SARIF support for code scanning

In this article

To display results from a third-party static analysis tool in your repository on GitHub, you'll need your results stored in a SARIF file that supports a specific subset of the SARIF 2.1.0 JSON schema for code scanning. If you use the default CodeQL static analysis engine, then your results will display in your repository on GitHub automatically.

Code scanning is available for all public repositories on GitHub.com. Code scanning is also available for private repositories owned by organizations that use GitHub Enterprise Cloud and have a license for GitHub Advanced Security. For more information, see "About GitHub Advanced Security."

About SARIF support

SARIF (Static Analysis Results Interchange Format) is an OASIS Standard that defines an output file format. The SARIF standard is used to streamline how static analysis tools share their results. Code scanning supports a subset of the SARIF 2.1.0 JSON schema.

To upload a SARIF file from a third-party static code analysis engine, you'll need to ensure that uploaded files use the SARIF 2.1.0 version. GitHub will parse the SARIF file and show alerts using the results in your repository as a part of the code scanning experience. For more information, see "Uploading a SARIF file to GitHub." For more information about the SARIF 2.1.0 JSON schema, see `sarif-schema-2.1.0.json`.

If you're using GitHub Actions with the CodeQL analysis workflow or using the CodeQL CLI, then the code scanning results will automatically use the supported subset of SARIF 2.1.0. For more information, see "Configuring advanced setup for code scanning" or "Using code scanning with your existing CI system."

GitHub uses properties in the SARIF file to display alerts. For example, the `shortDescription` and `fullDescription` appear at the top of a code scanning alert. The `location` allows GitHub to show annotations in your code file. For more information, see "Managing code scanning alerts for your repository."

If you're new to SARIF and want to learn more, see Microsoft's `SARIF tutorials`

repository.

Providing data to track code scanning alerts across runs

Each time the results of a new code scan are uploaded, the results are processed and alerts are added to the repository. To prevent duplicate alerts for the same problem, code scanning uses fingerprints to match results across various runs so they only appear once in the latest run for the selected branch. This makes it possible to match alerts to the correct line of code when files are edited. The `ruleID` for a result has to be the same across analysis.

Reporting consistent filepaths

The filepath has to be consistent across the runs to enable a computation of a stable fingerprint. If the filepaths differ for the same result, each time there is a new analysis a new alert will be created, and the old one will be closed. This will cause having multiple alerts for the same result.

Including data for fingerprint generation

GitHub uses the `partialFingerprints` property in the OASIS standard to detect when two results are logically identical. For more information, see the "partialFingerprints property" entry in the OASIS documentation.

SARIF files created by the CodeQL analysis workflow, or using the CodeQL CLI include fingerprint data. If you upload a SARIF file using the `upload-sarif` action and this data is missing, GitHub attempts to populate the `partialFingerprints` field from the source files. For more information about uploading results, see "Uploading a SARIF file to GitHub."

If you upload a SARIF file without fingerprint data using the `/code-scanning/sarifs` API endpoint, the code scanning alerts will be processed and displayed, but users may see duplicate alerts. To avoid seeing duplicate alerts, you should calculate fingerprint data and populate the `partialFingerprints` property before you upload the SARIF file. You may find the script that the `upload-sarif` action uses a helpful starting point: https://github.com/github/codeql-action/blob/main/src/fingerprints.ts. For more information about the API, see "Code Scanning."

Understanding rules and results

SARIF files support both rules and results. The information stored in these elements is similar but serves different purposes.

- Rules are an array of `reportingDescriptor` objects that are included in the `toolComponent` object. This is where you store details of the rules that are run during analysis. Information in these objects should change infrequently, typically when you update the tool.
- Results are stored as a series of `result` objects under `results` in the `run` object. Each `result` object contains details for one alert in the codebase. Within the `results` object, you can reference the rule that detected the alert.

When you compare SARIF files generated by analyzing different codebases with the same tool and rules, you should see differences in the results of the analyses but not in the rules.

Specifying the root for source files

Code scanning interprets results that are reported with relative paths as relative to the root of the repository analyzed. If a result contains an absolute URI, the URI is converted to a relative URI. The relative URI can then be matched against a file committed to the repository.

You can provide the source root for conversion from absolute to relative URIs in one of the following ways.

