PARASOFT®

WHITEPAPER

DevOps Best Practices for Automotive Software Development

OVERVIEW

As the automotive industry continues to embrace an Agile development methodology, they start to uncover other processes that can be accelerated, such as delivery by DevOps, particularly including continuous testing. DevOps is aimed at automating all of the necessary steps required to take application code and deliver it to the end user.

DevOps works like a conveyor belt to move application code through the distinct phases. Today's DevOps and "continuous everything" initiatives require the ability to assess the risks associated with a release candidate—instantly and continuously.

Continuous testing provides an automated, unobtrusive way to obtain immediate feedback on the business risks associated with a software release candidate. It guides development teams to meet business expectations and helps managers make informed trade-off decisions in order to optimize the business value of a release candidate. Much of this is performed through continuous integration (CI).

In this whitepaper, we'll dive into the following best practices for DevOps in automotive software.

» Use Test Automation to Enable Continuous Integration.

» Remove Validation & Verification Roadblocks With Continuous Testing on Host & Target Systems.

» Leverage Containers to Create Consistent, Reproducible, & Secure Development Environments.

» Grow Test Coverage With Automated Test Generation & Smart Test Execution.

» Make Data-Driven Decisions From Centralized Reporting & Analysis.

USE TEST AUTOMATION TO ENABLE CONTINUOUS INTEGRATION

It's important to fully understand the DevOps phases and when to start using the right tools for the best results. There are also various tried and true CI/CD pipeline tools or solutions in the market that support your DevOps deployment. Some tools used by the automotive industry include Parasoft, Jenkins, GitHub, GitLab, Azure DevOps, Bitbucket, Bamboo, Docker (containers), and more. We'll cover a few of these later in this paper.

CONTINUOUS INTEGRATION

One of the core components in DevOps is continuous integration. CI is the merging of code features, fixes, or small changes backed by process which include version control and software build automation. The goal of CI is to establish a consistent and automated way to build, package, and test applications in what is called a software "pipeline." Testing, in most cases, is the roadblock to making

CI seamless and efficient. Manual testing is out of the question, so it comes down to automating testing or not testing at all. Automated testing includes more than just unit testing, it also includes best practices such as static analysis and regression testing. Automated testing generates immediate feedback which is used to adjust and replan for the next iteration or sprint.

TEST AUTOMATION IN CI/CD WORKFLOW

Test automation in DevOps involves automating your testing methods. It's important to be familiar with each of the phases of this methodology and where to trigger your testing verification and validation methods.

You may have also heard the term DevSecOps, which is the practice of integrating security into the DevOps workflow. It requires a mindset shift in teams to integrate security tools and practices into this methodology. Security has become important for many auto manufacturers due to regularoty requirements from WP.29 and the need to incorporate processes from standards like ISO 21434. The image below provides a visual overview of the phases and where test methods are applicable.

An important benefit of the CI/CD pipeline workflow is the feedback loop it provides to make the "continuous" part work properly. Any hangups in the loop cause inefficiencies that lead to delays in development and deployment.

- Plan
- Code (refactor)
- Build
- Test
- Release
- Deploy

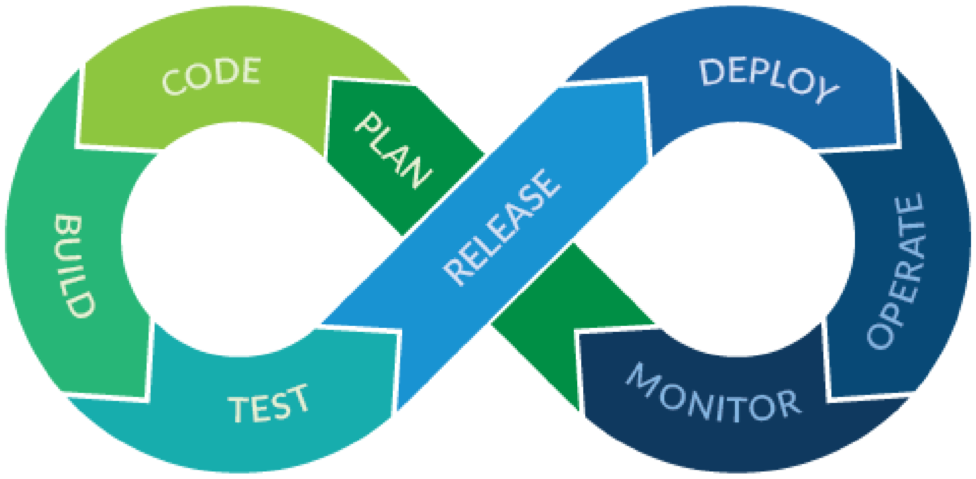


*Figure 1:
The phases in the cycle are represented by the well-known CI/CD and DevOps cycle.*

The essential workflow hinges upon the automated testing aspect. Version control merges all changes to the main repository. After deployment to the test environment, automated testing begins. In the final stage of continuous deployment, this is the last moment when production feedback can take place.

Integration Points for Automated Testing

Test automation, particularly unit testing and static analysis, play the most important role during the code, build, and test phases of the cycle.

» **Code.** Static analysis tools are essential during the coding phase both from a quality and security point of view but also in terms of coding standard compliance. For automotive software that must conform to MISRA C or C++, for example, clean reports at the coding and check-in are needed to ensure continuous compliance.

» **Build.** The build stage triggers both application-wide static analysis and regression testing. At this stage static analysis tools run on the complete scope of the project, offering better precision, and finding potentially missed errors at the unit level. It's also at this stage that regression tests are initiated. The results of regression tests drive the next phase, testing.

» **Test.** At this stage, testing for newly added features begins along with handling failed regressions tests. The results of regression and new tests drive the next planning cycle.

Static analysis tools are ideal for integrating an important security and quality check in the developer workflow. For example, the diagram below shows how static analysis is integrated into both the developer desktop when code is committed but also part of the regular build phase. Results from the build can be used to drive relatively quick fixes before these issues cause more disruption later in the lifecycle.

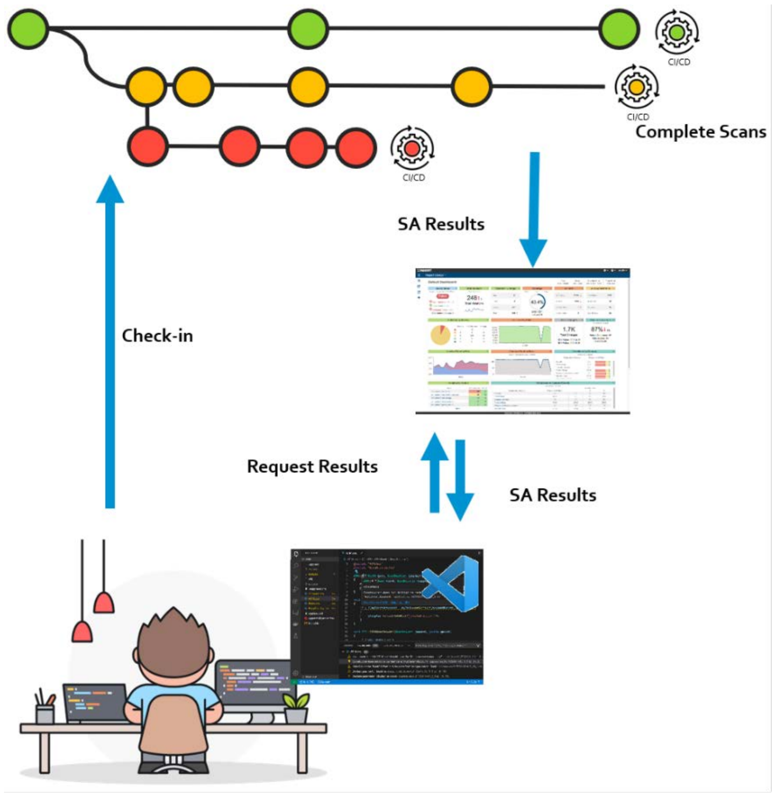


Figure 2:
Static analysis is an important part of automated testing. When done early during code development and refactoring, it eliminates downstream errors that are more difficult and expensive to fix.

