PARASOFT

BUYER'S GUIDE

Static Code Analysis for Embedded Development

OVERVIEW

All static analysis tools tend to look the same from 50,000 feet. When planning to deploy static analysis, it is important to select a solution that fits the needs of the organization and can grow with future requirements.

The features and capabilities that a tool should have can be broken into two groups.

The first group is the common, expected technical features around items like supported languages, IDEs, CI/CD pipelines, industry safety and security standards, reporting, and the like.

The second group is the often-overlooked intangibles that can make or break a static analysis initiative and begs answers to the following questions.

- Does the tool come with support?
- Is it “static” itself or continually growing and evolving?
- Does the vendor work with customers and seem to care about their success?
- Will the tool fit into an organization’s software development lifecycle (SDLC) and development culture?
- When and where is it best to use free and open source software (FOSS)?
- When are commercial tools needed?

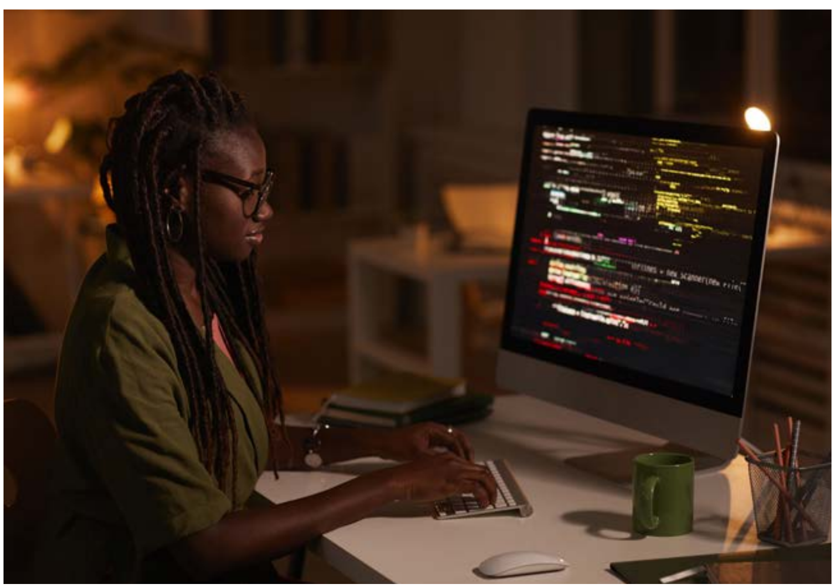
This paper provides a framework to use when evaluating static analysis tools for embedded software development that moves beyond simple proofs of concept, bakeoffs, and evaluations.

BACKGROUND

Software continues to increase in complexity while delivery timeframes continue to shrink. It's not uncommon today to have software that is released multiple times per day in support of complex, multi-platform, distributed systems that need to be safe, reliable, secure, and meet government and industry safety and security standards.

The Internet-of-things (IoT), for example, is made up of a surprisingly large amount of code in edge devices reliant on cloud-enabled services. IoT offers consumers and businesses useful technology and provides the building blocks for better factory automation, infrastructure and utility control, and the basis for autonomous driving.

The common strategy to meet this demand of better quality, in less time, with more security, leads organizations to static analysis tools to ensure that code meets uniform expectations around security, reliability, performance, and maintainability. When trying to determine which static analysis tool will work best, many evaluators take a common approach to selecting a tool for their group or organization: they run each tool on the same code, compare the results, then choose the tool that reports the most violations out-of-the-box.



This isn't really a product evaluation. It's a bakeoff. And the winner is not necessarily the best tool for establishing a sustainable, scalable static analysis process within the team or organization. In fact, many of the key factors that make the difference between successful static analysis adoption and yet another failed initiative are commonly overlooked during these bakeoffs.

This paper recommends the steps for selecting a static analysis tool that a software team will actually use. One that suits the team's current situation, can be deployed, and maintained across the organization, will assist in and survive safety and security certification, and will grow as needs evolve.

ASSESS YOUR NEEDS

Before searching for a tool that meets an organization's needs, your team needs to make a brutally honest assessment of where the organization stands today and where it hopes static analysis will take it.

CONSIDERATIONS

- What specific pain points are being addressed with static analysis? For example, is the improvement of code quality and reliability needed? Or, for example, is the goal to reduce the number of defects uncovered during QA and prevent release delays?
- Does the organization have regulatory compliance requirements such as functional safety standards or industry coding standards (FDA, MISRA, AUTOSAR, JSF, SEI CERT, CWE)?
- What initiatives are underway, such as security improvement, DevOps, DevSecOps, IoT, and so on. Does static analysis have a direct or indirect effect on these initiatives?
- Does the team need visibility into static analysis results and reports as it relates to risk management and/or compliance to industry standards?

WHERE YOU STAND

- Is the development process stable, repeatable, and streamlined enough to provide a strong foundation for static analysis? Are there weaknesses to address first such as lack of a fully automated build process?
- What does the existing pipeline look like? What is the build frequency—daily, hourly, continuous? Do tools in the pipeline need to run in the integrated development environment (IDE) on local servers and virtual machines (VMs) or in the cloud?
- Has static analysis been tried before? Was it successful? What was learned and what can be done to prevent the same obstacles to success this time?
- How is the development organization structured? Will there be a fixed set of quality policies organization wide and/or more specific checker configurations to suit the needs of specific projects and teams?
- How will static analysis efforts vary across current projects? What new projects are anticipated in the foreseeable future and how will static analysis apply?
- Where is the organization to be in terms of static analysis in two to three years from now? What about 10 years from now?

Gathering this information helps create a list of requirements which drive the evaluations of tools and vendors that best meet an organization's needs. Whether a formal request for proposal (RFP) is created or just an internal comparison, it's a good practice to establish these requirements ahead of time.

