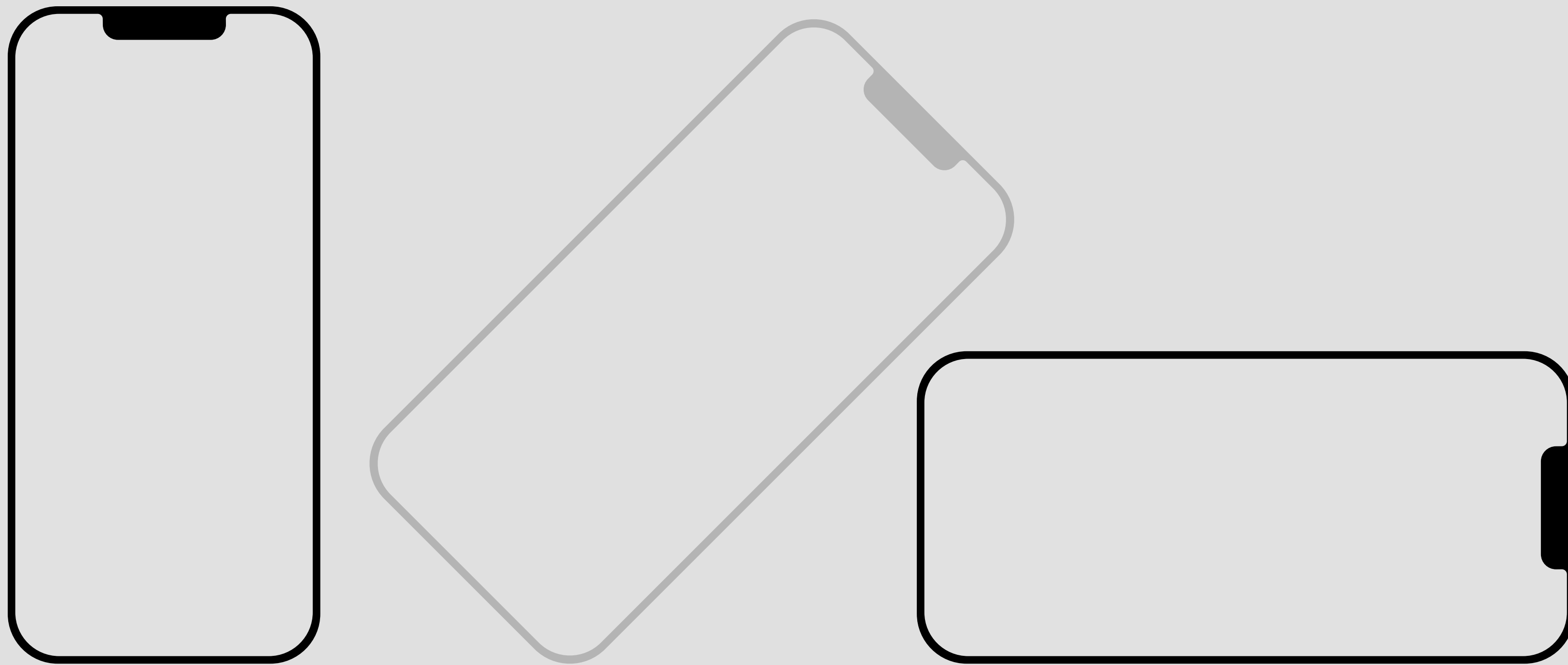


Mastering CUDA Code Quality

Your Playbook for Developing
Game-Changing CUDA Applications



Reading on a mobile device?
Turn it sideways for the best reading experience.



Preface

This guide is long – some might even argue too long.

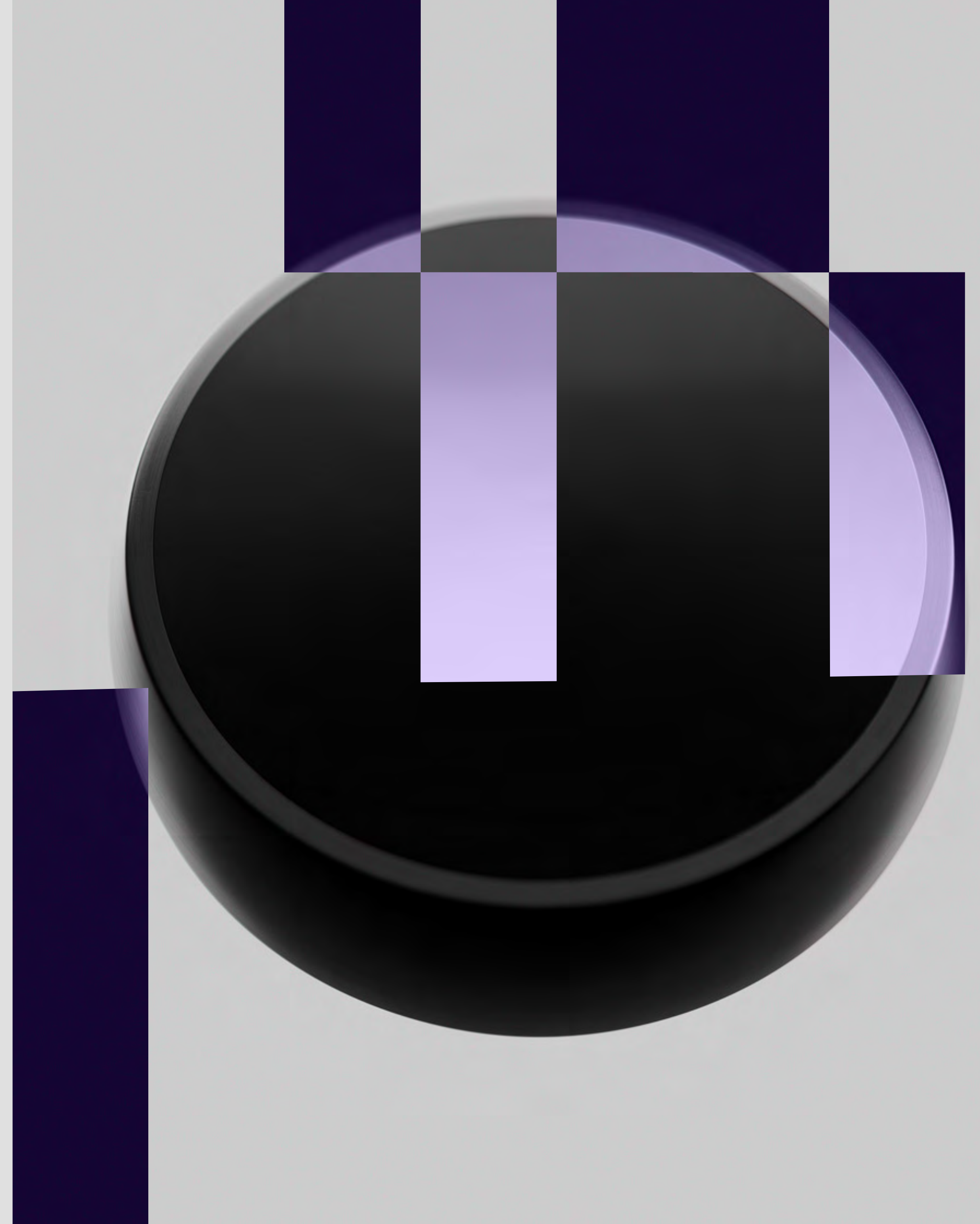
But software quality is no trivial matter. It is, in essence, what makes or breaks a product and in safety-critical environments even a matter of life and death. We therefore believe talking (or in your case: reading) about it, is time well spent.