The following diagram illustrates how code commits trigger build and test automation. This workflow is in context of the typical development environment, which includes a repository, like Git, developer desktop tools, like IDE, and local and cloud-based build and test servers.

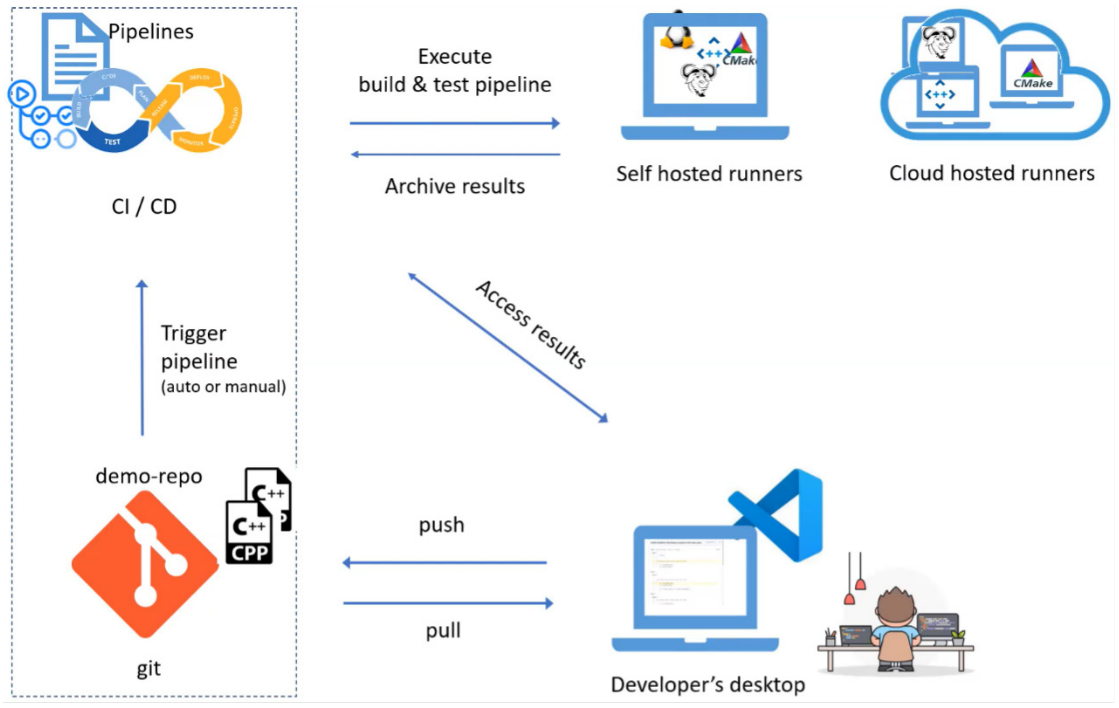
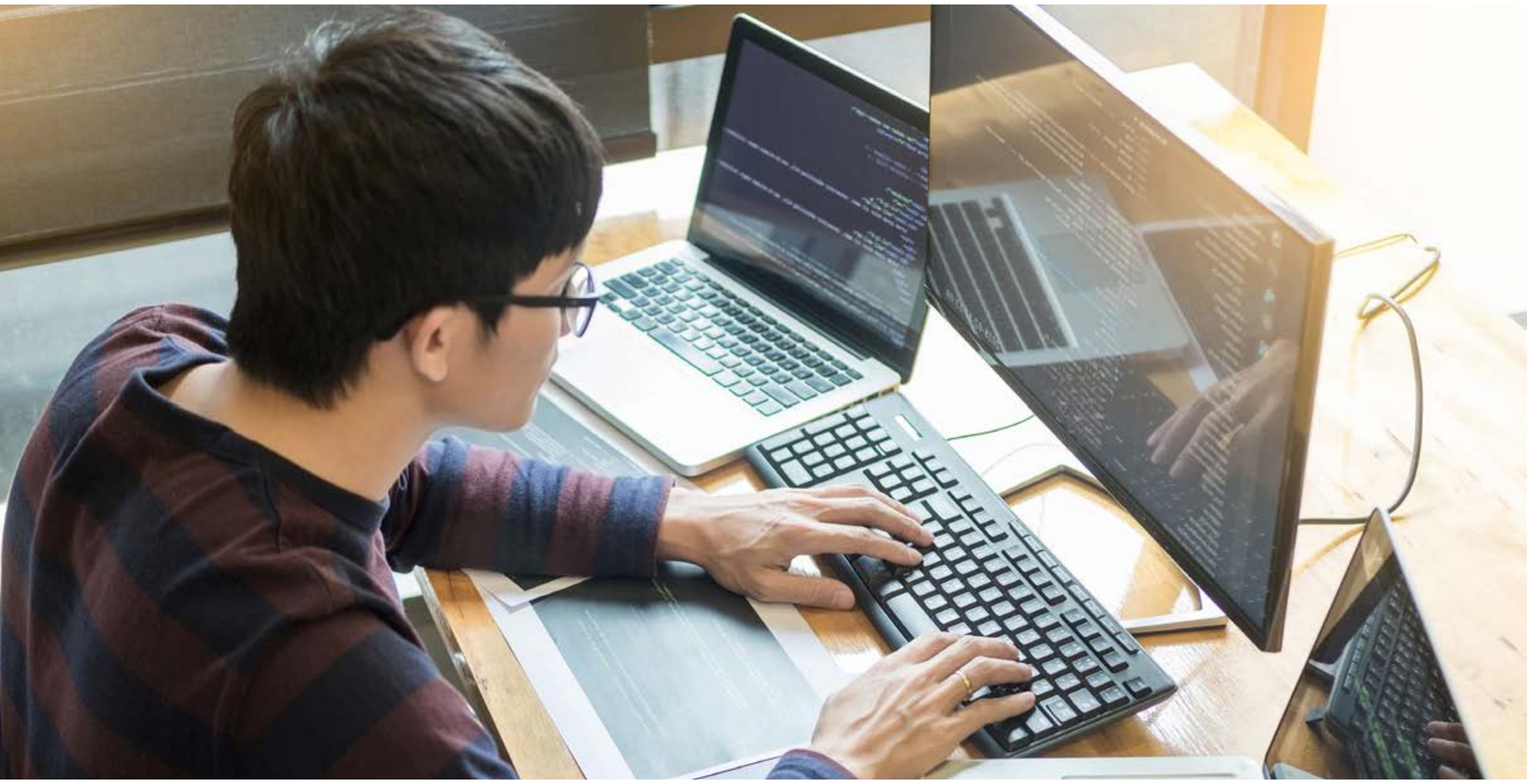


Figure 3:
A continuous integration workflow.