STATIC ANALYSIS OVERVIEW

In simple terms, static analysis is the process of examining source and binary code without execution, usually for the purposes of finding bugs or evaluating quality. Unlike dynamic analysis/unit testing (Parasoft's C/C++test & Insure++), which requires a running program to work, static analysis can be run on source without the need for an executable.

This means static analysis can be used on partially complete code, libraries, and third-party source code. Static analysis is accessible to the developer, to be used as code is being written or modified, or to be applied on any arbitrary code base. In the application security domain, static analysis goes by the term static application security testing (SAST). Many commercial tools support both security vulnerability detection alongside bug detection, quality metrics and coding standard conformance.

Static analysis tools are mandated or highly recommended by safety standards such as ISO 26262, DO-178B/C, IEC 62304, IEC 61508, and EN 50128, for their ability to detect hard-to-find defects and improve security of software. Static analysis tools also help software teams conform to coding standards such as MISRA, AUTOSAR, or SEI CERT.

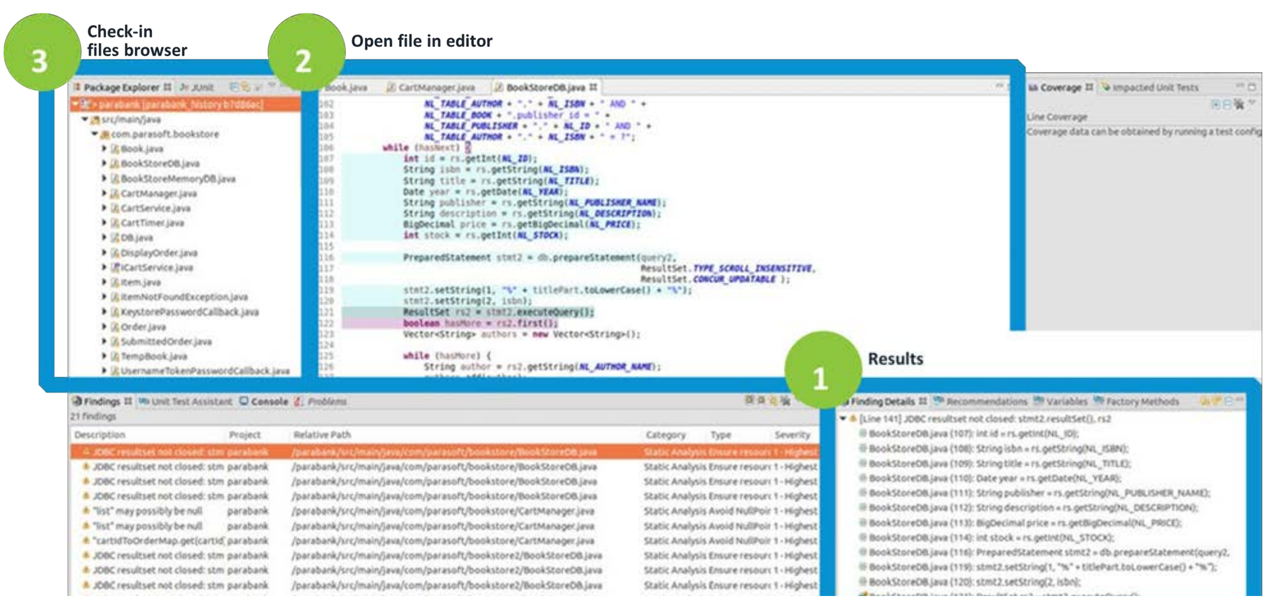


Figure 1:
An example of integrating static analysis into a developer's IDE:
1) Warning delivered directly into error windows.
2) Code highlighting and tracingwhich quik to line of code based on warning selected.
3) Support for project view and code check in.

To learn more about how static analysis works, read our whitepaper Getting Started With Static Analysis.

COMMON CAPABILITIES

Static analysis tools have matured in the last decade. Below is a list of expected capabilities that advanced modern static analysis solutions have, from configuration, customization and integration through compliance-oriented reporting and analytics. It's important to understand what value each of the below capabilities provides, decide which ones apply, and their respective priority.

Configuration	Integration	Ease of Use	Reporting & Analytics	Standards & Compliance
☐ Centralized configuration	☐ Desktop & server scanning	☐ Integrated clickable docs	☐ Configurable dashboards & reports	☐ Built-in support for common security standards
☐ Custom checkers	☐ CI/CD plugins	☐ Right/wrong code examples for each checker	☐ Custom widgets	☐ Built-in support for common safety standards
☐ Support for inline and external suppressions	☐ Roundtrip results from CI/CD to IDE	☐ Online training links	☐ Custom data sources	☐ "Mapless" standards-centric configuration & reporting
☐ Flexible configuration controls and permissions	☐ IDE plugins	☐ On-the-fly IDE analysis	☐ Support for security risk models	☐ Supports multiple models of checkers (prevent,smells,detect)
☐ Scan projects with millions of lines of code	☐ Web-based UI	☐ Automated violation assignment	☐ Code author information	☐ Complete comprehensive line-item support for compliance & security standards
☐ Configuration supports legacy code/age	☐ CLI for automation	☐ Built-in configuration for common standards	☐ Built-in history & analytics	☐ Common industry metrics with thresholds
☐ Configurable checker severity levels	☐ Open APIs for integration	☐ IDE quick fix	☐ Custom analytics	☐ Dead code detection
☐ Flexible licensing models	☐ Source control integration		☐ Simple PDF report export	☐ Duplicate code detection
☐ Parameterized checkers	☐ Bug tracking integration		☐ Open output API	
☐ Supports dynamic CI/cloud deployment	☐ Requirements management integration			

Figure 2:
Static analysis tool evaluation criteria

Configuration is an often overlooked aspect of static code analysis. It's important that a tool can be set up to consider a project's required standards, risk model, and associated legacy code as well as fit into reasonable schedules and workflow.

