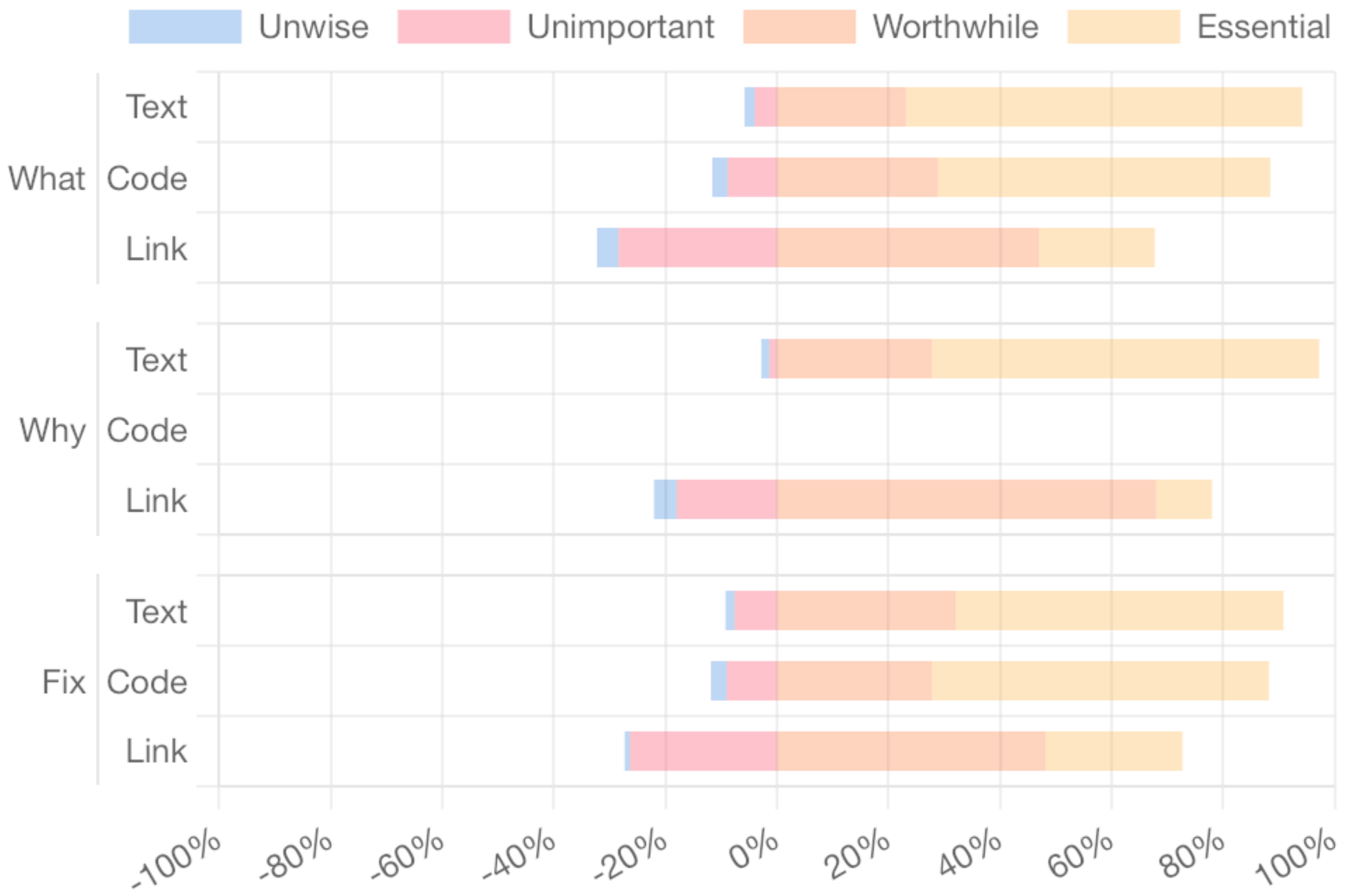
Quantitative Results



Why needs to be improved

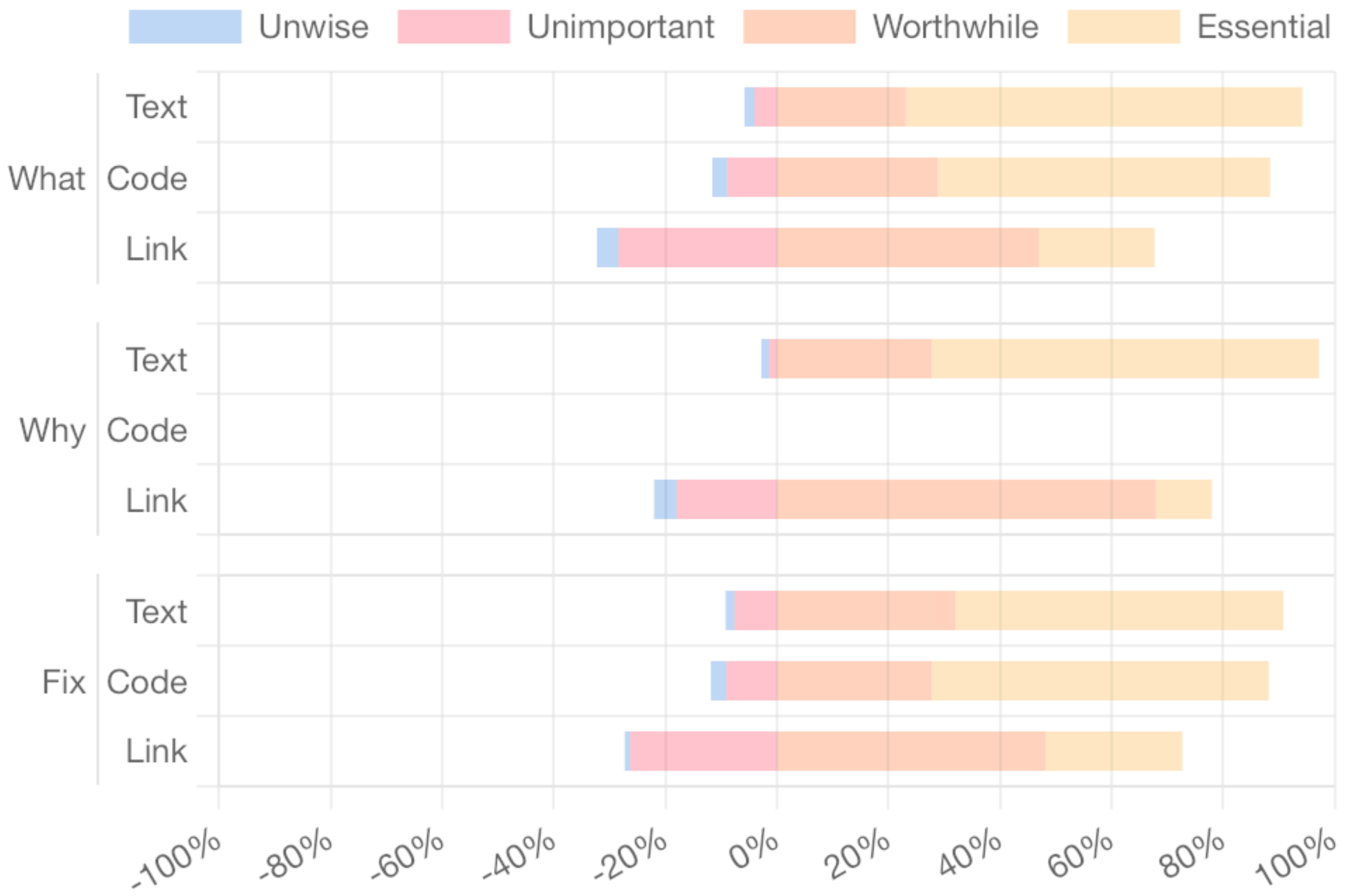
Quantitative Results

Importance of content types to document purposes



Quantitative Results