Let's look at this workflow in more detail.

- Typically, after a developer has completed a fix or created new code, a commit into the repository triggers both static analysis on the unit and an incremental build and test.
- Developers use these results to fix and refactor as needed.
- New features imply new unit tests that might be performed individually at first but then become part of the regression test suite soon after.
- Project-wide builds are triggered manually or at certain times of the day. The builds kick off a complete static analysis run and a regression test suite, which now includes new and updated unit tests.

Although this seems straightforward, achieving good results from unit testing is still challenging. Typically, development teams do a minimal amount of unit testing or skip it altogether. Often this is due to some combination of the pressure to deliver more and more functionality and the complexity and time-consuming nature of creating valuable unit tests.



This breaks down into some common reasons developers cite that limit the adoption of unit testing as a core development practice.

» There's a lot of manual coding involved. Sometimes even more than was required to implement a specific feature or enhancement.

» It's difficult to understand, initialize, and/or isolate the dependencies of the unit under test.

» Defining appropriate assertions is time-consuming and often requires cycles of running and manually adjusting tests or performing intelligent guesswork.

» It's just not that interesting. Developers don't want to feel like testers, they want to spend time delivering more functionality.

How Test Automation Tools Help with Unit Testing

Test automation tools provide a way around the following unit test hurdles.

» Unit testing and assertion frameworks provide standardized execution formats like CPPUnit.

» IDE integration delivers unit testing results directly to the developer.

» Guided unit test generation reduces the coding effort.

» Function mocking isolates the code from its dependencies.

» Code coverage shows what code was executed.

» Host and target-based test execution.

By making testing more efficient within the CI/CD pipeline, it's possible to improve test coverage with each cycle and not impede the progress of the project with continuous testing.

REMOVE VALIDATION & VERIFICATION ROADBLOCKS WITH CONTINUOUS TESTING ON HOST & TARGET SYSTEMS

Continuous integration is just part of a continuous development process that needs testing and delivery to reap the benefits of the approach.

Continuous testing provides an automated, unobtrusive way to obtain immediate feedback on a software release candidate. Continuous testing isn't simply more test automation. The purpose is to build quality and security into the product as part of a continuous integration/release/delivery process.

SOME CONTINUOUS TESTING ACTIVITIES

» **Use static analysis early** for detection of bugs and security vulnerabilities. Early detection, usually at the developer's desktop, prevents bugs from wasting unit testing time and entering the software build.

» **Enforce a coding standard to** conform to required corporate and industry standards, like MISRA C/C++ or SEI CERT C. Adhering to a coding standard prevents whole classes of defects and poor coding practices from entering the build to become larger issues later on.

» **Automate test execution on host and target systems** as soon as units are coded and when integrated subsystems are ready. The required tests that need to verify units also include nonfunctional, load, security, and performance testing. These tests are executed directly from the CI orchestration system. The results from these tests get pulled back into the same build and gathered. Code coverage information (statement, branch, and MC/DC) is cross referenced by unit, file, test, and build number.

» **Maintain requirements traceability** to correlate code, tests, and other assets with stakeholder requirements. This provides an objective assessment of the requirements that are working as expected, which ones require validation, and the ones at risk.

» **Use test impact analysis** to focus where testing efforts need to go. From a risk perspective, changed code impacts more than the software itself. It impacts relevant tests and assets. As teams make code changes, questions arise:

  » Do we need new tests or modify existing ones?

  » What are the impacts on dependencies?

  Automation helps teams focus only on the tests that are impacted.

» **Maintain test data automatically to** increase the effectiveness of a continuous testing strategy. Good test data and test data management practices increase coverage and drive more accurate results. However, developing or accessing test data can be a considerable challenge in terms of time, effort, and compliance.

» **Rely on test and data generation** to make continuous testing work. You can continuously generate data appropriate for the type of scenario you're trying to execute instead of trying to rely on production data sources and hoping that all the right data is in the right place. Combining data generation with simulation will allow you to inject the right data in the right place at the right time.

It should be clear at this point that test automation and the tools that support traceability and management of tests enable continuous testing, which is a key aspect of achieving quality and security in a CI/CD pipeline. In turn, it becomes clear that test automation needs to be a focus for improvement and optimization.

CONTINUOUS TESTING ON HOST & EMBEDDED AUTOMOTIVE TARGET SYSTEMS

At the development level, automotive software isn't much different than typical application development. It requires IDEs, compilers, static and dynamic analysis, and build tools. However, tools often target different architectures than they work on, for example, host versus target environment. Versions of tools are important to ensure a homogenous development environment across the team.

Automating testing for automotive software is more challenging due to the complexity of initiating and observing tests on embedded targets, not to mention the limited access to target hardware that software teams have. Software test automation is essential to making automotive testing workable on a continuous basis from the host development system to the target system.

Testing automotive software is particularly time consuming. Automating the regression test suite provides significant time and cost savings. In addition, test results and code coverage data collection from the target system are essential for validation and standards compliance.

Traceability between test cases, test results, source code, and requirements must be recorded and maintained, which means data collection is critical in test execution.

A solution like Parasoft C/C++test comes with an optimized test harness to take minimal additional overhead for the binary footprint and provides it in the form of source code, where it can be customized if platform-specific modifications are required.

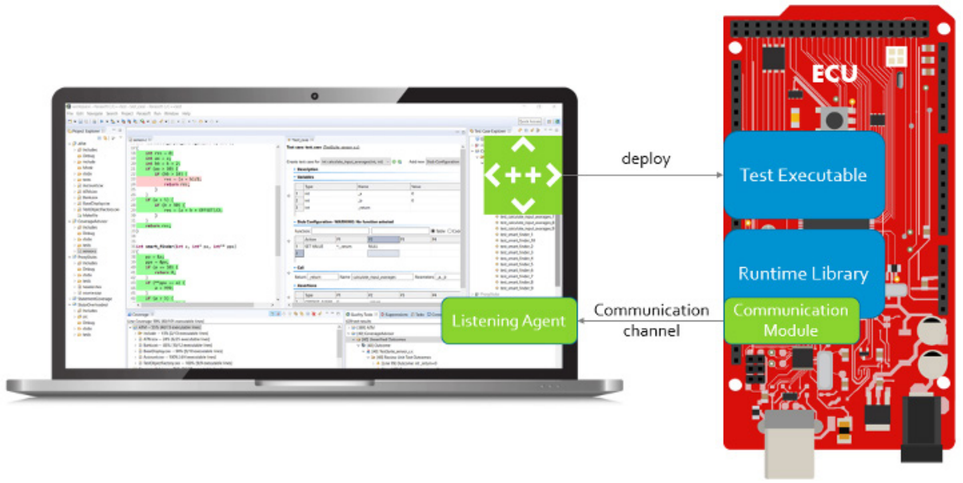


Figure 4:
A high-level view of deploying, executing, and observing tests from host to target.