> Getting the configuration right saves trouble down the road. Getting it wrong almost always means long-term failure.

For example, if your team is complaining about false positives, they probably already started off on the wrong foot with improper configuration.

Integration is important so that the tool fits into the existing workflow, pipeline, process, and toolchain. A tool that doesn't play well with others is best avoided. Integration is important both in the build toolchain as well as into the developer's desktop tools and IDEs.

Ease of use is more important than first realized. It's not only about how easy it is to set up and learn a tool, but also what it takes to work with the output of static analysis on a day-to-day basis. Ultimately, the sustainability of a static analysis initiative is dependent on how well it can seamlessly work with the people who actually write the code and the people who manage them.

Reporting and analytics are a critical part of static analysis, helping to understand where risk lies in the code, which warnings are most important, and which warnings can be safely ignored. Reporting and analytics help the business to understand trends (improvement over time) and status (will the project deliver on time?) and even return on investment (is my static analysis tool saving me time? Money? Bugs?)

Most tools have basic reports like histograms, a list of violation by severity and category. It's important to also have risk scores, prioritization models, and flexible report output that fits your organization's reporting needs.

Standards and compliance are often key drivers for static analysis. Many standards require general use of static analysis. Other standards lay out general principles and some spell out exactly what must be done. An effective solution supports the standards required without the tedious mapping of tool checkers to standard guidelines and provides reports that support audit requirements and clearly illustrate exactly what was done and how.

Modern tools should support an entire standard, not some fraction of it. In addition, automation of the documentation and reports needed to demonstrate compliance is a critical feature.

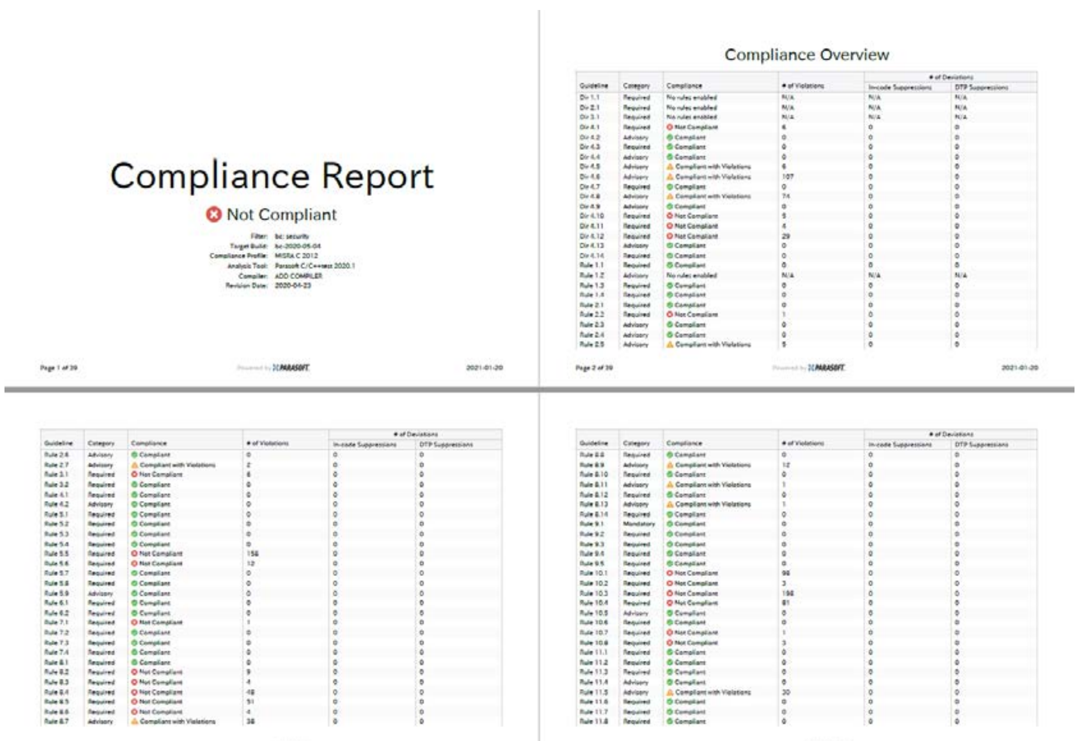


Figure 3:
Static analysis compliance report

MORE STATIC ANALYSIS TOOL FACTORS

There are other key aspects of static analysis tools that need to be considered depending on the scale of usage and intended project environment. These factors should also be considered during the evaluation depending on needs.

» **Scalability** determines how well a tool scales to projects large and small. Here are some questions to ask:

  » Is the tool able to handle extremely large amounts of code?

  » Is desktop and server-based usage supported?

  » How will the tool impact a continuous integration/deployment pipeline?

  » Does the tool work with our embedded operating system development tools and platform?

» **Flexibility** of tools is important for integrating any tool in day-to-day workflows and pipelines. It's also a key factor in how the tool is being used. If the focus is on security, for example, can the tool be configured easily across the organization to focus on security vulnerabilities and standards? Or it may mean customizing the tools to support in-house coding standards, guidelines, and checkers.

» **Centralized and distributed** sounds contradictory but it relates to the ability to support remote operation on a developer's desktop and simultaneously supporting centralized analysis on the complete project. Centralized collection of results, analysis and reporting is important for management and project status evaluation. A modern static analysis tool needs to support both of these key environments.

» **Managing tool output (findings, warnings, bugs, vulnerabilities).** All static analysis tools create lists of warnings. What separates them is how well they manage these results. Once a static analysis tool has been installed and configured in a project and all dependency issues sorted out, there is usually a lengthy report of violations and warnings reported by the tool. This can be overwhelming.