The guide is meant for developers – junior and senior alike – and looks at software quality specifically for CUDA C++ projects. While the importance of software quality is the same – irrespective of the programming languages used – there are few tools which help CUDA developers achieve the level of quality they desire.

Axivion for CUDA was designed specifically to support the development of CUDA C++ applications, and the guide will show you how. It focuses on software used in safety-critical environments, but the methods described can be applied to any use case.

The guide also touches on economic benefits of high-quality code. While these might not be key drivers for developers, knowing about them can help explain to non-technical staff, why investing in software quality is the opposite of wasting resources.

Whether you read this guide cover to cover or just pick the chapters most interesting to you, we hope it will help you write code which makes a difference .



Contents

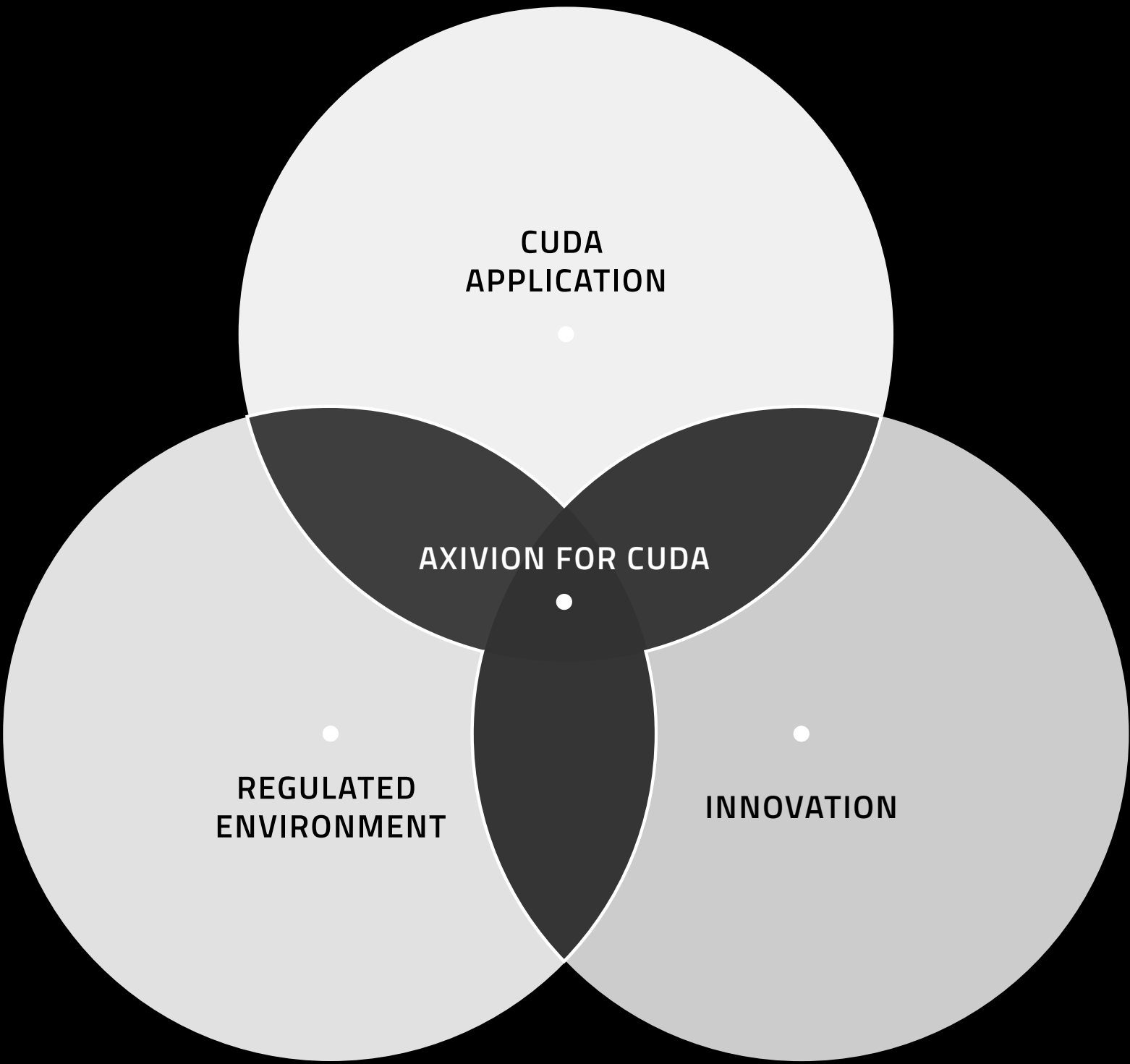
1. Understanding Software Erosion and Technical Debt	06	4. Software Quality as Economic Advantage	27
Software Quality – How to Maintain and Protect it	07	Software Quality as Economic Advantage	28
Definition and Causes of Software Erosion	08	Improved ROI/TCO: Get More for Less	30
Software Erosion vs. Technical Debt	09	Make Sure the Best of the Best Work for you	31
Software Erosion Corrupts Software Quality	10	Protect What you Have Created	32
Detecting Software Erosion	13		
2. Axivion Static Code Analysis for CUDA	14	5. Where it Matters Most: CUDA Applications in Safety-Critical Environments	33
Getting Started With Axivion for CUDA	15	CUDA Applications in Safety-Critical Environments	34
Beyond Static Code Analysis	17	Software Safety & Security	35
		Coding Guidelines are a Must	36
		Functional Safety in CUDA Applications	37
3. Axivion Architecture Verification for CUDA Projects	18	NVIDIA's CUDA C++ Guidelines for Robust and Safety-Critical Programming	38
Architecture Verification: The Secret to High-Performance Computing	19	Other Safety and Security Measures	39
Architecture Checks / Architecture Recovery	23		
Architecture Archaeology	24	6. Leveraging Code Analysis for CUDA	40
Common Architectural Pitfalls in CUDA Development	25		