One huge benefit that the Parasoft C/C++test solution offers is dedicated integrations with embedded IDEs and debuggers that make the process of executing test cases smooth and automated. Supported IDE environments include those listed below. See all technical specifications.

- Eclipse
- VS Code
- MS Visual Studio
- Green Hills Multi
- Wind River Workbench
- IAR EW
- ARM MDK
- ARM DS-5
- TI CCS and more

AUTOMATED REGRESSION TESTING

The Parasoft solution supports the creation of regression testing baselines as an organized collection of tests and will automatically verify all outcomes. These tests run automatically on a regular basis to verify whether code modifications change or break the functionality captured in the regression tests. If any changes are introduced, these test cases will fail to alert the team to the problem. During subsequent tests, C++test will report tasks if it detects changes to the behavior captured in the initial test.

The parity of capabilities of remote target execution with host-based testing means that automotive software teams can reap the same benefits of automation as any other type of application development.

LEVERAGE CONTAINERS TO CREATE CONSISTENT, REPRODUCIBLE, & SECURE DEVELOPMENT ENVIRONMENTS

Development teams know that CI/CD workflows can function the same, regardless of whether they're containerized or not. The value that containers provide DevOps teams is that it allows applications to be easily deployed and patched. Organizations can scale containers to their needs to accelerate development, testing, and production within Agile and DevOps use cases.

When it comes to managing complex development environments, especially in the safety-critical space, teams usually struggle with the following challenges.

» Synchronizing upgrades for the entire team to a new version of a tool like a compiler, build toolchain, and so on.

» Dynamically reacting to a new security patch for the library or software development kit (SDK).

» Assuring consistency of the toolchain for all team members and the automated infrastructure or CI/CD.

» Ability to version the development environment and restore it to service the older version of the product that was certified with a specific toolchain.

» Onboarding and setting up new developers.

All these problems are easy to solve with containers. Within each container is the standardized versions of the development toolchain including static analysis tools, compiler, IDE, and build tools. These containers are updated in a centralized fashion and deployed on an as-needed basis when a developer is working on project code. The diagram below shows a representative example of this.

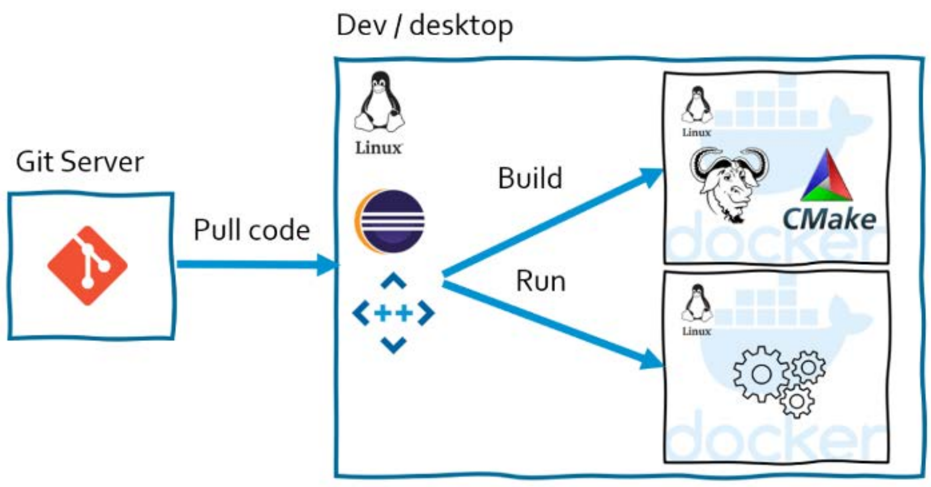


Figure 5: Example deployment using Parasoft C/C++test with a containerized compilation toolchain.

The consistency of the development environment eliminates any errors that might arise from developers having slightly different versions of tools on their desktop. It's a good way to ensure everyone is using the correct version of each tool, which might be important later for regulatory compliance. Containers also provide a centralized way to update tools and ensure these updates are propagated immediately across the organization.

GROW TEST COVERAGE WITH AUTOMATED TEST GENERATION & SMART TEST EXECUTION

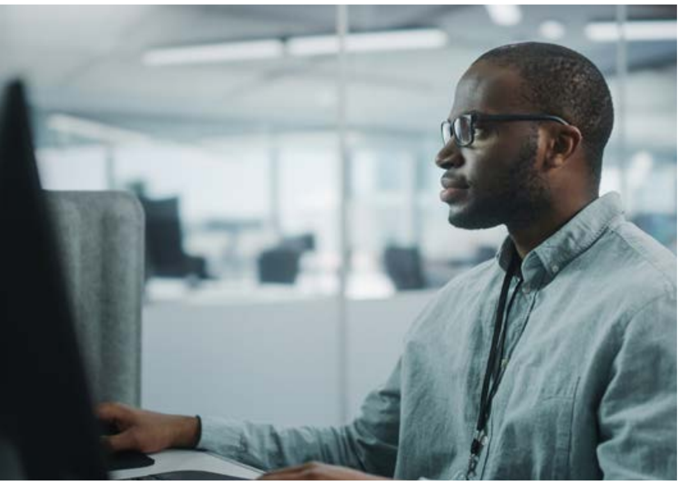
In general, code coverage is a measurement of how much production code is executed while your automated tests are running. By running a suite of tests and looking at code coverage data, there's a general sense of how much of the application is being tested.

There are multiple kinds of code coverage. For automotive systems, there may be highly recommended requirements to perform statement, branch, and MC/DC depending on compliance standards, such as with ISO 26262 ASIL D.

STRUCTURAL CODE COVERAGE

Collecting and analyzing code coverage metrics is an important aspect of safety-critical automotive software development. Code coverage measures the completion of test cases and executed tests. It provides evidence that validation is complete, at least as specified by the software design.

Code coverage also demonstrates the absence of unintended behavior. Code that isn't covered by any test is a liability since its behavior and functionality are unknown. The amount and extent of code coverage depends on the automotive safety integrity level. The higher the integrity level, the higher the rigor used. And, inevitably, the higher the number and complexity of test cases. Below are examples of types of recommended code coverage.



» **Statement coverage** requires that each program statement be executed at least once. Branch and MC/DC coverage encompasses statement coverage.

» **Branch coverage** ensures that each possible decision branch (if-then-else constructs) is executed.

» **Modified condition/decision coverage (MC/DC)** requires the most complete code coverage to ensure test cases executes each decision branch and all the possible combinations of inputs that affect the outcome of decision logic. For complex logic, the number of test cases can explode so the modified condition restrictions are used to limit test cases to those that result in standalone logical expressions changing. See this tutorial from NASA.

Advanced unit test automation tools like Parasoft C/++test provide all of these code coverage metrics and more. C/C++test automates this data collection on host and target testing and accumulates test coverage history over time. This code coverage history can span unit, integration, and system testing to ensure coverage is complete and traceable at all levels of testing.