Importance of content types to document purposes

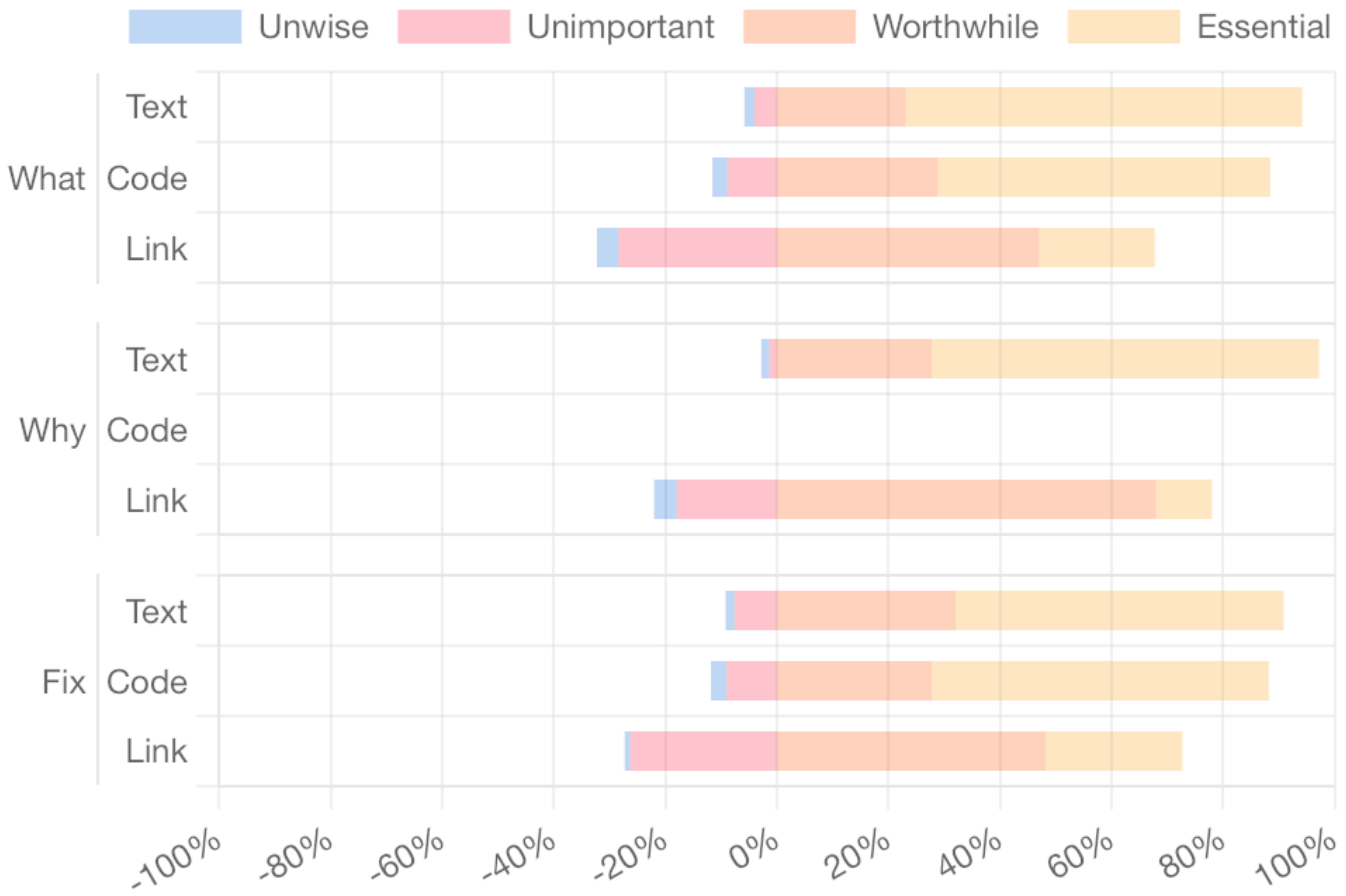


To document

- **What**: Text & Code
- **Why**: Text
- **Fix**: Text & Code

Quantitative Results

Importance of content types to document purposes