- `checkout_path` input to the `github/codeql-action/analyze` action
- `checkout_uri` parameter to the SARIF upload API endpoint. For more information, see "Code Scanning" in the REST API documentation
- `invocation.workingDirectory.uri` property in the SARIF file

If you provide a source root, any location of an artifact specified using an absolute URI must use the same URI scheme. If there is a mismatch between the URI scheme for the source root and one or more of the absolute URIs, the upload is rejected.

For example, a SARIF file is uploaded using a source root of `file:///github/workspace`.

```
# Conversion of absolute URIs to relative URIs for location artifacts

file:///github/workspace/src/main.go -> src/main.go
file:///tmp/go-build/tmp.go          -> file:///tmp/go-build/tmp.go
```

The file is successfully uploaded as both absolute URIs use the same URI scheme as the source root.

Validating your SARIF file

You can check a SARIF file is compatible with code scanning by testing it against the GitHub ingestion rules. For more information, visit the Microsoft SARIF validator.

For each gzip-compressed SARIF file, SARIF upload supports a maximum size of 10 MB. Any uploads over this limit will be rejected. If your SARIF file is too large because it contains too many results, you should update the configuration to focus on results for the most important rules or queries. For more information, see "SARIF results file is too large."

Code scanning supports uploading a maximum number of entries for the data objects in the following table. If any of these objects exceeds its maximum value the SARIF file is rejected. For some objects, there is also a additional limit on the number of values that will be displayed. Whenever possible the most important values are shown. To get the most out of your analysis when it includes data above the supported limits, try to optimize the analysis configuration (for example, for the CodeQL tool, identify and disable the most noisy queries). For more information, see "SARIF results exceed one or more limits."

SARIF data	Maximum values	Display limits
Runs per file	20	None
Results per run	25,000	Only the top 5,000 results will be included, prioritized by severity.
Rules per run	25,000	None
Tool extensions per run	100	None

Thread Flow Locations per result	10,000	Only the top 1,000 Thread Flow Locations will be included, using prioritization.
Location per result	1,000	Only 100 locations will be included.
Tags per rule	20	Only 10 tags will be included.

For information about other errors, see "Troubleshooting SARIF uploads"

Uploading more than one SARIF file for a commit

You can upload multiple SARIF files for the same commit, and display the data from each file as code scanning results. When you upload multiple SARIF files for a commit, you must indicate a "category" for each analysis. The way to specify a category varies according to the analysis method:

- Using the CodeQL CLI directly, pass the `--sarif-category` argument to the `codeql database analyze` command when you generate SARIF files. For more information, see "About the CodeQL CLI."
- Using GitHub Actions with `codeql-action/analyze`, the category is set automatically from the workflow name and any matrix variables (typically, `language`). You can override this by specifying a `category` input for the action, which is useful when you analyze different sections of a monorepo in a single workflow.
- Using GitHub Actions to upload results from other static analysis tools, then you must specify a `category` input if you upload more than one file of results for the same tool in one workflow. For more information, see "Uploading a SARIF file to GitHub."
- If you are not using either of these approaches, you must specify a unique `runAutomationDetails.id` in each SARIF file to upload. For more information about this property, see "`runAutomationDetails` object."

If you upload a second SARIF file for a commit with the same category and from the same tool, the earlier results are overwritten. However, if you try to upload multiple SARIF files for the same tool and category in a single GitHub Actions workflow run, the misconfiguration is detected and the run will fail.

Supported SARIF output file properties

If you use a code analysis engine other than CodeQL, you can review the supported SARIF properties to optimize how your analysis results will appear on GitHub.

> **Note:** You must supply an explicit value for any property marked as "required". The empty string is not supported for required properties.

Any valid SARIF 2.1.0 output file can be uploaded, however, code scanning will only use the following supported properties.

`sarifLog` object

Name	Required	Description
`$schema`	✓	The URI of the SARIF JSON schema for version 2.1.0. For example, `https://json.schemastore.org/s`

		`arif-2.1.0.json`.
`version`	✓	Code scanning only supports SARIF version `2.1.0`.
`runs[]`	✓	A SARIF file contains an array of one or more runs. Each run represents a single run of an analysis tool. For more information about a `run`, see the `run` object.