INCREASE CODE COVERAGE WITH AUTOMATED UNIT TEST CASE CREATION

The creation of productive unit tests has always been a challenge. Functional safety standards compliance demands high-quality software, which drives a need for test suites that affect and produce high code coverage statistics.

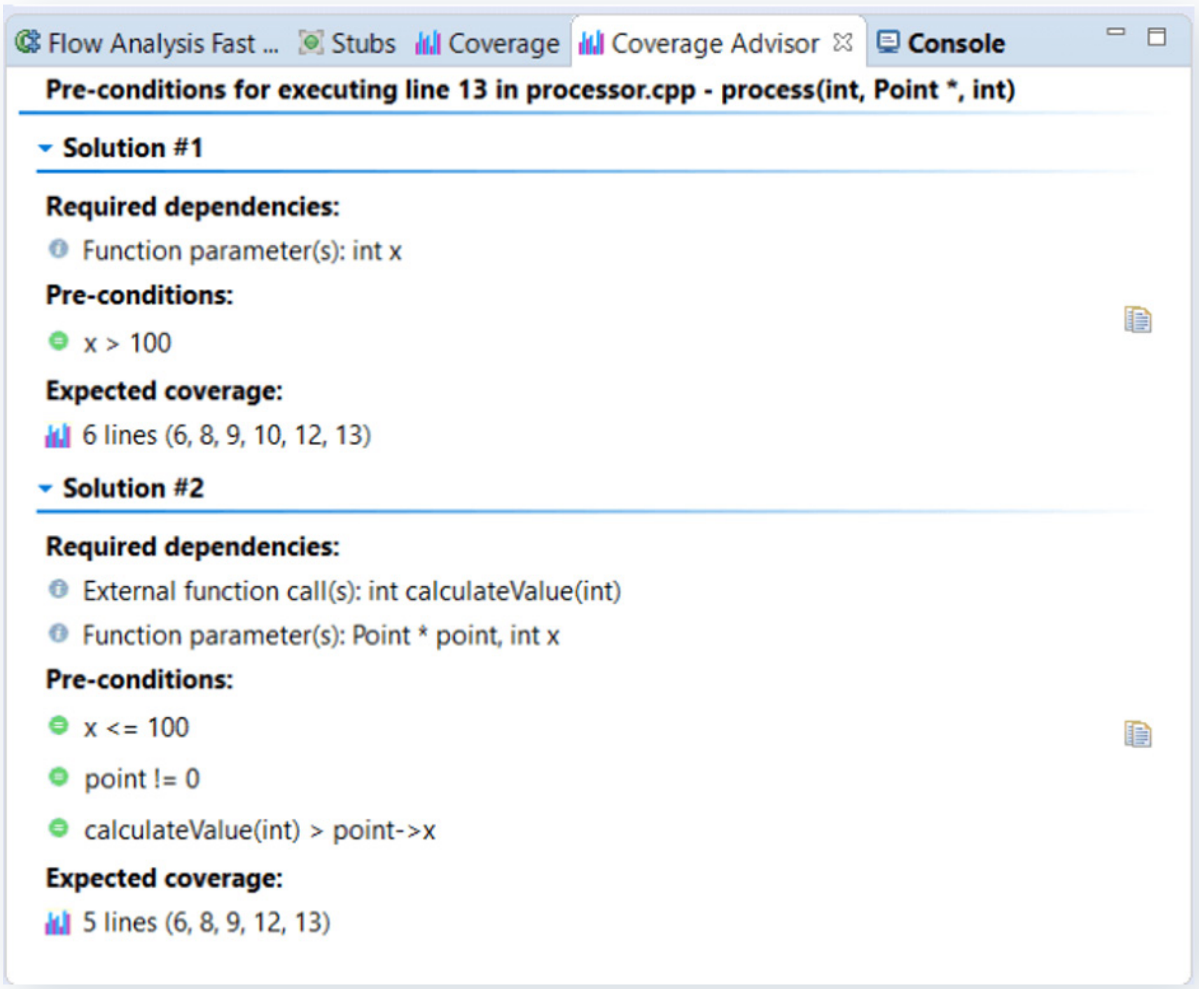
Teams require unit test cases that help them achieve their coverage goals. These goals are important even outside the realm of safety-critical software. Any code not covered by at least one test is shipping untested!

Increasing code coverage can be challenging. Analyzing branches in the code and trying to find reasons why certain code sections aren't covered continues to steal cycles from development teams.

RESOLVE COVERAGE GAPS

Teams can resolve coverage gaps in test suites using a coverage advisor. Parasoft discovered how to use advanced static code analysis (data and control flow analysis) to find values for the input parameters required to execute specific lines of uncovered code.

This analysis computes preconditions for function parameters, global variables, and external function calls required to execute a specific line of code. The Coverage Advisor view presents a collection of solutions for the user-selected lines of code. Presented values are used for creating new unit test cases. The functionality boosts the productivity of developers working on unit test cases to improve code coverage.



Pre-conditions for executing line 13 in processor.cpp - process(int, Point *, int)

Solution #1

Required dependencies:

Function parameter(s): int x

Pre-conditions:

x > 100

Expected coverage:

6 lines (6, 8, 9, 10, 12, 13)

Solution #2

Required dependencies:

External function call(s): int calculateValue(int)

Function parameter(s): Point * point, int x

Pre-conditions:

x <= 100

point != 0

calculateValue(int) > point->x

Expected coverage:

5 lines (6, 8, 9, 12, 13)

Figure 6: Parasoft Coverage Advisor displays what input values, global variables, and external calls are needed for a test case to obtain the needed code coverage.

Each coverage solution includes the following.

» Required dependencies. Dependencies that need to be customized to cover the selected line. These may include function parameters, external function calls, global variables, local variables, and class members.

» Preconditions. Conditions that must be satisfied by the required dependencies to cover the selected line. Clicking a precondition navigates to the related code line.

» Expected coverage. Code lines that will be covered if all of the preconditions are satisfied.

MAKE DATA-DRIVEN DECISIONS FROM CENTRALIZED REPORTING & ANALYSIS

Any CI/CD pipeline produces a lot of data. Whether it's build information, test data results, static analysis reports, or code coverage information, it's too much to consume manually. Teams need tools to collate this data, analyze it, and report it in human readable form. Teams can then use this data to make better decisions and assign resources within the organization to gain the most benefit. Here are some examples.

» Automating bidirectional requirements traceability.

» Reducing the burden of compliance reporting.

» Tracking progress through intelligent reporting and dashboards.

AUTOMATE BIDIRECTIONAL REQUIREMENTS TRACEABILITY

Requirements traceability is defined as "the ability to describe and follow the life of a requirement, in both a forwards and backwards direction (i.e. from its origins, through its development and specification, to its test verification and validation, including subsequent deployment and use, and through periods of on-going refinement and iteration in any of these phases)."

Many requirements in automotive software are derived from safety analysis and risk management. The system must perform its intended functions, of course, but it must also mitigate risks to greatly reduce the possibility of injury. Moreover, in order to document and prove that these safety functions are implemented and tested fully and correctly, traceability is critical.