To document

- **What**: Text & Code
- **Why**: Text
- **Fix**: Text & Code

Use Link sparingly

Qualitative Results

Qualitative Results

Learning vs Saving time

Qualitative Results

Learning vs Saving time

Use of a structured template, with

- summary
- code examples

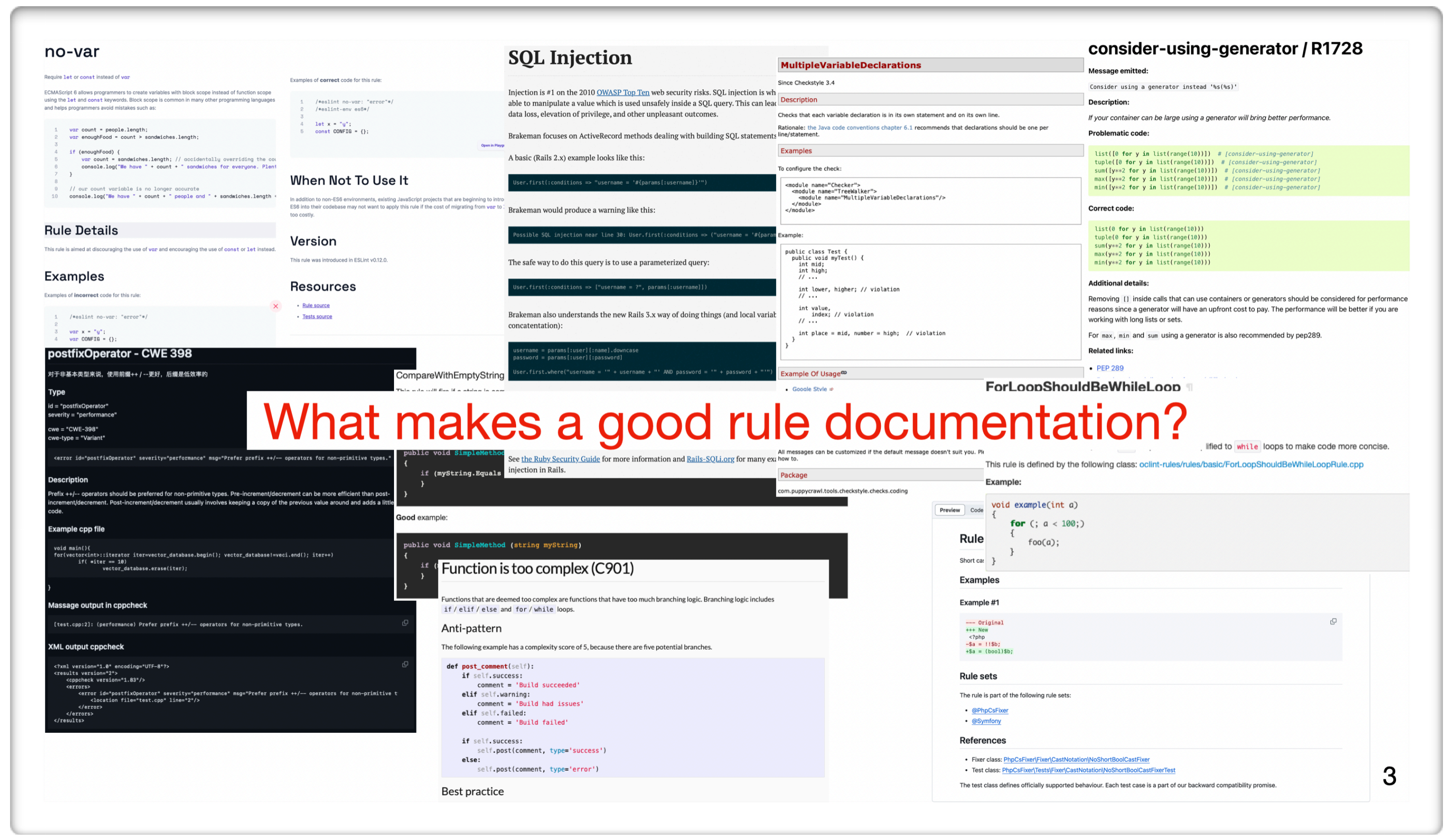
Qualitative Results

Learning vs Saving time

Use of a structured template, with

- summary
- code examples

Use external links sparingly