`run` object

Code scanning uses the `run` object to filter results by tool and provide information about the source of a result. The `run` object contains the `tool.driver` tool component object, which contains information about the tool that generated the results. Each `run` can only have results for one analysis tool.

Name	Required	Description
`tool.driver`	✓	A `toolComponent` object that describes the analysis tool. For more information, see the `toolComponent` object.
`tool.extensions[]`	✗	An array of `toolComponent` objects that represent any plugins or extensions used by the tool during analysis. For more information, see the `toolComponent` object.
`invocation.workingDirectory.uri`	✗	This field is used only when `checkout_uri` (SARIF upload API only) or `checkout_path` (GitHub Actions only) are not provided. The value is used to convert absolute URIs used in `physicalLocation` objects to relative URIs. For more information, see "Specifying the root for source files."
`results[]`	✓	The results of the analysis tool. Code scanning displays the results on GitHub. For more information, see the `result` object.

`toolComponent` object

Name	Required	Description
`name`	✓	The name of the analysis tool. Code scanning displays the name on GitHub to allow you to filter results by tool.
`version`	✗	The version of the analysis tool. Code scanning uses the version number to track when results

		may have changed due to a tool version change rather than a change in the code being analyzed. If the SARIF file includes the `semanticVersion` field, `version` is not used by code scanning.
`semanticVersion`	✗	The version of the analysis tool, specified by the Semantic Versioning 2.0 format. Code scanning uses the version number to track when results may have changed due to a tool version change rather than a change in the code being analyzed. If the SARIF file includes the `semanticVersion` field, `version` is not used by code scanning. For more information, see "Semantic Versioning 2.0.0" in the Semantic Versioning documentation.
`rules[]`	✓	An array of `reportingDescriptor` objects that represent rules. The analysis tool uses rules to find problems in the code being analyzed. For more information, see the `reportingDescriptor` object.

`reportingDescriptor` object

This is where you store details of the rules that are run during analysis. Information in these objects should change infrequently, typically when you update the tool. For more information, see "Understanding rules and results" above.

Name	Required	Description
`id`	✓	A unique identifier for the rule. The `id` is referenced from other parts of the SARIF file and may be used by code scanning to display URLs on GitHub.
`name`	✗	The name of the rule. Code scanning displays the name to allow results to be filtered by rule on GitHub. Limited to 255 characters.
`shortDescription.text`	✓	A concise description of the rule. Code scanning displays the short description on GitHub next to the associated results. Limited to 1024 characters.
`fullDescription.text`	✓	A description of the rule. Code scanning displays the full description on GitHub next to

		the associated results. Limited to 1024 characters.
`defaultConfiguration.level`	✕	Default severity level of the rule. Code scanning uses severity levels to help you understand how critical the result is for a given rule. By default, the `defaultConfiguration.level` is set to `warning`. However, you can override the default level for a rule by setting the `level` attribute in the `result` object associated with the result. For more information, please refer to the documentation on the `result` object. The valid values for `defaultConfiguration.level` are: `note`, `warning` and `error`.
`help.text`	✓	Documentation for the rule using text format. Code scanning displays this help documentation next to the associated results.
`help.markdown`	✕	(Recommended) Documentation for the rule using Markdown format. Code scanning displays this help documentation next to the associated results. When `help.markdown` is available, it is displayed instead of `help.text`.
`properties.tags[]`	✕	An array of strings. Code scanning uses `tags` to allow you to filter results on GitHub. For example, it is possible to filter to all results that have the tag `security`.
`properties.precision`	✕	(Recommended) A string that indicates how often the results indicated by this rule are true. For example, if a rule has a known high false-positive rate, the precision should be `low`. Code scanning orders results by precision on GitHub so that the results with the highest `level`, and highest `precision` are shown first. Can be one of: `very-high`, `high`, `medium`, or `low`.
`properties.problem.severity`	✕	(Recommended) A string that indicates the level of severity of any alerts generated by a non-security query. This, with the `properties.precision` property, determines whether the results are displayed by

		default on GitHub so that the results with the highest `problem.severity`, and highest `precision` are shown first. Can be one of: `error`, `warning`, or `recommendation`.
`properties.security-severity`	×	(Recommended only for security rules) If you include a value for this field, results for the rule are treated as security results. A string representing a score that indicates the level of severity, between 0.0 and 10.0, for security queries (`@tags` includes `security`). This, with the `properties.precision` property, determines whether the results are displayed by default on GitHub so that the results with the highest `security-severity`, and highest `precision` are shown first. Code scanning translates numerical scores as follows: over 9.0 is `critical`, 7.0 to 8.9 is `high`, 4.0 to 6.9 is `medium` and 3.9 or less is `low`.