Maintaining traceability records on any sort of scale requires automation. This is particularly important in a CI/CD pipeline since manually maintained traceability would slow down each iteration. Application life cycle management tools include requirements management capabilities that are mature and tend to be the hub for traceability.

Integrated software testing tools can complete the verification and validation of requirements by providing an automated bidirectional traceability to the executable test case, which includes the pass or fail result and can also trace down to the source code that implements the requirement.

Parasoft integrates with market-leading requirements management and Agile planning systems like Intland codebeamer, Polarion from Siemens, Jama Connect, Atlassian Jira, and a few others.

As shown in the image below, each of Parasoft's test automation tools, C/C++test, Jtest, dotTEST, SOAtest, and Selenic, support the association of tests with work items defined in these systems. That includes requirements, stories in IT, defects, test runs, test cases, and more. Traceability is managed through a central reporting and analytics dashboard, Parasoft DTP.

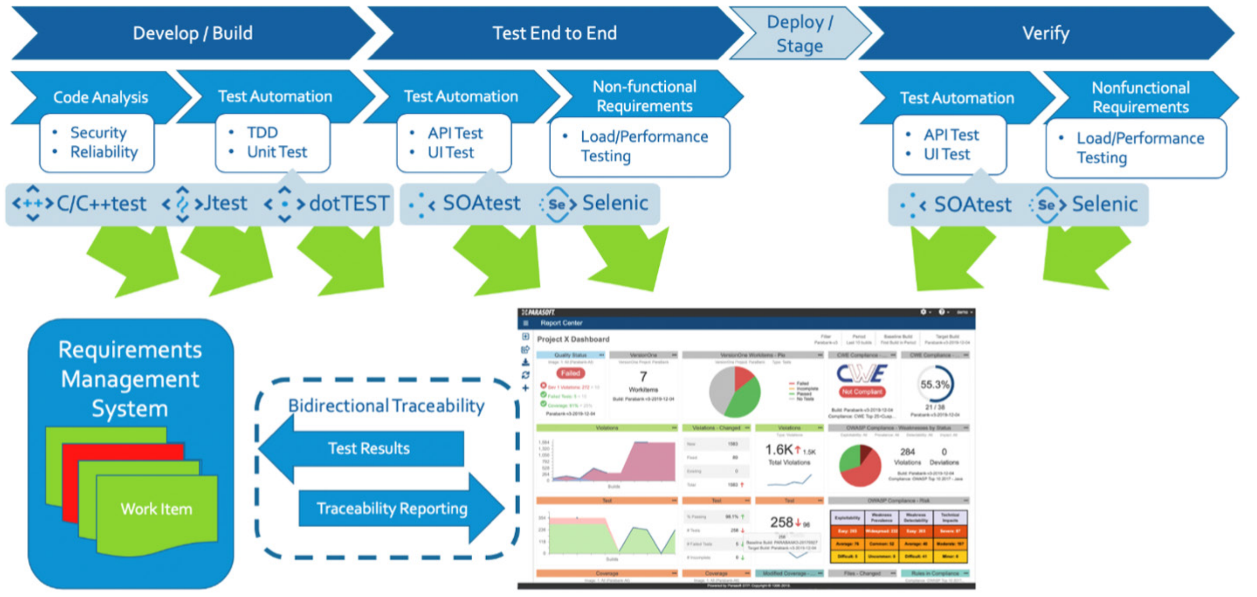
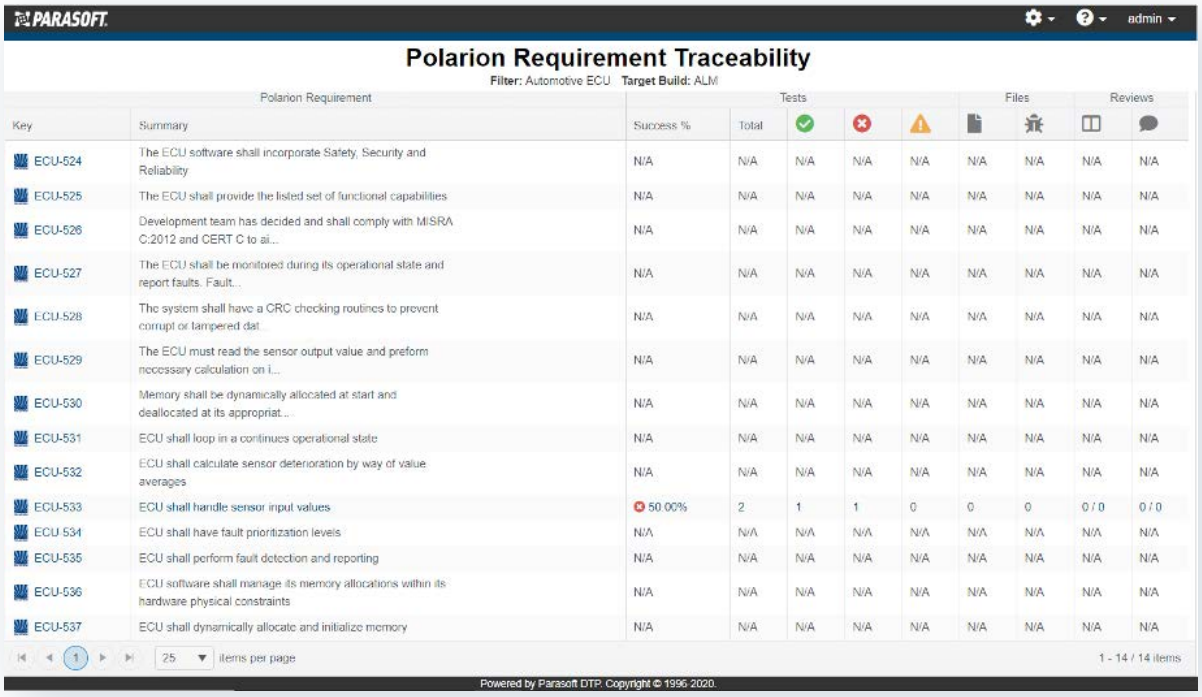


Figure 7: Bidirectional traceability from work items to test cases and test results. Traceability reports are displayed, and results are sent back to the requirements management system.

Parasoft DTP correlates the unique identifiers from the management system with static analysis findings, code coverage, and test results from unit, integration, and functional tests. Results are displayed within traceability reports and sent back to the requirements management system. They provide full bidirectional traceability and reporting as part of the system's traceability matrix.

The traceability reporting is highly customizable. The following image shows a requirements traceability matrix template that traces to the test cases, static analysis findings, source code files, and manual code reviews.