  How these initial reports are managed influences the success of the tool integration into the project. Not all warnings are critical and don't need to be dealt with immediately. The tool must support management of results, workflows for bug tracking, integration with developer tasks, and automated prioritization rather than manual triage. Tools must also be able to consider issues with legacy code and varying policy.

- **Industry risk models.** Support for risk profiles is a good way to prioritize static analysis findings. Those that are in the high-risk category should receive the highest priority and those that are low risk get low priority. SEI CERT categorizes risks into three levels based on anticipated severity and cost to repair: low, medium, and high.

  CWE has categories around the impact of the particular vulnerability based on its context. Make sure that your tool supports these risk scoring models without manual effort.

- **Artificial intelligence (AI) and machine learning (ML) tools** leap forward productivity and adoption of static analysis. AI solutions available can review new static analysis findings in the context of both historical interactions with the codebase and prior static analysis findings to predict relevance and prioritize the new findings.

- **Configuration and filtering** modern static analysis tools should provide the ability to configure the set of checkers enabled for the analysis and provide the ability to filter out results within the respective reporting tool's warnings based on warning category, file name, severity, and other attributes.

  Both methods are available to help developers focus on the types of warnings that they are interested in and reduce the amount of information provided at any one time. Shockingly, some tools have little to no capability in this area, requiring you to run their predetermined set of checkers, which likely don't align with your business needs and risk.

Appendix A provides more details about each evaluation criteria.

Learn more tips and training for a successful static analysis deployment in the whitepaper, Getting Started With Static Analysis.

INTANGIBLES

Succeeding with static analysis is more than just a feature checklist. There are several intangibles that can make or break the initiative, including:

- Is the tool scalable?
- Does the vendor keep up with current standards as they evolve?
- Does the vendor provide support, training, documentation, and generally work well with their customers?

The selection process below lays out how to incorporate these important nonfunctional requirements into the evaluation effort.

TOOL SELECTION PROCESS

COMPILE A PRELIMINARY LIST OF NEEDS AND CRITERIA

The first step is to explore the available options and compile a preliminary list of tools that seem like strong contenders. What are the criteria to consider?

CONSIDER—BUT DON'T BLINDLY ACCEPT—RECOMMENDATIONS

When word gets around that an organization or team is investigating new tools, they are likely to hear some suggestions. For instance, someone may recommend tool A because it was used on a previous project. Maybe a star developer has been using tool B on his own code and thinks everyone else should use it, too.

These endorsements are great leads on tools to investigate. However, don't make the mistake of thinking that a strong recommendation—even from a trusted source—is an excuse to skip the evaluation process.

The problem with these recommendations is that the person offering them probably had a different set of requirements than exists now. They know that the tool worked well in one context. However, the current need is to select a tool that works well in the current environment and helps accomplish departmental and organizational goals. To accomplish this, it's important to keep the big picture in sight during a comprehensive evaluation.

EXPLORE VENDORS

When an organization acquires a tool, they are committing to a relationship with the vendor of choice. Behind most successful tool deployments, there's a vendor dedicated to helping the organization achieve business objectives, address the challenges that surface, and drive adoption.

It's important to consider several layers of vendor qualification and assessment across the span of the evaluation process. At this early stage, start a preliminary investigation by getting a sense for what the vendor thinks of their own tool by reading whitepapers, watching webinars, and more. Focus on the big picture, not the fine granularity details.

Points to Consider

- **Vision.** If the vendor's vision is not aligned with requirements and goals, or if the vendor isn't poised to support anticipated growth, it's best to learn this early in the process. It's inadvisable to evaluate a vendor who is misaligned with an organization's goals unless options are extremely limited.
- **Best practices.** Learn about the vendor's recommended best practices for using their tool. Ask these questions:
  - Do they have a coherent strategy for how to deploy the tool across the organization?
  - Will they evolve the tool as the organization's needs change?
  - Does the strategy align with the team and organization's goals?

Remember that if developers don't end up using the tool daily, it's not going to deliver value to the organization—no matter the rich functionality the tool offers. The lack of apparent best practice doesn't mean a tool is ruled out (although a possible red flag.) However, a usage model needs to be developed, which makes the evaluation and actual deployment significantly more complicated.

- **Reputation.** Research the vendor and find out the following:
  - What organizations are using the tool?
  - What do the case studies reveal about its deployment, usage, and benefits?
  - What are industry experts saying in reviews, writeups, and awards?

EVALUATE VENDORS

The next step is to contact the vendors. Full tool evaluations are potentially time consuming and disruptive, so research is recommended before ever installing a tool on a developer desktop. You can get answers to many key questions just talking to the vendor. Consider the topics below during discussions with tool vendors. See Appendix B for more details about evaluating vendors.

Free and Open Source Solutions (Foss)

An obvious question arises about the use of open source tools for a static analysis solution. There are few issues with FOSS to keep in mind.

Open source software is often described as “free like a puppy, not free like beer” meaning that costs are incurred regardless of the free license. Looking at FOSS solutions is not discouraged, but an evaluation needs to include costs for important features, services, and support that are lacking. Details about costs and benefits of FOSS in general are available elsewhere, including issues like:

- Is support available? Will I need it?
- Is the project active? If not, do I want to effectively take it over?
- Is it good enough to solve the problems I need it to solve?
- If I’m working with a standard, how much is covered by the tool?
- Will it scale well in an embedded environment? Often tools that work well for small groups struggle in large organizations.
- Is tool qualification needed for safety or security certification? Using open source tools in safety critical software development may incur the added responsibility and cost of certifying the tools to be fit for purpose.