`result` object

Each `result` object contains details for one alert in the codebase. Within the `results` object, you can reference the rule that detected the alert. For more information, see "Understanding rules and results" above.

You can check that the SARIF properties have the supported size for upload and that the file is compatible with code scanning. For more information, see "SARIF support for code scanning".

Name	Required	Description
`ruleId`	×	The unique identifier of the rule (`reportingDescriptor.id`). For more information, see the `reportingDescriptor` object. Code scanning uses the rule identifier to filter results by rule on GitHub.
`ruleIndex`	×	The index of the associated rule (`reportingDescriptor` object) in the tool component `rules` array. For more information, see the `run` object. The allowed range for this property 0 to 2^63 - 1.
`rule`	×	A reference used to locate the rule (reporting descriptor) for this result. For more information, see the `reportingDescriptor` object.

`level`	✗	The severity of the result. This level overrides the default severity defined by the rule. Code scanning uses the level to filter results by severity on GitHub.
`message.text`	✓	A message that describes the result. Code scanning displays the message text as the title of the result. Only the first sentence of the message will be displayed when visible space is limited.
`locations[]`	✓	The set of locations where the result was detected up to a maximum of 10. Only one location should be included unless the problem can only be corrected by making a change at every specified location. **Note:** At least one location is required for code scanning to display a result. Code scanning will use this property to decide which file to annotate with the result. Only the first value of this array is used. All other values are ignored.
`partialFingerprints`	✓	A set of strings used to track the unique identity of the result. Code scanning uses `partialFingerprints` to accurately identify which results are the same across commits and branches. Code scanning will attempt to use `partialFingerprints` if they exist. If you are uploading third-party SARIF files with the `upload-action`, the action will create `partialFingerprints` for you when they are not included in the SARIF file. For more information, see "Providing data to track code scanning alerts across runs." **Note:** Code scanning only uses the `primaryLocationLineHash`.
`codeFlows[].threadFlows[].locations[]`	✗	An array of `location` objects for a `threadFlow` object, which describes the progress of a program through a thread of execution. A `codeFlow` object describes a pattern of code execution used to detect a result. If code flows are provided, code scanning will expand code flows on GitHub for the relevant result. For more information, see the `location` object.

`relatedLocations[]`	✗	A set of locations relevant to this result. Code scanning will link to related locations when they are embedded in the result message. For more information, see the `location` object.

`location` object

A location within a programming artifact, such as a file in the repository or a file that was generated during a build.

Name	Required	Description
`location.id`	✗	A unique identifier that distinguishes this location from all other locations within a single result object. The allowed range for this property 0 to 2^63 - 1.
`location.physicalLocation`	✓	Identifies the artifact and region. For more information, see the `physicalLocation`.
`location.message.text`	✗	A message relevant to the location.

`physicalLocation` object

Name	Required	Description
`artifactLocation.uri`	✓	A URI indicating the location of an artifact, usually a file either in the repository or generated during a build. For the best results we recommend that this is a relative path from the root of the GitHub repository being analyzed. For example, `src/main.js`. For more information about artifact URIs, see "Specifying the root for source files."
`region.startLine`	✓	The line number of the first character in the region.
`region.startColumn`	✓	The column number of the first character in the region.
`region.endLine`	✓	The line number of the last character in the region.
`region.endColumn`	✓	The column number of the character following the end of the region.

`runAutomationDetails` object

The `runAutomationDetails` object contains information that specifies the identity of a run.

Name	Required	Description
`id`	×	A string that identifies the category of the analysis and the run ID. Use if you want to upload multiple SARIF files for the same tool and commit, but performed on different languages or different parts of the code.

The use of the `runAutomationDetails` object is optional.

The `id` field can include an analysis category and a run ID. We don't use the run ID part of the `id` field, but we store it.