Polarion Requirement Traceability

Filter: Automotive ECU **Target Build:** ALM

Polarion Requirement		Tests					Files		Reviews	
Key	Summary	Success %	Total	✓	✗	⚠	📄	🐞	◫	💬
ECU-524	The ECU software shall incorporate Safety, Security and Reliability	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
ECU-525	The ECU shall provide the listed set of functional capabilities	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
ECU-526	Development team has decided and shall comply with MISRA C:2012 and CERT C to ai...	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
ECU-527	The ECU shall be monitored during its operational state and report faults. Fault...	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
ECU-528	The system shall have a CRC checking routines to prevent corrupt or tampered dat...	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
ECU-529	The ECU must read the sensor output value and preform necessary calculation on i...	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
ECU-530	Memory shall be dynamically allocated at start and deallocated at its appropriat...	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
ECU-531	ECU shall loop in a continues operational state	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
ECU-532	ECU shall calculate sensor deterioration by way of value averages	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
ECU-533	ECU shall handle sensor input values	✗ 50.00%	2	1	1	0	0	0	0 / 0	0 / 0
ECU 534	ECU shall have fault prioritization levels	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
ECU-535	ECU shall perform fault detection and reporting	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
ECU-536	ECU software shall manage its memory allocations within its hardware physical constraints	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
ECU-537	ECU shall dynamically allocate and initialize memory	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

25 items per page 1 - 14 / 14 items

Powered by Parasoft DTP. Copyright © 1996 2020.

Figure 8: Requirements traceability matrix template

The bidirectional correlation between test results and work items provides the basis of requirements traceability. Parasoft DTP adds test and code coverage analysis to evaluate test completeness. Maintaining this bidirectional correlation between requirements, tests, and the artifacts that implement them is an essential component of traceability.

Bidirectional traceability is important so that requirement management tools and other life cycle tools can correlate results and align them with requirements and associated work items.

REDUCE THE BURDEN OF SAFETY-CRITICAL STANDARDS COMPLIANCE

Compliance to standards can place a significant burden on the development team, depending on the criticality of the software being developed the amount of record keeping changes. For example, development for products that fall under ISO 26262 ASIL D, requires a thorough paper trail of best practices all the way through the SDLC. In support of this, Parasoft provides the following automation points, most of which have been discussed already, to assist in reducing the compliance burden.

» **Bidirectional traceability** is an important part of illustrating to regulators that requirements were fulfilled through design, implementation, verification, and validation. Automated traceability is essential to keep accurate records without tedious manual records.

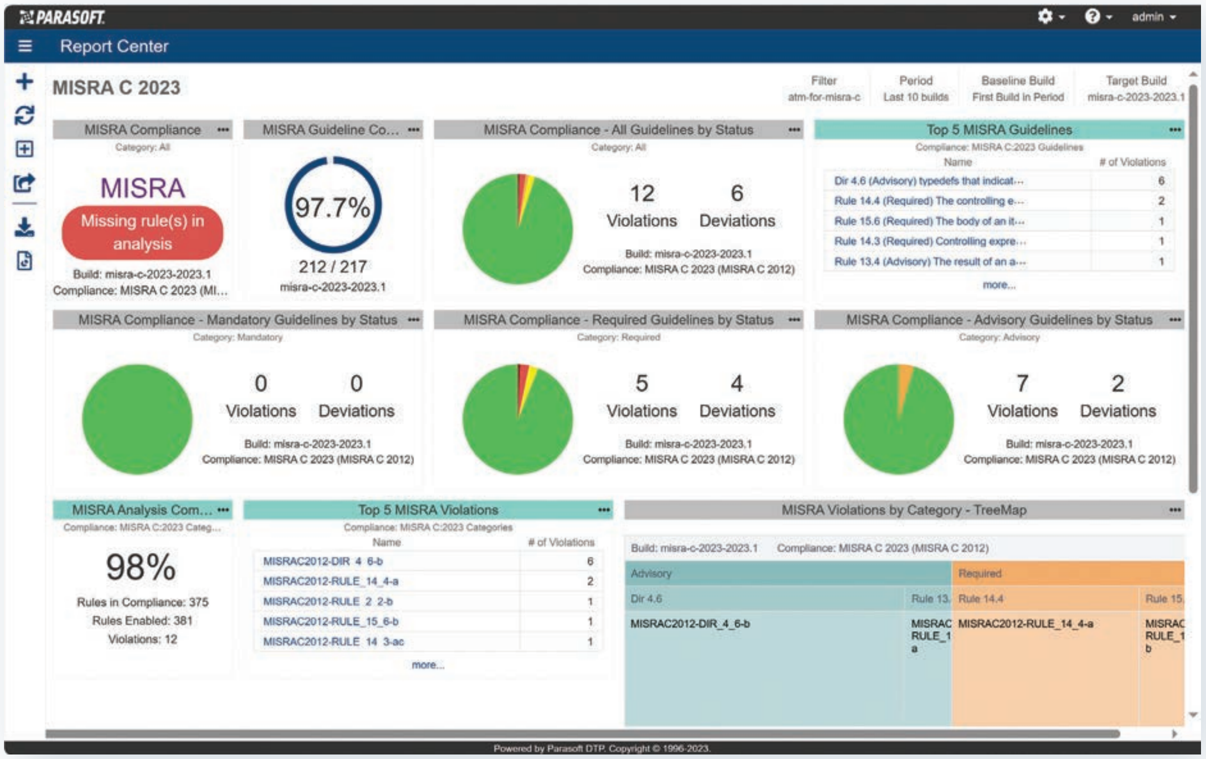
» **Coding standard compliance**, like with MISRA C/C++, is usually required for safety-critical automotive software. Without automated tools, such as static analysis, conformance to the standard would be impossible. An example of such compliance reports is shown below.

» **Code coverage** is critical in showing that tests have executed all the necessary parts of the application. Code coverage requirements for safety-critical software are strict. Automobile manufacturers need to keep records of their coverage reports. Automating code coverage reporting, especially across test types (unit, subsystem, manual, and system testing), provides a complete picture.

» **Validation and verification reports** from test execution is required to show that, indeed, tests have passed and been executed. These are often required to be performed on realistic target hardware so integration with embedded toolchains is essential.

» **Tool qualification** means that Parasoft tools meet the standards required to be used in safety-critical software. Parasoft C/++test, for example, has been certified by TÜV SÜD for ISO 26262, IEC 61508, IEC 62304, and EN 50128 standards. These certifications mean that the tools can be used in confidence for safety-critical software and that certification of the tools in a production environment is much easier.

The following is an example of a coding standard compliance report for MISRA C. Parasoft C/C++test provides dedicated reporting for documenting compliance to MISRA C. A dashboard on the Parasoft web portal provides at-a-glance views on the current state of the project, such as the one here:



Filter	Period	Baseline Build	Target Build
atm-for-misra-c	Last 10 builds	First Build in Period	misra-c-2023-2023.1

MISRA Compliance: Build: misra-c-2023-2023.1, Compliance: MISRA C 2023 (MI…

MISRA Guideline Co…: 97.7%, 212 / 217, misra-c-2023-2023.1

MISRA Compliance - All Guidelines by Status: 12 Violations, 6 Deviations. Build: misra-c-2023-2023.1, Compliance: MISRA C 2023 (MISRA C 2012)

Top 5 MISRA Guidelines — Compliance: MISRA C:2023 Guidelines

Name	# of Violations
Dir 4.6 (Advisory) typedefs that indicat…	6
Rule 14.4 (Required) The controlling e…	2
Rule 15.6 (Required) The body of an it…	1
Rule 14.3 (Required) Controlling expre…	1
Rule 13.4 (Advisory) The result of an a…	1