no-var

SQL Injection

consider-using-generator / R1728

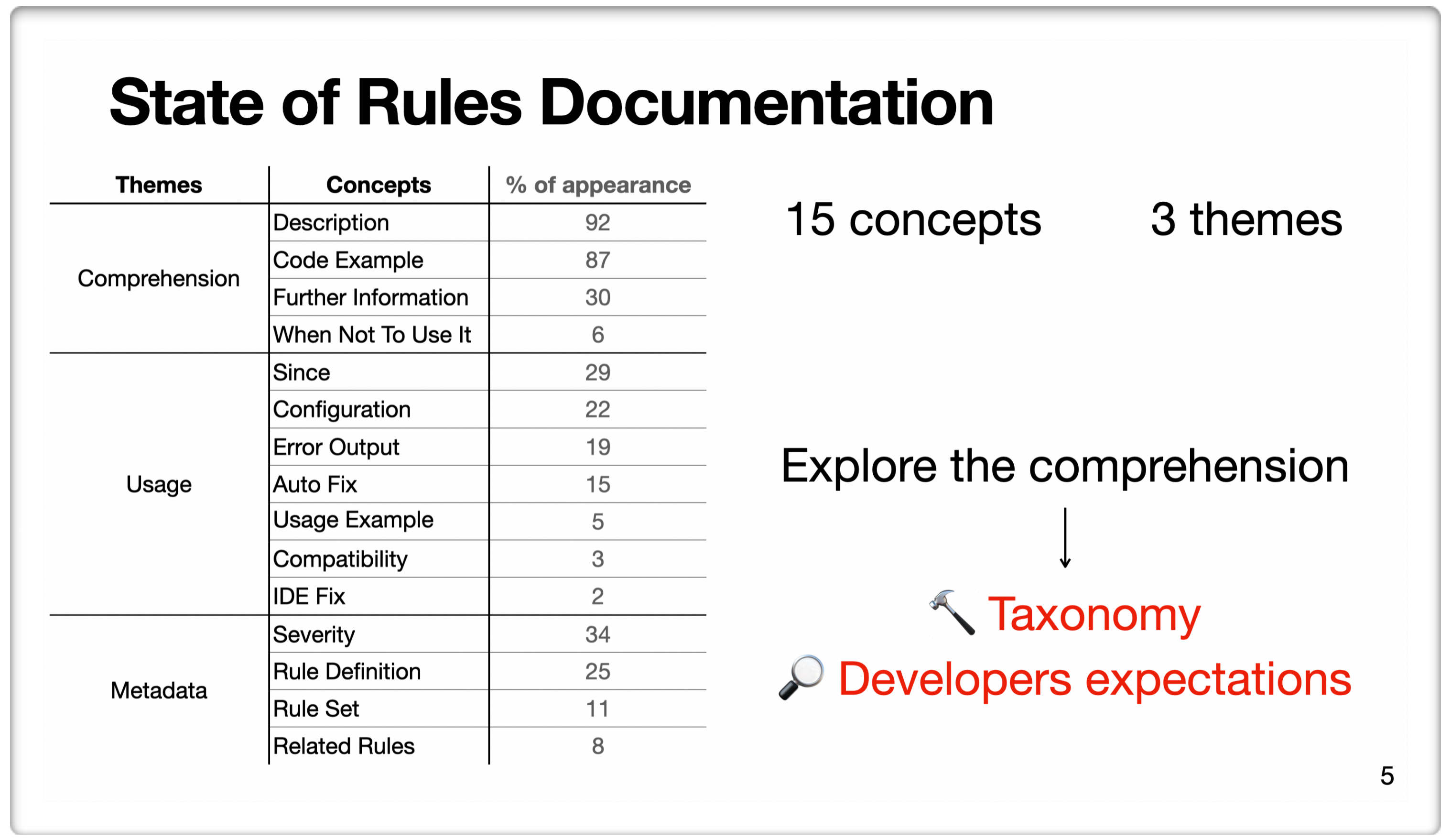
postfixOperator - CWE 398

ForLoopShouldBeWhileLoop

Function is too complex (C901)

What makes a good rule documentation?