One thing to keep in mind about FOSS static analysis tools is that studies by organizations such as NIST have shown them lacking. As of writing this paper, FOSS static analysis tools, although generally easy to use with relatively good performance, are not as thorough or as complete as the commercial solutions in terms of precision, coverage of coding standards, and set of comprehensive warning classes.

When working with standards such as CERT, AUTOSAR, CWE Top 25, and MISRA C/C++, investigate specifically what items in the standard are covered by the tool. Currently, FOSS tools have poor coverage for any of the well-known industry safety and security standards.

Evaluation Criteria

Below are criteria to consider during the technical evaluation of the candidate tools. These are expanded upon in Appendix A.

- Coverage of the necessary industry standards.
- Quality of the built-in checkers for the necessary guidelines.
- Depth and breadth of analysis.
- Practical means to reduce “noise” (ignorable checker violations).
- Reasonable number of and approach to false positives.
- Acceptable number of false negatives.
- Ease of adjusting built-in checkers to suit organization’s policies.
- Ease of adding new custom checkers to check unique requirements.
- Level of complexity supported for new custom checkers.

EVALUATING PILOT PROJECT RESULTS

When evaluating the results of each pilot project, the evaluation and final decision making should boil down to answering these three important questions.

Will the team really adopt it and use it?

The best tool in the world won’t deliver any value if it’s not deployable, developers won’t use it, or it’s too much of a disruption to the project progress. Deciding how smoothly something can be adopted requires a comprehensive evaluation of the tools, integrations, and the vendor's support, services, and training.

Some factors that affect adoption include:

- A robust and flexible checker configuration.
- Reducing “noise” in the results.
- A workflow that’s practical and repeatable for both your highly skilled engineers and junior developers.
- Scalability beyond the current project and across the organization.
- A vendor committed to working with your organization to achieve success.

The combination of all these factors work together to make the difference between a good tool and a great tool *for an organization*.

Often, developer adoption boils down to whether they recognize the time saved in the long run. That includes the perception of extra work required, which, at minimum, is reviewing and responding to reported violations. For instance, if the tool identifies the root cause of issues that have been troubling them—or alerts them to issues that they know will cause headaches later—they are much more likely to embrace it as a help rather than reject it as a hindrance.

Will it address the problems the organization and team are trying to solve?

Deployment of new technologies requires a focus on what problems are trying to be solved. Additionally, the expectations of the new technology to address the problem should be realistic. If you simply assume that static analysis will improve whatever software issues your team is experiencing, then you should expect to be disappointed.

An example of how a trial or evaluation can fall apart is when an organization rushes to solve a pervasive problem, turns on all the checkers (beyond typical default settings) in the static analysis tool, gets overwhelmed with warnings, and fails to solve the original problem.

This is either a mismatch between expectations of what static analysis tools can do or lack of understanding about how these tools should be introduced into a project.

It's also Important to quantify success and ROI. It's important to determine ahead of time how success is measured: lost time, missed releases, or field support cases. The ROI you get should be measured by addressing the problems for which you chose static analysis.

One common trap to avoid is the idea to assess value based on how many violations static analysis finds. Any well-structured deployment of static analysis will have more violations initially than later as the code comes into compliance. This doesn't mean the tool is less valuable. In fact, the less findings against the same checkers is indeed proof that the tool is doing its job. It's not just finding bugs. It's changing developer behavior by getting them to write better code.

Is this a long-term solution?

Evaluations are time consuming and require team commitment. Full deployments require more time and commitment. Settling for a tool that's "good enough for now" might save money in the short term but prove extremely costly in the long term.

Every software development organization needs to grow and evolve to remain viable today. It's not a question of if, but how. Whether the organization is trying to advance quality by adopting additional software verification methods, complying with evolving corporate governance policies, or extending into new types of development projects, tool requirements will change.

The ultimate question when evaluating tools is: Will this tool and vendor in the long run help reach the project, organization, and company goals, or hold them back?

Establishing a workable and sustainable quality process takes time. Starting this path early prepares the organization for the pressure of delivering software at a faster pace or improving quality. Procrastination results in efforts being too little, too late.

SUMMARY

Evaluating software tools for adoption and integration into a company's software development process is a time consuming yet important practice. It's critical that organizations have a clear understanding of their goal and motivation behind it when adopting any new tool, process, or technology. Without an end goal, success is indeterminable.

Static analysis tool evaluations often end up as a "bake off" where each tool is tested on a common piece of code and evaluated on the results. Although this is useful, it shouldn't be the only criteria used. Technical evaluation is important, of course, but evaluators need to look beyond these results to the bigger picture and longer timeline.

> Evaluators need to consider how well tools manage results including easy-to-use visualization and reporting.

Teams also need to clearly understand how each tool supports claims made in areas like coding standards, for example.

The tools that vendors use themselves need to be part of the evaluation. A vendor who becomes a partner in your success for the long haul is better than one that can't provide the support, customization, and training the team requires.

Most important of all is how each tool answers these three key questions:

- Is the team going to use the tool?
- Is the tool the solution that will help the organization reach its goals?
- Is the tool a long-term solution to problems that the team faces?

TAKE THE NEXT STEP

Learn how static analysis solutions for embedded software development can streamline your testing process. Contact one of our experts today to request a demo.

ABOUT PARASOFT

Parasoft helps organizations continuously deliver high-quality software with its AI-powered software testing platform and automated test solutions. Supporting the embedded, enterprise, and IoT markets, Parasoft's proven technologies reduce the time, effort, and cost of delivering secure, reliable, and compliant software by integrating everything from deep code analysis and unit testing to web UI and API testing, plus service virtualization and complete code coverage, into the delivery pipeline. Bringing all this together, Parasoft's award-winning reporting and analytics dashboard provides a centralized view of quality, enabling organizations to deliver with confidence and succeed in today's most strategic ecosystems and development initiatives—security, safety-critical, Agile, DevOps, and continuous testing.