Introduction

In today’s high-performance computing landscape, CUDA has become a cornerstone technology for accelerating applications across industries: from autonomous driving and robotics to scientific simulations and AI. But as CUDA applications grow in complexity and scale, performance alone is no longer enough. Safety, security, reliability, and maintainability are now essential pillars of successful software development. Finding the right balance between innovation and regulation ensures that cutting-edge CUDA applications are not only fast and scalable, but also safe, secure, and audit-ready.

Most static analysis solutions either ignore CUDA-specific constructs or treat them as generic C/C++ extensions, missing the nuances of GPU programming. But Axivion for CUDA has been designed specifically for CUDA applications, offering comprehensive support for CUDA syntax, architecture verification, and compliance with industry standards and coding guidelines – including NVIDIA’s CUDA C++ guidelines for Robust and Safety-Critical Programming.

By integrating seamlessly into CI/DevOps workflows, Axivion enables developers to maintain clean, compliant, and future-proof code. More importantly, it helps teams comply with the rigid rules and regulations in safety-critical environments.



This guide explains what is needed to develop high-quality CUDA applications and how Axivion for CUDA helps developers tackle software erosion, prevent technical debt, and maintain architectural integrity without losing the ability to innovate.

We will start by focusing on software quality, how to maintain it and the economic benefits of doing so. And then look at the practical aspects of developing CUDA applications in safety-critical environments.

1

Understanding Software Erosion and Technical Debt

Software Quality – How to Maintain and Protect it

Internal Quality

External Quality

In the fast-developing world of GPU-accelerated computing, high performance is the ultimate goal, but only half of the equation. Reliability, maintainability, and scalability are just as critical. As CUDA applications grow in complexity and scale, software quality becomes the bedrock upon which innovation stands.

There are various coding guidelines and industry standards to ensure internal and external software quality, such as the NVIDIA CUDA C++ Guidelines for Robust and Safety-Critical Programming. Following them is not optional, but a necessity. Adhering to them under time pressure can be challenging, as changes to the rules and regulations require decisions whether adopting these can be done fast but without compromising the quality.

So how can software quality be ensured when you have to meet deadlines?

The answer:
Stop software erosion and prevent technical debt.

Many companies do not prioritize software quality because this sounds like a time-consuming, sheer impossible task. But the truth is, many of the steps needed to maintain a high standard of software quality, can easily be automated by deploying advanced code analysis tools such as Axivion for CUDA.

Unlike other software quality tools, Axivion for CUDA was developed specifically for applications which extend C and C++ with additional keywords and APIs to allow developers to write code that executes directly on NVIDIA GPUs, alongside traditional CPU code. Once set up, Axivion for CUDA seamlessly integrates into your CI/DevOps environment and ensures your code not only is but also stays clean.

Additionally, Axivion can also check compliance with coding guidelines and various industry standards such as MISRA on CUDA. This not only adds another layer of quality to your software, it also helps to prepare and implement assessments and audits.

Axivion for CUDA ↗

TYPICAL CAUSES OF SOFTWARE EROSION INCLUDE:

Definition and Causes of Software Erosion

Software is inanimate. But it grows and evolves over time, just like living organisms. These changes to the code leave their marks.

Software erosion, software decay, software rot, technical debt: These all synonymously refer to the gradual deterioration of the structure of a software’s source code. This deterioration manifests in various ways, such as reduced performance, the constantly growing difficulty in modifying the code to meet new requirements, and an increase in programming errors.