3



State of Rules Documentation

Themes	Concepts	% of appearance
Comprehension	Description	92
	Code Example	87
	Further Information	30
	When Not To Use It	6
Usage	Since	29
	Configuration	22
	Error Output	19
	Auto Fix	15
	Usage Example	5
	Compatibility	3
	IDE Fix	2
Metadata	Severity	34
	Rule Definition	25
	Rule Set	11
	Related Rules	8

15 concepts 3 themes

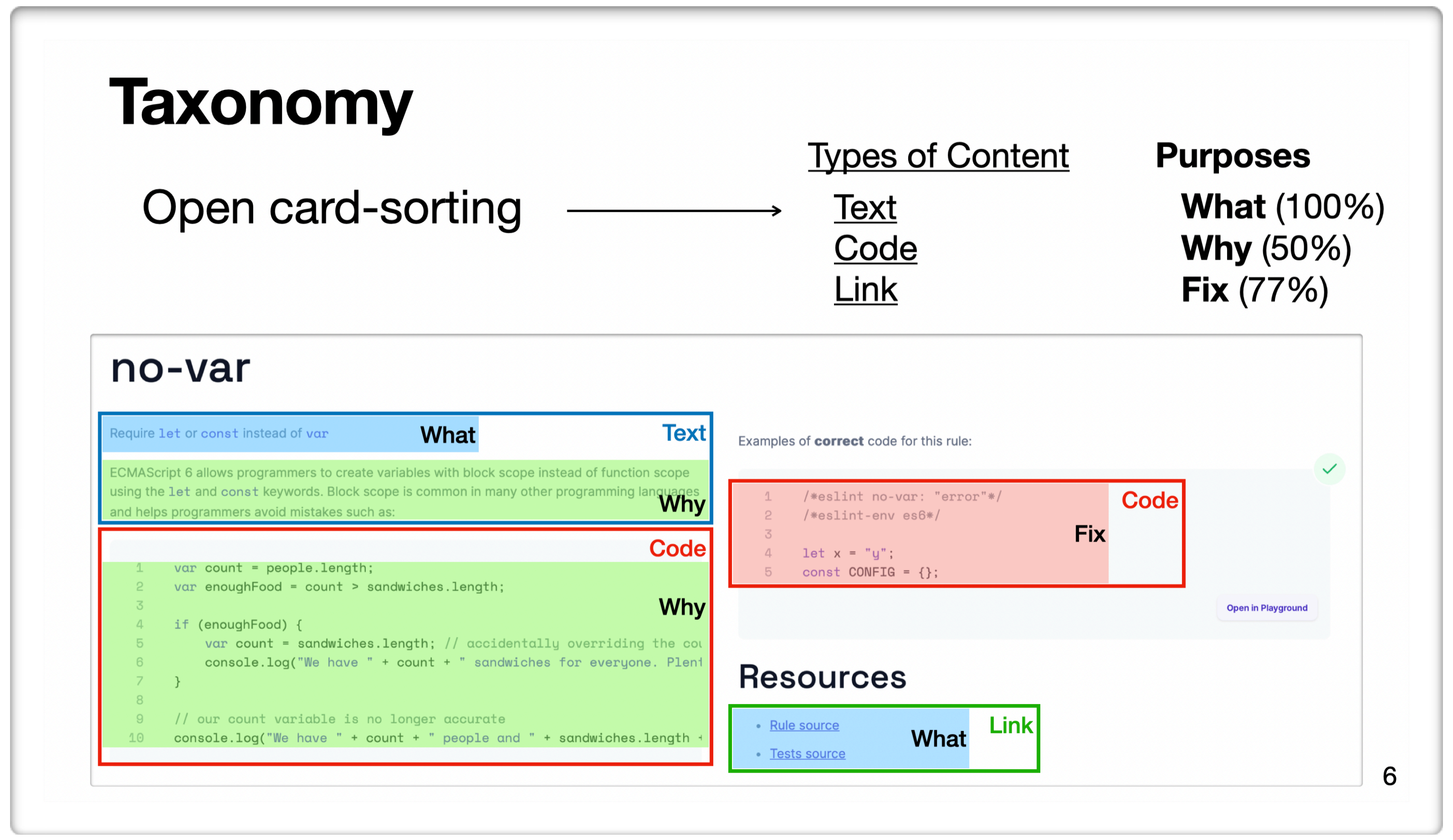
Explore the comprehension

↓

🔨 Taxonomy

🔎 Developers expectations

5



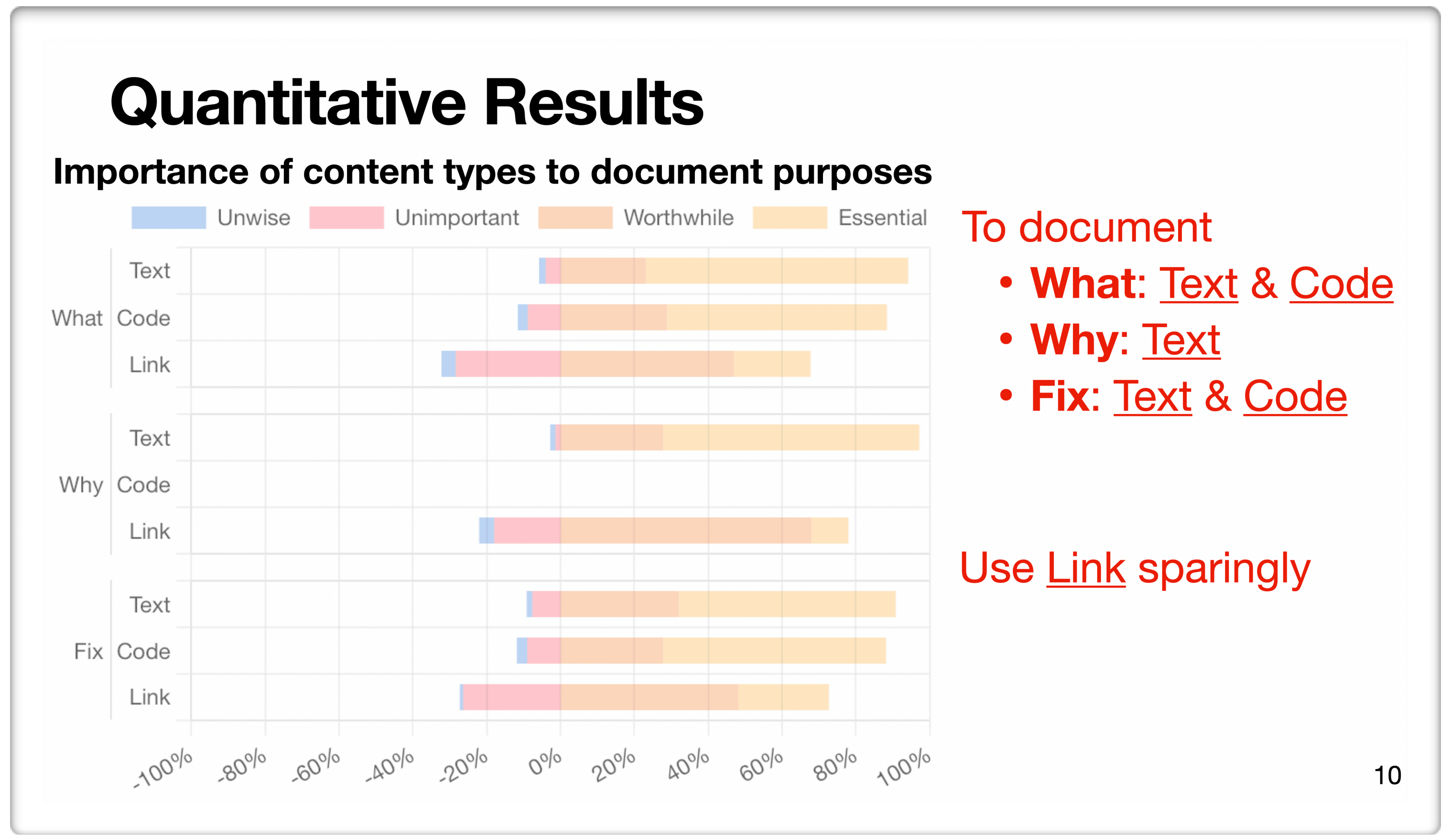
Taxonomy

Open card-sorting ⟶

Types of Content	Purposes
Text	**What** (100%)
Code	**Why** (50%)
Link	**Fix** (77%)