Use the category to distinguish between multiple analyses for the same tool or commit, but performed on different languages or different parts of the code. Use the run ID to identify the specific run of the analysis, such as the date the analysis was run.

`id` is interpreted as `category/run-id`. If the `id` contains no forward slash (`/`), then the entire string is the `run_id` and the `category` is empty. Otherwise, `category` is everything in the string until the last forward slash, and `run_id` is everything after.

`id`	`category`	`run_id`
my-analysis/tool1/2021-02-01	my-analysis/tool1	2021-02-01
my-analysis/tool1/	my-analysis/tool1	None
my-analysis for tool1	None	my-analysis for tool1

- The run with an `id` of "my-analysis/tool1/2021-02-01" belongs to the category "my-analysis/tool1". Presumably, this is the run from February 2, 2021.
- The run with an `id` of "my-analysis/tool1/" belongs to the category "my-analysis/tool1" but is not distinguished from other runs in that category.
- The run whose `id` is "my-analysis for tool1 " has a unique identifier but cannot be inferred to belong to any category.

For more information about the `runAutomationDetails` object and the `id` field, see runAutomationDetails object in the OASIS documentation.

Note that the rest of the supported fields are ignored.

SARIF output file examples

These example SARIF output files show supported properties and example values.

Example with minimum required properties

This SARIF output file has example values to show the minimum required properties for code scanning results to work as expected. If you remove any properties, omit values, or use an empty string, this data will not be displayed correctly or sync on GitHub.

```
{
  "$schema": "https://json.schemastore.org/sarif-2.1.0.json",
  "version": "2.1.0",
  "runs": [
    {
      "tool": {
        "driver": {
          "name": "Tool Name",
          "rules": [
            {
              "id": "R01"
                      ...
              "properties" : {
                 "id" : "java/unsafe-deserialization",
                 "kind" : "path-problem",
                 "name" : "...",
                 "problem.severity" : "error",
                 "security-severity" : "9.8",
               }
            }
          ]
        }
      },
      "results": [
        {
          "ruleId": "R01",
          "message": {
            "text": "Result text. This result does not have a rule associated."
          },
          "locations": [
            {
              "physicalLocation": {
                "artifactLocation": {
                  "uri": "fileURI"
                },
                "region": {
                  "startLine": 2,
                  "startColumn": 7,
                  "endColumn": 10
                }
              }
            }
          ],
          "partialFingerprints": {
            "primaryLocationLineHash": "39fa2ee980eb94b0:1"
          }
        }
      ]
    }
  ]
}
```

