



Coco Code Coverage

Moving Away From Assumed Test Effectiveness to
Measured Code Execution

Supporting **C, C++, C#, QML, Tcl & Python** — from
desktop to safety-critical embedded systems

What Measured Code Coverage Makes Possible

A passing test suite is reassuring. But passing tests and well-tested code are not the same thing.

Tests can complete successfully while entire branches of logic go unexecuted, edge cases remain unexplored, and error-handling paths are never triggered. Without coverage measurement, there is no reliable way to know which lines, decisions, and conditions were actually exercised during testing, and which were quietly skipped.

That gap between assumed coverage and measured coverage is where production defects live, and code coverage makes it visible, measurable, and actionable.

Teams, measuring code coverage with Coco, gain:

- **Full execution visibility** — see exactly which statements, branches, and conditions ran during testing, not just whether tests passed
- **Production confidence** — catch coverage gaps before they reach customers, rather than discovering them through defects in the field
- **Targeted retesting** — understand which tests are affected by a code change, so teams retest precisely instead of broadly
- **Audit-ready data** — coverage reports with decision, condition, and MC/DC criteria built in, ready for reviews and certification without rework
- **Full-stack coverage** — C, C++ and Python layers measured together, closing the gaps that partial tooling leaves behind

Measured code coverage provides objective insight into test execution. It makes untested behaviour visible, supports earlier detection of gaps in testing, and improves confidence in release decisions. The result is clearer insight into software risk and a more accurate understanding of test effectiveness: **In regulated environments**, this measured evidence is also required to demonstrate that verification objectives have been met.

Why Common Approaches Fall Short

Many teams find themselves in a situation where coverage data exists but is not reliable enough for decision-making, optimisation, or certification.

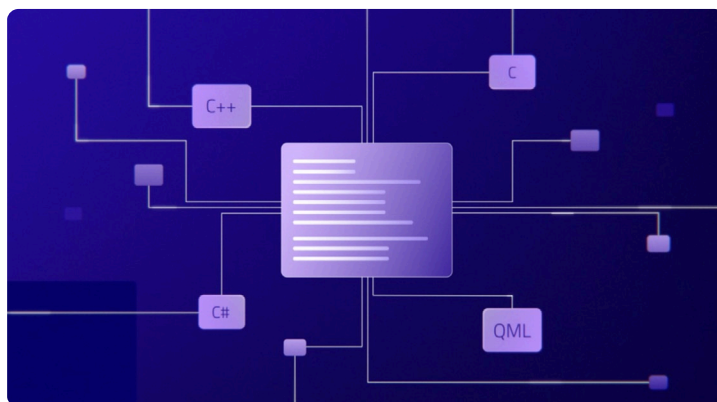
Many commonly used coverage approaches provide limited or incomplete insight:

- Compiler-generated coverage is typically limited to basic metrics and is not designed for embedded or cross-compiled workflows
- Unit-test coverage alone often fails to capture system-level and hardware-dependent behaviour, especially in embedded contexts
- Manual reviews and traceability provide valuable context, but cannot substitute for objective runtime execution data

Coco addresses these gaps through compiler-level instrumentation that works across languages, platforms, and real hardware targets without replacing existing workflows or requiring tests to be rewritten.

What Coco Delivers

Coco is a modern code coverage tool designed for software teams developing both desktop and embedded systems. It supports C, C++, C#, QML, Tcl, and Python (via coverage.py integration), using compiler-level instrumentation to capture detailed execution data — including statements, branches, conditions, and MC/DC — across a wide range of platforms.