6

What the Fix? A Study of ASATs Rule Documentation



Quantitative Results

Importance of content types to document purposes

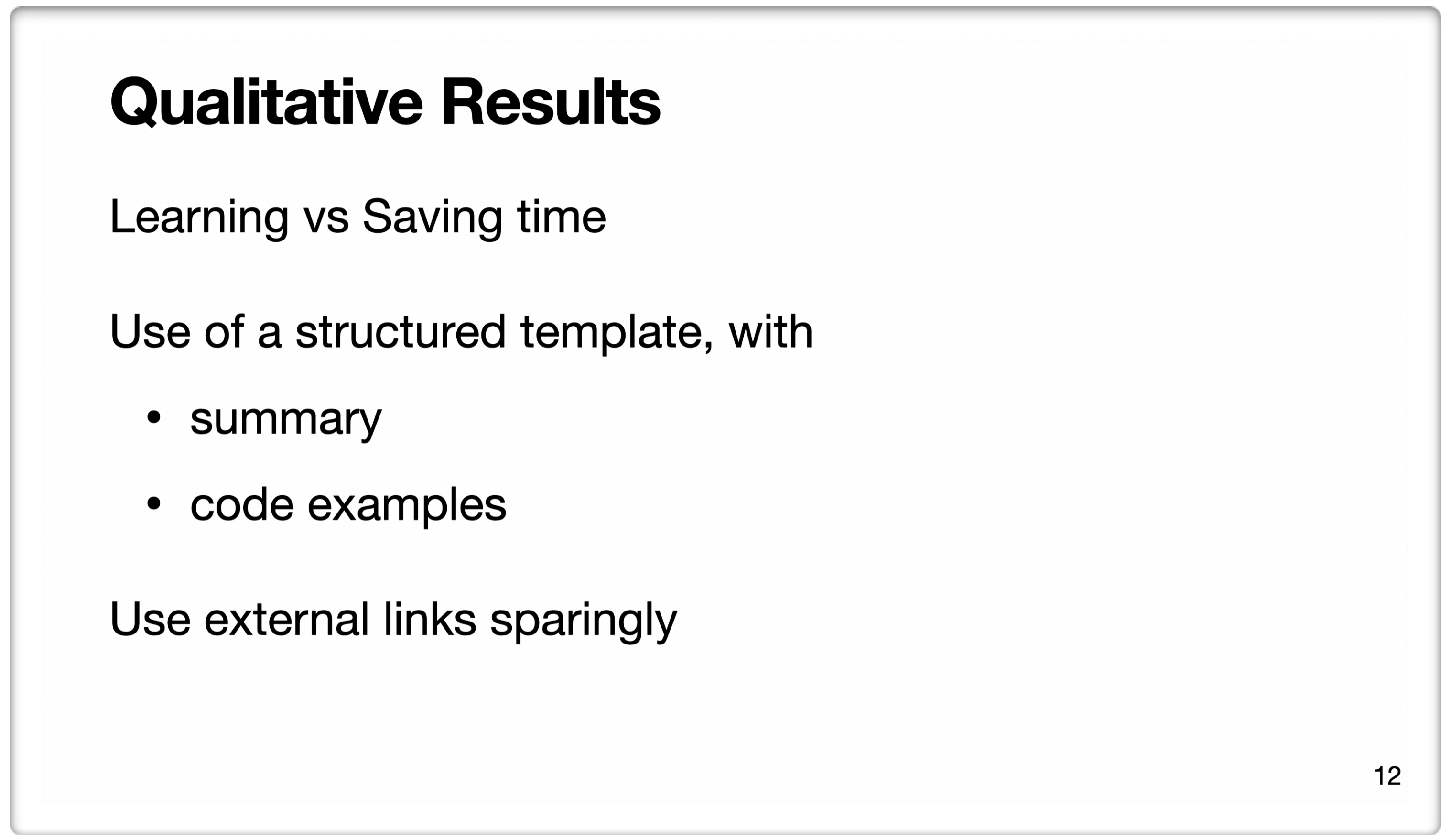
To document

- **What:** Text & Code
- **Why:** Text
- **Fix:** Text & Code

Use Link sparingly

10

https://icpc2024-asats.github.io/



Qualitative Results

Learning vs Saving time

Use of a structured template, with

- summary
- code examples

Use external links sparingly

12

Sophia de Mello Breyner Andresen Room

Corentin LATAPPY ✉ **corentin.latappy@labri.fr**

Tuesday, April 16, 2024

[⁄⁄] packmind LaBRI