What makes software erosion so tricky is that the individual flaws, which cause software erosion on the outside aren’t necessarily incorrect or prevent the program from running as intended. But what begins with small and undocumented omissions or additions over time accumulates and impacts the understanding of the software. The various deviations from the original architecture make the code incomprehensible. It becomes difficult to test and failures in the field become more frequent.

To avoid rendering the code unusable, it needs to be healed, meaning refactoring is required. By refactoring your code, you stop the software erosion momentarily. But this is just treating the symptoms. To cure the disease, you have to prevent technical debt from happening.

Architectural drift or architectural debt

Occurs when the implementation diverges from the intended architecture. It is an accumulation of expedient short-term solutions, that can make future modifications impossible.

Dead code

Unreachable code which is not executed during runtime and wastes resources (memory, processor time, performance).

Clones or duplicated code

Identical pieces of code that exist in more than one location. While updating code, you run the risk of forgetting to update the clone in other parts of the software, too.

Metric violations

Metrics (e.g. lines and tokens, MacCabe complexity or NPath counts) are used to keep code clean. Violating these amplifies software erosion.

Call cycles

Functions call each other and therefore pose the risk of endless recursion. If a direct cycle occurs, there is a risk the system can crash at any time.

Software Erosion vs. Technical Debt

While the terms Software Erosion and Technical Debt are very often used as synonyms, there is a small, yet important difference, which should be highlighted.

Technical debt can be compared to financial debt and can imply conscious decision and with that the notion that it is someone's fault or at least responsibility. Technical debt occurs when pros and cons are weighed, and it is decided to go with a quick-fix for the moment and make necessary corrections later. Very often there are good reasons to work this way, and the intentional decision assumes that this technical debt will be paid back sometime.

However, more often than not, "sometime" never comes. Software quality is not the main concern in companies and thus not given the necessary attention and resources. At some point the debt has piled up so much that it bankrupts you: There is no way you can refactor the code; it is beyond repair, and all you can do is start developing the software from scratch.

Software erosion on the other hand – while used interchangeably with technical debt – is the process over time that can lead to technical debt. It is better explained by comparing it to natural erosion. It happens. Whether we want it or not. We believe it is important not to assume that someone deliberately tried to corrupt the software; instead, these issues in the code can be compared to growing pains. They are a natural part of software development.

The eroding factors in nature are heat, cold, wind, water, ice etc. and they continuously damage a cliff, rock or shoreline, even if the damage at first is not visible. In software development, time pressure and scarce resources are the eroding factors. Code clones, violations against coding guidelines, metric outliers, architecture deviations are the cracks in the rock of source code, which ultimately make it break apart.

But just because "it happens", doesn't mean you can't protect yourself from it. Stopping software erosion and preventing the need for technical debt ensures your software can function happily ever after.



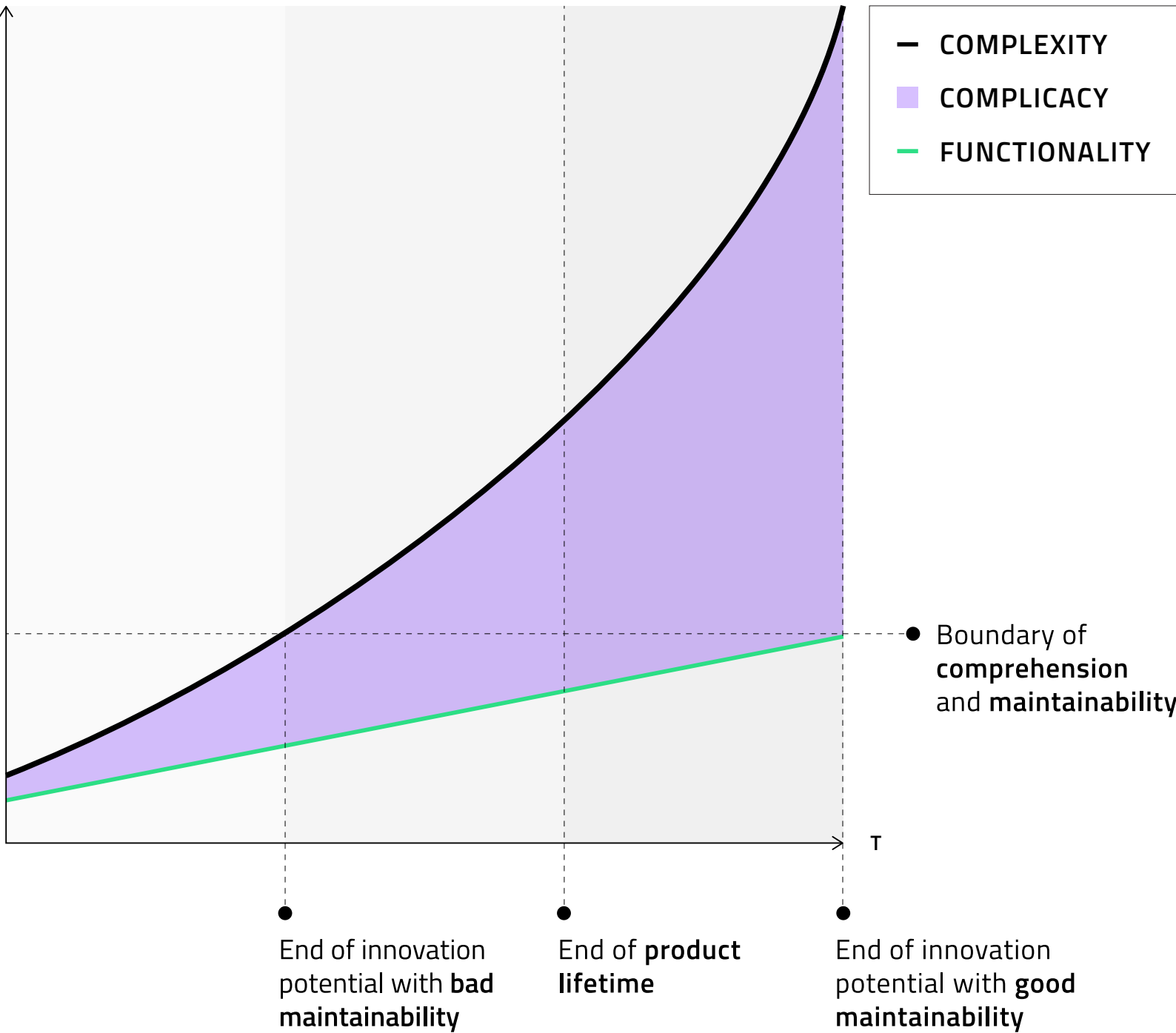
Software Erosion Corrupts Software Quality