Example showing all supported SARIF properties

This SARIF output file has example values to show all supported SARIF properties for code scanning.

```
{
  "$schema": "https://json.schemastore.org/sarif-2.1.0.json",
  "version": "2.1.0",
  "runs": [
    {
      "tool": {
        "driver": {
          "name": "Tool Name",
          "semanticVersion": "2.0.0",
          "rules": [
```

```
            {
              "id": "3f292041e51d22005ce48f39df3585d44ce1b0ad",
              "name": "js/unused-local-variable",
              "shortDescription": {
                "text": "Unused variable, import, function or class"
              },
              "fullDescription": {
                "text": "Unused variables, imports, functions or classes may be a
symptom of a bug and should be examined carefully."
              },
              "defaultConfiguration": {
                "level": "note"
              },
              "properties": {
                "tags": [
                  "maintainability"
                ],
                "precision": "very-high"
              }
            },
            {
              "id": "d5b664aefd5ca4b21b52fdc1d744d7d6ab6886d0",
              "name": "js/inconsistent-use-of-new",
              "shortDescription": {
                "text": "Inconsistent use of 'new'"
              },
              "fullDescription": {
                "text": "If a function is intended to be a constructor, it should
always be invoked with 'new'. Otherwise, it should always be invoked as a normal
function, that is, without 'new'."
              },
              "properties": {
                "tags": [
                  "reliability",
                  "correctness",
                  "language-features"
                ],
                "precision": "very-high"
              }
            },
            {
              "id": "R01"
            }
          ]
        }
      },
      "automationDetails": {
        "id": "my-category/"
      },
      "results": [
        {
          "ruleId": "3f292041e51d22005ce48f39df3585d44ce1b0ad",
          "ruleIndex": 0,
          "message": {
            "text": "Unused variable foo."
          },
          "locations": [
            {
              "physicalLocation": {
                "artifactLocation": {
                  "uri": "main.js",
                  "uriBaseId": "%SRCROOT%"
                },
                "region": {
                  "startLine": 2,
                  "startColumn": 7,
                  "endColumn": 10
                }
              }
            }
          ],
```

```
          "partialFingerprints": {
            "primaryLocationLineHash": "39fa2ee980eb94b0:1",
            "primaryLocationStartColumnFingerprint": "4"
          }
        },
        {
          "ruleId": "d5b664aefd5ca4b21b52fdc1d744d7d6ab6886d0",
          "ruleIndex": 1,
          "message": {
            "text": "Function resolvingPromise is sometimes invoked as a
constructor (for example [here](1)), and sometimes as a normal function (for
example [here](2))."
          },
          "locations": [
            {
              "physicalLocation": {
                "artifactLocation": {
                  "uri": "src/promises.js",
                  "uriBaseId": "%SRCROOT%"
                },
                "region": {
                  "startLine": 2
                }
              }
            }
          ],
          "partialFingerprints": {
            "primaryLocationLineHash": "5061c3315a741b7d:1",
            "primaryLocationStartColumnFingerprint": "7"
          },
          "relatedLocations": [
            {
              "id": 1,
              "physicalLocation": {
                "artifactLocation": {
                  "uri": "src/ParseObject.js",
                  "uriBaseId": "%SRCROOT%"
                },
                "region": {
                  "startLine": 2281,
                  "startColumn": 33,
                  "endColumn": 55
                }
              },
              "message": {
                "text": "here"
              }
            },
            {
              "id": 2,
              "physicalLocation": {
                "artifactLocation": {
                  "uri": "src/LiveQueryClient.js",
                  "uriBaseId": "%SRCROOT%"
                },
                "region": {
                  "startLine": 166
                }
              },
              "message": {
                "text": "here"
              }
            }
          ]
        },
        {
          "ruleId": "R01",
          "message": {
            "text": "Specifying both [ruleIndex](1) and [ruleID](2) might lead to
inconsistencies."
          },
```

```
          "level": "error",
          "locations": [
            {
              "physicalLocation": {
                "artifactLocation": {
                  "uri": "full.sarif",
                  "uriBaseId": "%SRCROOT%"
                },
                "region": {
                  "startLine": 54,
                  "startColumn": 10,
                  "endLine": 55,
                  "endColumn": 25
                }
              }
            }
          ],
          "relatedLocations": [
            {
              "id": 1,
              "physicalLocation": {
                "artifactLocation": {
                  "uri": "full.sarif"
                },
                "region": {
                  "startLine": 81,
                  "startColumn": 10,
                  "endColumn": 18
                }
              },
              "message": {
                "text": "here"
              }
            },
            {
              "id": 2,
              "physicalLocation": {
                "artifactLocation": {
                  "uri": "full.sarif"
                },
                "region": {
                  "startLine": 82,
                  "startColumn": 10,
                  "endColumn": 21
                }
              },
              "message": {
                "text": "here"
              }
            }
          ],
          "codeFlows": [
            {
              "threadFlows": [
                {
                  "locations": [
                    {
                      "location": {
                        "physicalLocation": {
                          "region": {
                            "startLine": 11,
                            "endLine": 29,
                            "startColumn": 10,
                            "endColumn": 18
                          },
                          "artifactLocation": {
                            "uriBaseId": "%SRCROOT%",
                            "uri": "full.sarif"
                          }
                        },
                        "message": {
```

```
                        "text": "Rule has index 0"
                      }
                    }
                  },
                  {
                    "location": {
                      "physicalLocation": {
                        "region": {
                          "endColumn": 47,
                          "startColumn": 12,
                          "startLine": 12
                        },
                        "artifactLocation": {
                          "uriBaseId": "%SRCROOT%",
                          "uri": "full.sarif"
                        }
                      }
                    }
                  }
                ]
              }
            ]
          }
        ],
        "partialFingerprints": {
          "primaryLocationLineHash": "ABC:2"
        }
      }
    ],
    "columnKind": "utf16CodeUnits"
  }
]
}
```