If you experience	With Coco, you can
Insufficient visibility into untested logic	Identify untested logic, code, and unexecuted statements/conditions through instrumentation and coverage metrics.
Inefficient dynamic testing efforts	View which statements, branches, and conditions were executed during testing so that teams can improve and optimize unit, functional, and manual testing. Coco can also analyze coverage from manually executed runs if instrumentation is present.
The need for consistent coverage instrumentation across supported compilers and platforms	Produce coverage for C, C++, C#, QML, Tcl across desktop, embedded, desktop, embedded, and real hardware target, overcoming the fragmentation limitations of many other tools.
Growing volumes of untested code during ongoing development	Identify the tests needed for modified code using patch analysis, avoiding running all tests by selecting only those affected by the code changes.
High-risk release cycles	Get patch-based test recommendations to better control and assure during release preparation, especially in safety-critical contexts.
Uncertainty about system behaviour	Get actual runtime execution paths exposed with Coco's instrumentation, making internal behaviour observable.
Difficulty demonstrating test completeness in audits	Generate audit-ready, standards-aligned reports for ISO 26262, DO-178C, DO-330, IEC 61508, IEC 62304.

How Coco Produces Coverage Results

Outcome: Coverage results remain consistent across environments and over time.

Coco instruments the software during the build process and generates a structural description of the executable. During test execution, runtime data records which statements, decisions, and conditions were exercised.

- Coverage data from multiple runs, configurations, and targets can be merged and analysed together.



You can't manage what you can't see.

Coco shows which code actually executed during testing, and which didn't.

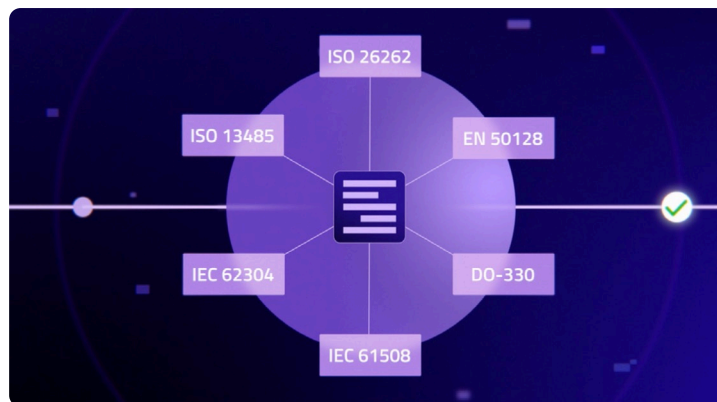
Get a free trial and see it in your own codebase.

Supported Certification and Regulatory Requirements

Coco supports structural coverage criteria commonly referenced or required by regulatory standards, including statement, decision, condition, and Modified Condition/Decision Coverage (MC/DC). Coverage results are derived from real runtime execution and can be consolidated across multiple runs, configurations, and targets.

Outcome: Teams can generate traceable, reviewable coverage evidence that aligns with certification expectations and supports audits, assessments, and long-lifecycle compliance activities.

Built for environments that demand reliability and traceability, Coco generates audit-ready coverage reports aligned with major international safety and compliance standards — ISO 26262, DO-330, DO-178C, IEC 61508, and IEC 62304 — making it well suited for automotive, aerospace, industrial, and medical software.



Coco is assessed by TÜV (Technischer Überwachungsverein), an independent certification body that evaluates tools against recognised safety and quality standards. This qualification directly reduces your team's tool qualification burden — Coco is pre-validated for use in projects targeting ASIL-D under ISO 26262, and equivalent levels under DO-178C and IEC 62304. You don't need to build qualification evidence from scratch.

Tool Qualification for Regulated Industries



The Coco Qualification Kit gives you everything needed to demonstrate that Coco is safe and reliable for use in safety-critical development — without building test suites or writing justification from scratch.

Qualification Kits are available separately per standard — ISO 26262, DO-178C, IEC 62304, and EN 50128 — so teams working under a specific standard receive targeted documentation and pre-built test cases relevant to their certification context.

These kits provide documentation, qualification material, and guidance to support the use of Coco within a defined development and verification context, in accordance with applicable safety standards.

CRAP Metric — Turning Coverage Gaps into a Risk Ranking

Knowing you have 80% coverage doesn't tell you whether the untested 20% is trivial or critical. Coco's CRAP (Change Risk Anti-Patterns) metric does. It combines complexity analysis with coverage data into a risk score for every function — so instead of guessing where to focus, teams work from a ranked list of the functions most likely to cause problems when changed. The higher the complexity and the lower the coverage, the higher the score. Functions above 30 are flagged as high-risk in CoverageBrowser, rising to the top of a sortable list so engineers and QA leads know exactly where a new test will reduce the most risk without reviewing the entire codebase. Results are exportable for reporting and review.