As already mentioned, software erosion can result in reduced performance and a growing difficulty in modifying the code. But what does this mean? In short: It limits your innovation potential .

In the beginning, the functionality and complexity grow in parallel and software erosion has little to no impact on the maintainability and your ability to add features. This ability is based on your understanding of the software and knowing what effect changes to it will have.

But as the software continues to develop, this changes very quickly and the complexity grows exponentially. Once you cross this boundary of comprehension the software becomes more complicated to maintain and the number of errors per line of code increases dramatically. Ignoring software erosion means that instead of adding features, you spend most of the time trying to figure out what needs to be changed or fixed and innovation ends. Refactoring becomes virtually impossible and your development slows down to a point that the software becomes unreliable.

However, stopping software erosion and having a clean code allows you to continue to develop your software. The reduced complexity means you spend time with innovation and not bug-fixing .



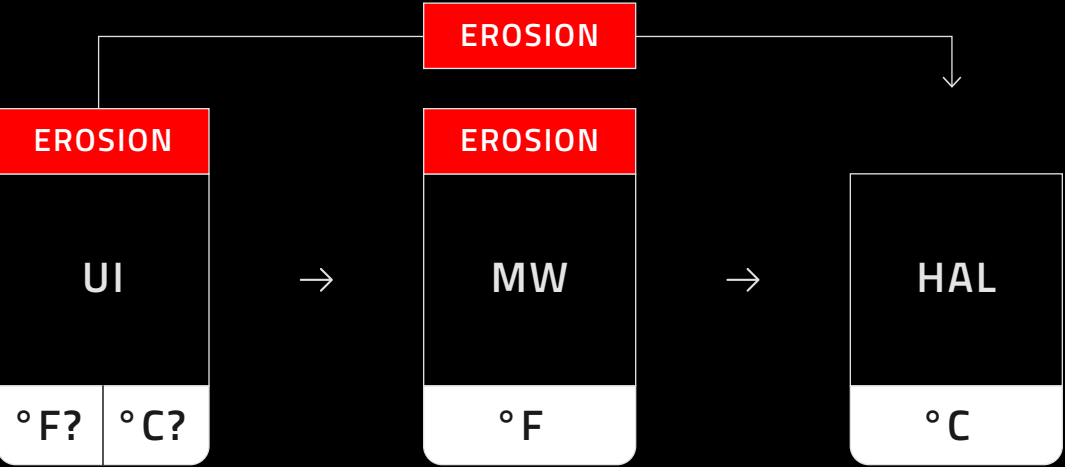
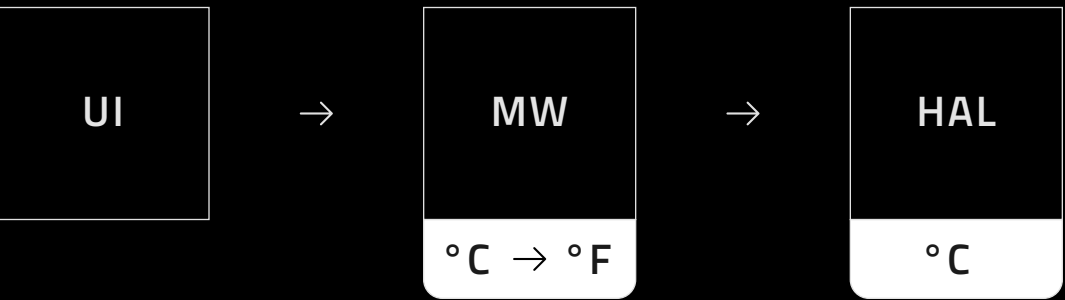
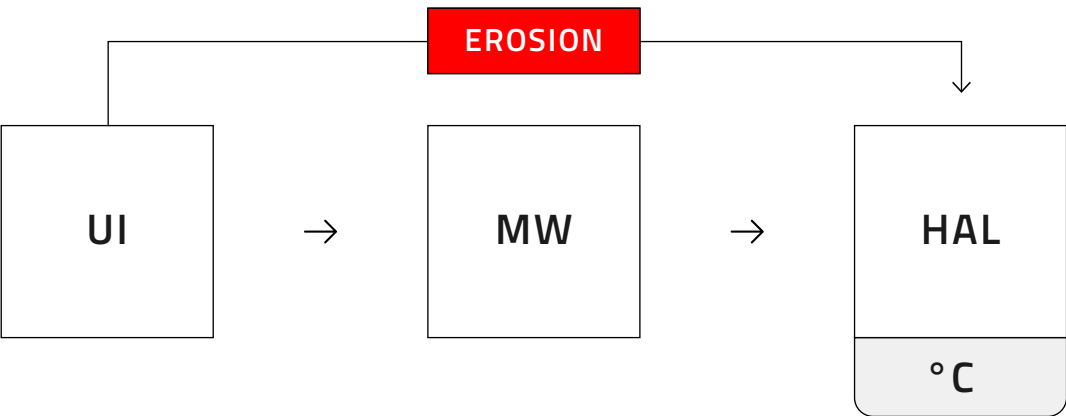
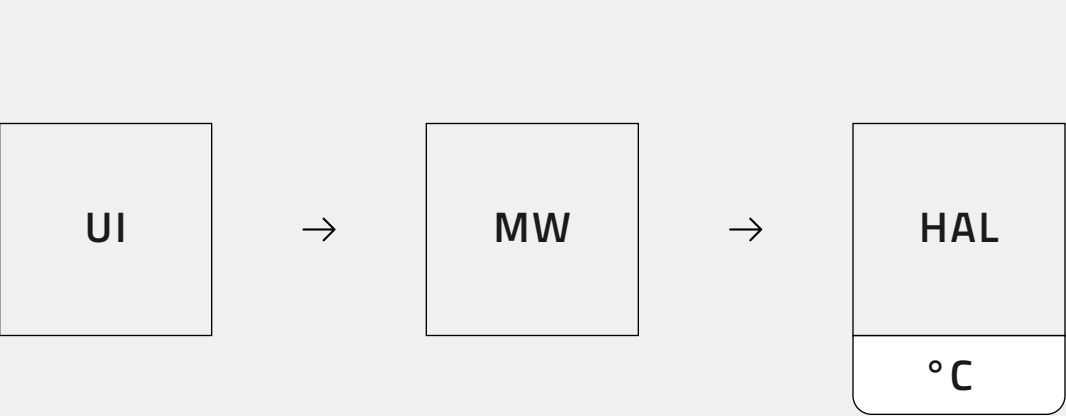
Software erosion happens faster than you think and, in the beginning, doesn't even seem to be an issue.