Appendix A: Tool Evaluation Capabilities & Criteria

TECHNICAL EVALUATION CRITERIA

Coverage of the needed checkers. The evaluation should focus on the checkers that the team and organization are willing to enforce—both now and in the foreseeable future. Enforcement may mean stopping the release or deployment of an application that has violations of a particular checker.

Quality of the built-in checkers for necessary industry standards and guidelines. Evaluate each tool's checker accuracy for the guidelines to be enforced. Although many checkers initially appear useful, the tool under evaluation may report so many false positives (incorrect warnings) that this guideline and checker combination is not useful. The lack of checker precision may be a result of poor implementation, or it could be ill-suited for verification by static analysis. Other verification techniques may work better. In terms of the tool evaluation, the existence of a checker to support the needed guidelines isn't enough by itself. Precision matters.

Coverage of the needed industry and corporate standards. Evaluate each tool on the vendor's support for the common industry standards like CERT, AUTOSAR, CWE Top 25, SEI CERT C, and MISRA C/C++. Even if one of these standards doesn't apply now, it could in the future. Also consider support for compliance to functional safety standards like ISO 26262, ISO 61508, ISO 62304, EN 50128, and others like DO-178B/C. Be sure to investigate how deep the support is for each standard. Evaluate each tool on how well it supports audits required by these standards and the vendor's experience in each of these areas.

Depth and breadth of analysis. Evaluate each tool on depth of analysis such as support for advanced control and data flow analysis for improved results in finding critical bugs and security vulnerabilities. Also evaluate each tool on its breadth of analysis such as support for so-called "code smells", industry and de facto coding standards and guidelines, and proactive checkers that prevent bugs from occurring in the future. An equally important criteria is the scope of the analysis. Ideally, it should be the entire program.

Practical means to reduce noise (ignorable warnings). The more noise is reported, the more likely team members are to ignore all warnings, including important ones. Reducing noisy reports can be accomplished by disabling checkers, modifying checker parameters, suppressing checkers in specific contexts. Tools that produce too much noise might increase the burden of the tools on the development team. It also impacts the CI/CD pipelines that rely on automation to provide go/no-go build and deploy decisions with minimal human review.

Reasonable number of false positives. There are broad interpretations of false positives, which, by definition, mean warnings reported are incorrect and don't violate the guideline being checked. These also include the following:

» Correct warnings for checkers that developers don't like or may disagree with.

» Misunderstood checkers.

» A real error that has a mitigating circumstance missed by the analysis.

» Checkers that are ignored in certain contexts such as in legacy code.

Regardless, false positives, whether meeting the strict definition or not, are the most likely reason for users to dislike using static analysis tools.

To improve the perception of the tools, it's important to understand the root cause of false positives. Verifiable incorrect warnings can often be traced to incomplete analysis due to missing dependencies. Like a compiler, static analysis tools require the full context of dependencies to perform precise analysis.

Other issues such as checkers that the team doesn't agree with, should simply be turned off. Tools should be evaluated on how they can handle both "real" false positives and usability issues with the warnings produced. Configuration options, for example, go a long way in improving tool output.

Acceptable number of false negatives. False negatives are instances where code violates a checker, but the tool misses it, and no warning is reported. With all static analysis tools there is a trade off between producing a low number of false positives and missing real bugs and security vulnerabilities, the false negatives. There is balance needed between the number of false negatives and false positives since missing real bugs is a concern. Each tool should be evaluated on more than false positive rate alone. Missing important warnings is of equal concern.

Ease of adjusting built-in checkers to suit team and organization policies. Each tool should be evaluated on how simple it is to adjust checkers to suit team and organizational requirements. Also consider if the checker modifications can be done without scripting or complicated configuration.

Ease of adding new custom checkers. Evaluations should include modifying checkers and creating completely new checkers (or ones based on existing checkers) via scripting or other provided techniques such as APIs. Evaluate the complexity of creating new checkers and how well it's supported by each tool. Does the tool provide a UI for creation and customization? If a complex process is required or an API, how well suited is that to the team's needs? If consulting or professional services are required, be sure to include the estimated cost.

TOOL SCALABILITY CRITERIA

Scalable usage model. Scaling to current and future requirements is a key criterion for tool evaluation. Not all static analysis tools are designed for large scale deployment and analysis. Consider whether the vendor's proposed usage model (in terms of deployment, updating, and training) scale to current requirements and the future. Does the product licensing model work with the organization's goals?

Ease of updating the tool configuration across the entire team or organization. Adopting static analysis organization-wide requires the ability to deploy the tool equally to each developer. Evaluate the tool and the vendor's process for deploying and updating the tool configuration across all applicable tool installations.

» Is there a way to guarantee that everyone is using the correct configuration?

» Is there a role based access control to ensure that only the appropriate people, like team leads, modify the checkers and configurations?

» Can the deployment of the tool support an audit when developing safety-critical software, for example?

Ability to support tiered configurations. Each tool should be able to enforce a fixed set of quality policies organization-wide, but still be able to support customization to suit the needs of specific projects and teams.

Extensibility. Each tool should be evaluated on how well it supports customizations.

» Is there an API or scripting support? If so, is the API well documented?

» Are there ways to automate and integrate through programming APIs, CLIs, and REST APIs?

Support for target operating systems, tools, and other languages and verification methods.

» How well can each tool be extended to support other best practices such as peer code review support, unit, integration, and system testing?

» Does the tool support all the programming languages that the organization requires?

» Does the tool work with the target tool chain such as cross compilers, embedded operating systems, board support packages, and system libraries?