Function	Decision %	CRAP Metric - Change Risk Anti-Patterns
f() Parser::eval_function(const std::string&...)	0.000% (0/42)	240.0
f() Parser::eval_operator(const int op_id, ...)	21.428% (9/37)	192.8
f() Error::msgdesc(const int id)	13.793% (4/29)	159.1
f() Parser::get_operator_id(const std::string&...)	49.091% (27/55)	66.6
f() factorial(double value)	0.000% (0/10)	20.0
f() Parser::getToken()	92.000% (46/50)	17.1
f() Parser::parse_level100()	20.000% (2/10)	12.2
f() sign(double value)	0.000% (0/7)	12.0
f() Parser::parse(const std::string&new_...)	36.364% (4/11)	11.4
f() Parser::parse_number()	50.000% (5/10)	8.1
f() toupper(char upper[], const char str[])	0.000% (0/4)	6.0
f() isNotDelimiter(const char c)	0.000% (0/4)	6.0

Typical Results Achieved with Coco

The screenshot displays the Qt IDE interface with COCO analysis results. The main window shows the source code for 'parser.cpp' with line numbers 176 to 203. The left sidebar contains several panels:

- Statistics:** Shows overall project statistics: MCJDC: 44.28% (146/425).
- Functions:** Lists functions with their MCJDC percentages:

Function	MCJDC %
ParserResult	100.000% (9/9)
Parser	0.000% (0/1)
- Sources:** Lists source files with their MCJDC percentages:

Source	MCJDC %
variablelist.h	0.000% (0/0)
variablelist.cpp	57.576% (19/33)
parser.h	100.000% (1/1)
- Metrics:** Shows McCabe - Cyclomatic Complexity:

Project	Cumulated
MCJDC	179
Average	13.89
Deviation	23.36
Minimum	0
Maximum	125
- Function Profiler:** Shows execution statistics:

Function	Total Duration	Count	Mean Duration
isDigit	0.3...	506747	0.000000739 s
isAlpha	0.1...	98076...	0.000000013 s
isDigit	0.0...	60	0.000000025 s

The code editor on the right shows the implementation of the `Parser::getToken()` function, with line numbers 176 to 203. The code includes comments and logic for tokenization, such as skipping whitespaces and checking for end of expression or minus signs.

Technical Snapshot

Coverage Levels *

```
75 /*
76  * Get value of variable with given name
77  */
78 bool Variablelist::get_value(const char* name,
79                             double* value)
80 {
81     int id = get_id(name);
82     if (id != -1)
83     {
84         *value = var[id].value;
85         return true;
86     }
87     return false;
88 }
```

Visualization: Source code is colour-coded to show at a glance whether a line, branch, or condition was covered or not.

- **Function Coverage** — Counts how many functions were called and how often, including member functions in object-oriented languages like C++.
- **Line Coverage** — The number of executed lines divided by total lines. Only lines containing executable statements are considered.
- **Statement Coverage** — Tracks executed program statements as a ratio of executed to total statements.
- **Decision Coverage (Branch Coverage)** — Verifies that all statements are executed and all decisions produce all possible outcomes. Each decision counts twice: once for true, once for false.
- **Condition Coverage** — Like Decision Coverage, but decisions are split into elementary subexpressions connected by AND/OR operators. Each condition counts twice.
- **MC/DC – Modified Condition/Decision Coverage** — Every condition in a decision must be tested independently to reach full coverage, with each condition shown to independently affect the outcome.
- **MCC – Multiple Condition Coverage** — All combinations of truth values in each decision must occur at least once to reach full coverage.

* The Call Coverage metric is implicitly covered by Condition Coverage, MC/DC, and MCC.