Let's look at a simple example - a temperature sensor in a car using °C – and assume the following structure: There is a user interface (UI) which accesses the middleware (MW), which in turn accesses the hardware abstraction layer (HAL) with the temperature sensor. The user interface is not supposed to access the HAL directly

But what happens if a developer ignores the rule and the UI directly gets information from the HAL? In the beginning, seemingly nothing. No one will notice a difference as long as it doesn't cause any issues. But in reality, you already encounter software erosion as the intended architecture and the implemented code don't match. You have added an (invisible) layer of unnecessary complexity.

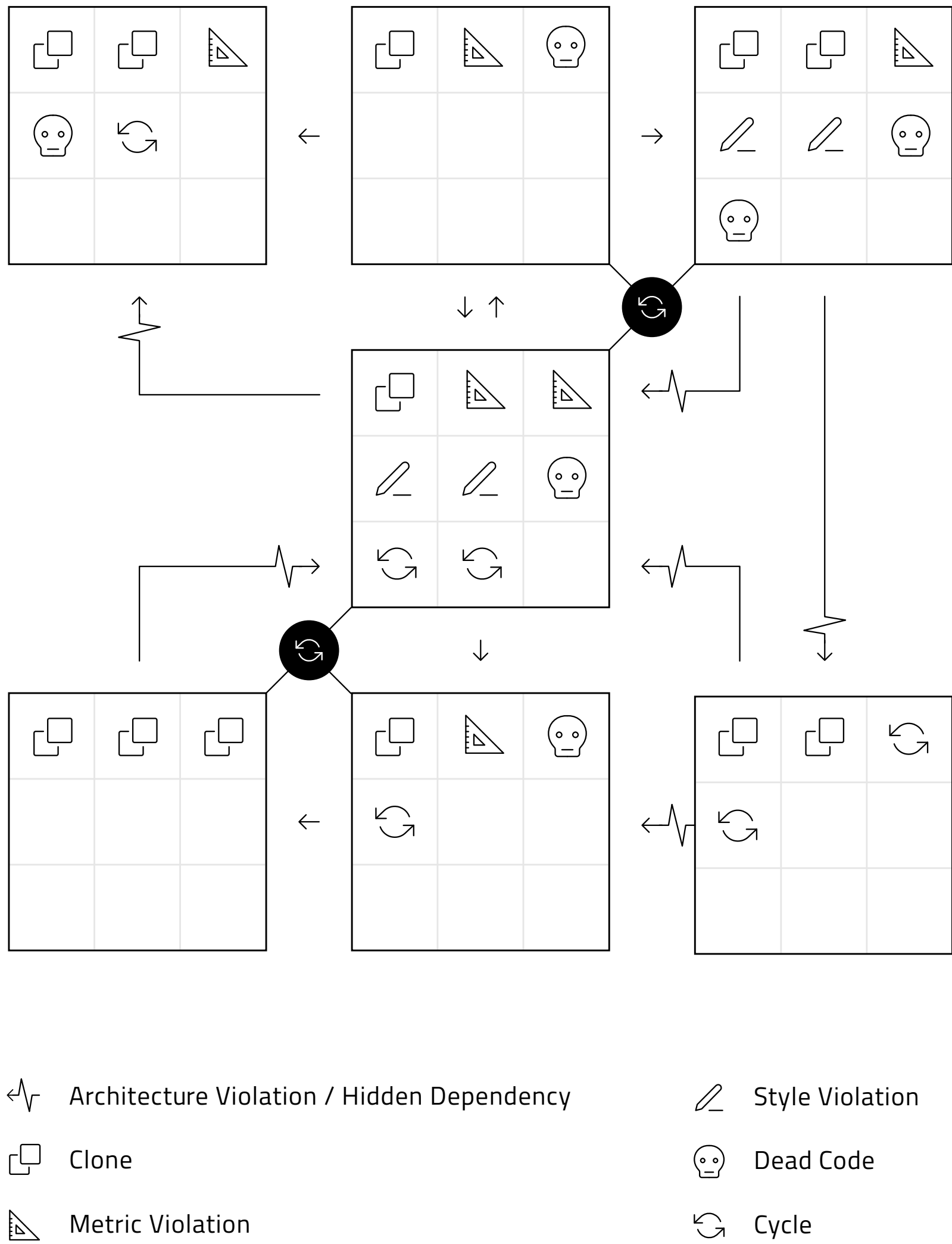
Now imagine you want to display the temperature in Fahrenheit instead of Celsius. For this the Software Architect simply adds the conversion in the middleware. The conversion in the MW is tested and works as intended.