Speed of analysis. For large code bases, the speed of analysis becomes an important factor in tool evaluation. Consider whether there is a significant discrepancy in the desktop analysis speed between the different tools. Does the tool support different modes of analysis such as fast checkers on the desktop and more in-depth analysis in batch mode?

Be sure to measure speed in terms of the end-to-end process. Consider whether developers need to open another tool, run it, then bring results back into their original environment. For automated/build execution, speed is mostly a factor that the analysis completes within the allotted timeframe. Consider whether the analysis requires additional servers and the cost therein.

Cloud deployable. If applicable to your development ecosystem, does each tool integrate with cloud services like AWS, Microsoft Azure, and others to run the analysis? Is it possible to set up servers in a private cloud?

CONFIGURATION EVALUATION CRITERIA

Centralized configuration. Tools under evaluation should support configuration that can be set by team leads and distributed to developers on the team to support a common set of guidelines and standards to follow. Local configurations can add to this but shouldn't contradict the project-wide settings. Tools should support grouping and categorization of settings for different purposes such as new code versus legacy code. Warning severity should be customizable both at configuration time and in warning reports.

Custom checkers. Customizing checkers should be supported as well as the ability to distribute these custom checkers to the rest of the team easily and automatically. Creating new checkers should be straightforward if based on existing checkers. An API should be available for more sophisticated customization.

Support for inline and external suppressions. Warnings need to be suppressed in the right circumstances and developers should have the flexibility to deal with this directly in the code with an inline expression or via the tool either in the IDE or via a web interface at the project level.

INTEGRATION EVALUATION CRITERIA

IDE integrations. Evaluate how each tool supports the team's development environment.

» If not supported, what is the path to support?

» Does the integration meet the required usage for day to day workflows?

Cross compilation, build mode, target operating systems support.

» Does the candidate tool support command line operation?

» Can the analysis be invoked in a batch mode?

» How are results from batch mode handled?

» Does the tool work with the target operating system and tool chain?

» Can it easily be integrated into a complex embedded system build environment?

CI/CD pipelines.

» Does the tool work in your existing toolchain?

» Can it be used as a gate for making decisions to promote or not promote your code in a true continuous environment?

» Does it work well in a distributed execution environment?

Warning reporting/review mechanisms. Evaluate each tool on how easy it is to understand warnings and the reports generated.

» Are they extensible/customizable, if needed?

» Do the reports show historical information and trends on a time or build-by-build basis, or are they a snapshot in time?

» Are there additional analytics like alerts for areas of concern, coding standard compliance, and guidance on next steps?

Connection to bug tracking. Evaluate the tools on their integrations to other critical systems in the development environment. Bug tracking is a common integration with static analysis since warnings can be real bugs that need to be tracked and fixed. For example, does the tool support integration to JIRA?

Connection to requirement management tools. Certain requirements may need tracking into static analysis, for example, nonfunctional requirements for security or adherence to standards.

Automated assignment of errors to responsible developers. Candidate tools are evaluated on how warnings are managed.

» Are issues detected by batch mode tests assignable to the developer who wrote the related code?

» Is it possible to distribute the information to their desktop with direct links to the problematic code?

» Can violations be reassigned if needed?

» Can the violations assigned to one developer be mapped to another when someone leaves the group?

Legacy code identification and support. Tools should be able to deal with legacy code, using different configurations for new, existing, and legacy code. Consider whether each tool can apply a configuration unique to each category of code. Can it identify and ignore all legacy code if needed?

Checker severity customization. Evaluate whether each tool can change warning severity levels to help the team focus on the most important error types.

Ability to suppress warnings. Evaluate how well each tool supports suppression of warnings.

» Can a checker be enforced in general but be exempt in certain instances?

» Are suppressions shared across the team?

» Can they be defined in the code so everyone working on or reviewing the code can see them?

» If warnings are suppressed when developing with standards such as MISRA and CERT C, is there a mechanism to document them as deviations?

Automated violation correction. Can the tool refactor code to fix any of the violations you care about? If 100 checkers matter to you and tool A can fix 50 of them while tool B can fix none of them, that's a huge benefit for tool A.

On-the-fly analysis. Evaluate whether tools can analyze the code on demand inside the IDE before it's even checked into source control.

» How are these results handled?

» If a warning remains in the code after check-in, does this result show up in the batch/ build analysis?

Risk models. Does the tool under evaluation help prioritize warnings by risk profile? Does the tool support common risk models such as CERT? Are these risk models configurable?

EASE-OF-USE EVALUATION CRITERIA

Integrated and navigable documentation. Evaluating each product's documentation is an important part of the evaluation.

» Is the documentation easily accessible?

» Is it easy to navigate? Is the documentation available right in the IDE?

» Is each warning properly documented?

» When a warning is issued, is it easy to find the documentation for it?

Documentation should contain code examples for each error. For coding guidelines and checkers, examples that do and do not violate the checker should be illustrated.

Online training. Training is important for adopting any tool. Evaluating a vendor's training capability is important and so is the accessibility of training after initial deployment. Online, in person, and video based training should be available.

Tool usability. Ease of use should encompass all aspects of the tool's usage.

» Is it easy to use at the developer level in the IDE?

» Is it easy to assess the warning reports? Is the web interface easy to navigate?

» Does the tool integrate into daily workflows with little impact on developer productivity?

» How easy is customization?

» Are developers picking up tool usage easily?

There are many aspects of usability, but in general, users will provide feedback on their experience.

REPORTING AND ANALYTICS EVALUATION CRITERIA

Configurable dashboards and reports. Reports and dashboards are useful for condensing large amounts of data into an easy-to-understand format. Tools should be evaluated on the quality and configurability of their reporting.

- Are dashboards provided?
- How does the tool support high-level management of results?
- Are dashboard widgets configurable?
- Are data sources customizable?
- Are reports linkable to other activities such as unit tests, API, and UI tests?