MISRA Compliance - Mandatory Guidelines by Status: 0 Violations, 0 Deviations. Build: misra-c-2023-2023.1, Compliance: MISRA C 2023 (MISRA C 2012)

MISRA Compliance - Required Guidelines by Status: 5 Violations, 4 Deviations. Build: misra-c-2023-2023.1, Compliance: MISRA C 2023 (MISRA C 2012)

MISRA Compliance - Advisory Guidelines by Status: 7 Violations, 2 Deviations. Build: misra-c-2023-2023.1, Compliance: MISRA C 2023 (MISRA C 2012)

MISRA Analysis Com… — Compliance: MISRA C:2023 Categ…: 98%, Rules in Compliance: 375, Rules Enabled: 381, Violations: 12

Top 5 MISRA Violations — Compliance: MISRA C:2023 Categories

Name	# of Violations
MISRAC2012-DIR 4 6-b	6
MISRAC2012-RULE_14_4-a	2
MISRAC2012-RULE 2 2-b	1
MISRAC2012-RULE_15_6-b	1
MISRAC2012-RULE 14 3-ac	1

MISRA Violations by Category - TreeMap: Build: misra-c-2023-2023.1, Compliance: MISRA C 2023 (MISRA C 2012)

Powered by Parasoft DTP. Copyright © 1996-2023.

Figure 9:
An example of a Parasoft DTP report for MISRA C coding compliance

Each of these dashboard widgets is linkable to a more detailed view, containing detailed violation reports, files, and source code.

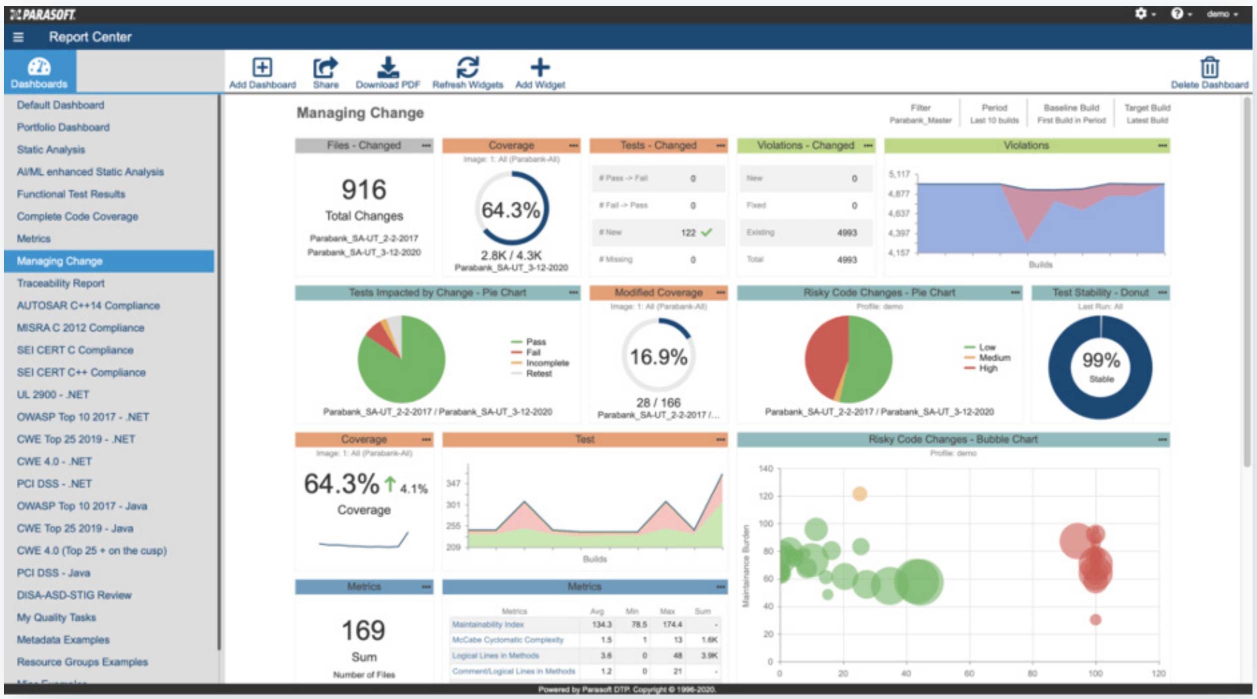
From here, you can automatically create the reports needed to document MISRA compliance as outlined in "MISRA Compliance 2020: Achieving Compliance with MISRA Coding Guidelines." Automating these reports is a big time saver, greatly reducing the amount of manual work required to document project compliance.

TRACK PROGRESS THROUGH INTELLIGENT REPORTING & DASHBOARDS

For team collaboration, Parasoft C/C++test publishes analysis results to DTP, a centralized server. Developers can access test results from automated runs and project managers can quickly assess the status of the project. Different dashboard views provide information of quality, code coverage, security, and coding standard conformance, for example. Reported results are stored with a build identifier for full traceability between the results and the build. Those results include:

- Static analysis findings
- Unit testing details
- Source code details
- Metric analysis details
- Code coverage details

When integrating into CI/CD workflows, Parasoft users benefit from a centralized and flexible web-based interface for browsing results. The dynamic web-based reporting dashboard includes customizable reporting widgets, source code navigation, advanced filtering, and advanced analytics from Parasoft's Process Intelligence Engine. Users can access historical data and trends, apply baselining and test impact analysis, and integrate with external systems like those for test requirements traceability.



Filter	Period	Baseline Build	Target Build
Parabank_Master	Last 10 builds	First Build in Period	Latest Build

# Pass -> Fail	0
# Fail -> Pass	0
# New	122 ✓
# Missing	0

New	0
Fixed	0
Existing	4993
Total	4993

Metrics	Avg	Min	Max	Sum
Maintainability Index	134.3	78.5	174.4	-
McCabe Cyclomatic Complexity	1.5	1	13	1.6K
Logical Lines in Methods	3.6	0	48	3.9K
Comment/Logical Lines in Methods	1.2	0	21	-

Figure 10: Centralized web-based dashboard for test impact analysis and more.

SUMMARY

DevOps in automotive software requires the ability to assess the risks associated with a release candidate—instantly and continuously. Continuous testing provides an automated, unobtrusive way to obtain immediate feedback on the quality, safety, and risks associated with a software release candidate. However, making this transition with embedded, safety-critical automotive software is not an easy task. Transitioning from a traditional waterfall method to an Agile one that drives a modern CI/CD pipeline is entirely feasible with the right preparation, training, and best practices.

These best practices include test automation to enable continuous integration. Another is to remove validation and verification roadblocks with continuous testing on host and target systems. Use of containers is a best practice that enables teams to create consistent, reproducible, and secure development environments. Next, grow test coverage with automated test generation and smart test execution. Lastly and importantly, use the feedback that the CI/CD pipeline provides to make data-driven decisions from centralized reporting and analysis.

The adoption of these modern, automated testing techniques and tools reduces the inefficiencies of traditional, manual automotive software testing—making new approaches attractive and shine.

TAKE THE NEXT STEP

Learn how your automotive DevOps software development team can implement best practices into your CI/CD workflow and streamline testing. Contact one of our experts today to request a demo.

ABOUT PARASOFT