But during the final, full system integration testing, you notice that the UI displays a warning about icy roads at 32°C when it should be 32°F. With the software release due in just a few days, you need a quick and easy to implement solution. You know that the conversion in the middleware works, so why not just copy it into the user interface? This works perfectly well and your problem is solved. But now you have added a clone and thereby more software erosion.



Over time all these small changes add up until the once clean and straightforward software turns into incomprehensible chaos.

It is a vicious circle: Lack of time leads to developers neglecting the inner software quality and the subsequent erosion of the software impairs comprehension. This makes costly workarounds necessary, which leads to the code deteriorating even more. The development slows down and now the time pressure increases even more.



Detecting Software Erosion

To end the vicious circle and to avoid code eroding beyond comprehension and maintainability, issues need to be fixed or even prevented as early as possible. This reduces the damage which issues can cause and requires fewer resources to do.

But how can you detect software erosion if in the beginning it is not necessarily noticeable? Even if you wanted to check your code regularly, some issues are virtually impossible for humans to detect. That's why implementing automated code analysis right from the start is so important.

But what if code is already suffering from software erosion and technical debt?

This might be the case if you are working with legacy or 3rd-party code. You need to ensure that this code has the same high quality as the rest of the software even if you had no control over it before. And this is where Axivion can help.

Control your software development.

Be in charge, no matter where the code comes from.

For this make sure to verify the software architecture and use static code analysis during development and checks for compliance with coding guidelines and standards.

2

Axivion Static Code Analysis for CUDA

Getting Started with Axivion for CUDA

What is static code analysis?

Static code analysis is a method to detect bugs, security vulnerabilities and deviation from coding guidelines and industry standards. Unlike software testing, static code analysis does not require the code to be executed. This allows to only check parts of the code before committing it and doesn't require the software to be complete.

The Static Code Analysis of Axivion for CUDA will detect:

Clones

Cycles

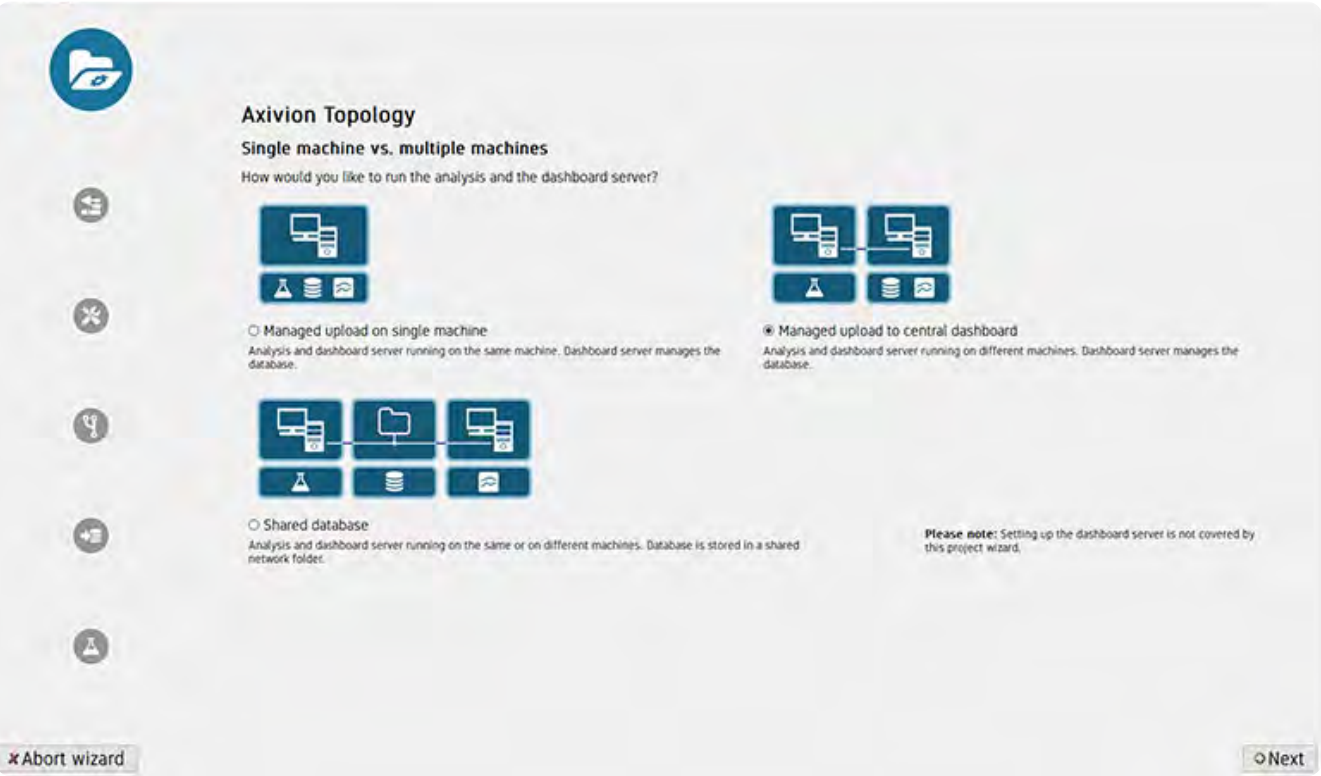
Dead Code

Metric Violations

Issues with coding guidelines and safety & security rules

All issues which cause software erosion and lower the code quality as described in Chapter 1.