Technical Snapshot

Supported Languages

- C, C++, C#, QML, Tcl
 - SystemC (when compiled as C++ code)
 - Python (via Coverage.py for Python modules, with Coco handling instrumentation for native C/C++/C#/QML/Tcl layers)
-

Supported Environments

Desktop & Server: Windows, Linux, macOS, Unix variants (including Solaris)

Embedded & Real-Time: Embedded Linux, Embedded Windows, QNX, VxWorks, FreeRTOS, bare-metal via cross-compiled targets

Hardware Targets: MCUs, MPUs, SoCs

Qt for MCU (Qt Quick Ultralite / QUL): Coco instruments the C/C++ backend logic used within Qt for MCU applications.

Compiler & Toolchain Support

Coco integrates with a wide range of compilers, including GCC, Clang, Microsoft Visual Studio, Intel C/C++, Oracle/Sun Studio, Mono C#, QNX (qcc, q++), ARM/Keil μ Vision, Green Hills, HighTec, Atmel Studio, and Wind River Diab.

Several embedded compilers require the Coco Cross-Compilation Add-on. Support for additional toolchains can be added via configuration file — contact us if yours is not listed.

Technical Snapshot

Build Systems, IDEs & Test Frameworks

Build Systems & IDEs: CMake, MSBuild, Visual Studio, Eclipse, Qt Creator

Test Frameworks: GoogleTest, Boost.Test, NUnit, Qt Test, CppUnit, Catch2
Coco does not replace test frameworks and does not require tests to be rewritten. Instrumentation is applied during the build process and fits into existing workflows.

Reporting & Integration

Coco provides flexible reporting for engineering, management, and compliance use, with results available via the interactive CoverageBrowser for source-level visualisation and side-by-side build comparison.

Export formats: HTML, XML, CSV, JUnit, Cobertura, EMMA-XML, SonarQube-compatible formats, plain text

CI/CD integration: Jenkins, GitHub, GitLab — coverage data can be collected automatically during pipeline execution and used for quality gates or dashboards.

Reports support CI dashboards, audits, and long-term traceability.

Get started with Coco by **exploring installation guides, release notes, and step-by-step tutorials** designed to help you integrate coverage quickly and efficiently. Learn how to instrument a simple project, discover new test data with the Coco Test Engine, and understand the fundamentals of coverage analysis. Dive deeper with dedicated guides for measuring code coverage.

Explore the full Coco documentation: <https://doc.qt.io/coco/>

Who Benefits from Coco

- Developers gain a precise view of which code paths are exercised, enabling faster, more targeted fixes.
- QA engineers gain visibility into which parts of the application were exercised by automated and manual tests, so testing is guided by data, not assumptions.
- Test managers and team leads gain objective metrics to track coverage progress, identify risk areas, and support confident release decisions.
- Compliance and safety teams gain measurable, auditable evidence to meet standards and satisfy regulatory requirements.

MEASURE COVERAGE

✓	parser.h	100.00%
✓	parser.cpp	90.67%
✓	parser.cpp	88.23%
✓	functions.cpp	100.00%

Achieve **Coverage Confidence** with Coco

Coco gives every member of your team what they need: developers see exactly which code ran, QA engineers close gaps with data-driven testing, test managers rank every function by the combination of complexity and coverage gap using CRAP scores, and safety teams satisfy audit requirements with TÜV Saar-assessed, standards-aligned evidence.



Learn more about how to measure code coverage **with Coco**.

WHITEPAPER



Code Coverage for **Safety-Critical Programs**

Don't let **code coverage be your weak spot**. Download this whitepaper to get an overview of what code coverage is, and why and to what degree the industry standards require code coverage to attain **certification for safety-critical software**.



Download
the whitepaper.

[qt.io/coco](https://www.qt.io/coco)

Qt Group (Nasdaq Helsinki: QTCOM) is a global software company, trusted by industry leaders and over 1.5 million developers worldwide to create applications and smart devices that users love. We help our customers increase productivity through the entire product development journey: from UI design to software development, optimizing embedded systems, and quality management. Our customers are in more than 70 different industries in over 180 countries. Qt Group employs some 1100 people, and its net sales in 2025 were 216.3 MEUR. To learn more, visit www.qt.io