Support for risk models. Are results reported in relation to industry standard risk models? For example, SEI CERT coding standards include a risk model and violations can be mapped to this model, which helps with evaluation and prioritization.

Warning history and analytics. Tools should support historical information for warnings and, preferably, analytics that provide further insight into trends.

- Can warnings be traced to a particular build or file modification?
- Is it possible to see the life of a warning over time?
- Are trends visible in the dashboard?
- Are these analytics configurable?

Report output. Tools should support reports that can be printed or used in an official manner as a record for milestones.

- Does the tool support PDF report export?
- Is there an open API for custom output options?

STANDARDS AND COMPLIANCE EVALUATION CRITERIA

Built-in support for common security standards. If one of the goals for static analysis adoption is improving security or adopting a secure coding standard, then it's reasonable to expect that the tool is being evaluated to support common standards. For example, does the tool support CWE/SANS Top 25, CERT secure coding standards?

It’s also important to determine how much coverage each tool has of each standard for which support is claimed. For example, sometimes vendors have MISRA C configuration that only covers a subset of the guidelines.

Built-in support for common safety standards. Similarly, if the intended use of the static analysis tool is on a safety-critical project, it’s reasonable to expect support for common standards.

- Does the tool under evaluation support MISRA C and MISRA C++, if required?
- Does the tool support AUTOSAR C++14, if necessary?
- What coverage of these standards does each support?
- How is compliance, reporting, and checker violation handled?

Mapless violation reporting and configuration. A common way to “support” common standards in static analysis tools is to map existing checkers into each standard. Developers must refer to this mapping to determine which checker is being violated by each warning.

This extra mapping layer increased the tedium of enforcing and compliance with standards. During tool evaluation, it’s important that the evaluation considers how easy it is to relate warnings with the standards needed and how easy each tool is to configure.

Supports multiple modes of checkers. During the evaluation, some vendors may tout the error detection capability but it's important to consider preventative methods as well.

- » Does each tool under evaluation do "code smell" detection?
- » Are their checkers designed to detect poor software coding techniques ahead of time?
- » How well is the defect and security vulnerability detection complimented by preventative checkers and coding standard support?

Common industry metrics with thresholds. Static analysis tools are ideal for collecting software metrics during their analysis. In fact, common metrics such as cyclomatic complexity may be collected by default. If metrics are important to the organization, then the evaluation should consider how well each tool supports metrics.

- » Are the metrics included in reports and dashboards?
- » Can thresholds be set for each metric?
- » Does exceeding metrics threshold raise a warning?
- » How easy is it to create new metrics?
- » Are metrics configurable?

Appendix B: Vendor Evaluation Criteria

VENDOR CRITERIA

Product stability.

- Was the product stable?
- Some issues are inevitable such as memory management, a checker not firing correctly, and so on, but does the big picture demonstrate a commitment to quality?

Defect reports.

- Were reported bugs resolved in a reasonable time?
- Were showstoppers fixed promptly?
- Were less significant issues addressed or at least scheduled for a future release?

Feature requests.

- How were your feature requests handled? Try to push at least a handful through as a test.
- How does the vendor proceed if you provide a list of feature requests that make business sense and would benefit the entire user base?

If the vendor works systematically at the feature requests and implements them quickly, it's a sign that they have robust development resources and are willing to invest R&D into improving the product.

Overall support. How promptly are your questions answered by support? As with feature requests, don't be shy. This is another important test. If you can't get reasonable response times for just a few users in the initial evaluation period, chances are you won't have adequate support for a global deployment.

Vendor viability. An investment in tools is also an investment in the vendor. Having confidence in their longevity and prosperity is important. Embedded systems have long lifespans and it's critical that vendors are available for support and maintenance for these long lifecycles.

- How long has the vendor been in business?
- If they're new to the market, are they well funded?
- Do they have a good track record of customer support and success?

IS YOUR VISION IN SYNC WITH THE VENDOR'S?

Initiate the conversation to understand the vendor's vision for how the tool would be deployed and used in an organization's environment.

START THE CONVERSATION

Follow these steps to get the conversation started.

1. Explain the problems that static analysis is required to address. Ask the following questions:
   - Does the vendor agree that static analysis is the best path to solving these problems?
   - Are other strategies suggested?
   - Can the vendor help set objective criteria for assessing whether their static analysis tool addresses the required problems?

Set goals that can be objectively measured from the start. It helps later when assessing whether the tool is helping to achieve the expected results.

2. Describe the target environment (project size, policies, infrastructure, and so on). Ask how the vendor has helped other organizations in similar situations.
3. Explain the team's vision for tool deployment, adoption, and usage over the next two to three years. Ask the vendor:
   - Does this seem feasible?
   - How are mismatches handled?
   - If there are significant mismatches apparent at this point, what kind of resolution is proposed?

It's reasonable to expect the vendor to accommodate new feature request that could benefit their other customers. There's widespread value in integrating the tool into a development environment that many other development organizations happen to use, like a problem reporting system or a requirements management system. Such integrations can lead to a significant advantage to them. Be aware that some vendors purposely expose their API so that users can extend the product for their own needs.

ARE YOU A MATCH?

Below are some signs and checkers.

- **If the vendor has issues with what you're trying to accomplish,** do they offer a convincing explanation of why it may not be a wise strategy and offer an alternative that makes sense? If a vendor is willing to provide valuable feedback—especially before you have committed to a contract—it's a positive sign of a good working relationship.
- **If the vendor seems to bend over backwards to accommodate any request,** like agreeing to implement functionality that isn't central to their capabilities and won't appeal to other customers, then this diminishes their credibility. How will the tool evolve if they are willing to accommodate anything and everything? And what gets left behind in the rush to add every feature request?