After installing Axivion for CUDA and activating the license, a set-up wizard walks you through the necessary steps for the project configuration.



Once the initial set-up is complete a full analysis of the code is performed to establish a baseline.

1

Define a project root directory and source code path and then specify your build environment.

2

Select the correct coding guidelines and metrics for your project.

3

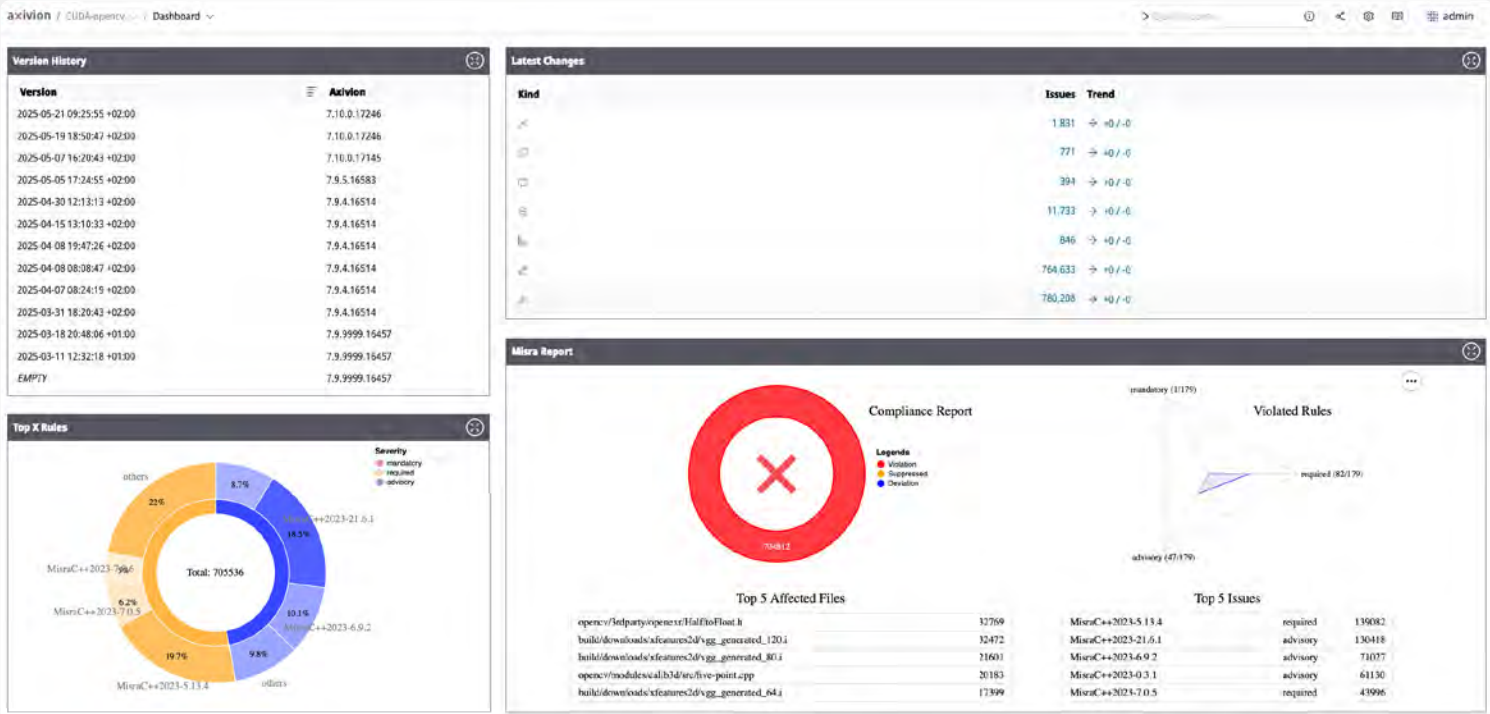
Configure the rule sets and thresholds. In this step you have the possibility to add customized rules to comply with company-specific requirements.

4

Set up the integration with your CI/DevOps environment.

5

Configure reports to be generated after each build.



Issue Table									
ID	Error Number	Message	Entity	Path	Prov...	Sev...	Owners		
SV814417	ax000	Missing check for errors (by calling cudaGetLastError) after launching a kernel	polarToCartImpl_1	polar_cart.cu (/opencyc_contrib/modules/cudaarithm/src/cuda/polar_cart.cu)	axivion	advisory	xaver		
SV814418	ax000	Missing check for errors (by calling cudaGetLastError) after launching a kernel	polarToCartImpl_1	polar_cart.cu (/opencyc_contrib/modules/cudaarithm/src/cuda/polar_cart.cu)	axivion	advisory	xaver		
SV814419	ax000	Missing check for errors (by calling cudaGetLastError) after launching a kernel	polarToCartInterleavedImpl_1	polar_cart.cu (/opencyc_contrib/modules/cudaarithm/src/cuda/polar_cart.cu)	axivion	advisory	xaver		
SV814420	ax000	Missing check for errors (by calling cudaGetLastError) after launching a kernel	polarToCartInterleavedImpl_1	polar_cart.cu (/opencyc_contrib/modules/cudaarithm/src/cuda/polar_cart.cu)	axivion	advisory	xaver		
SV814421	ax000	Kernel parameters passed by value should be trivial	mag	polar_cart.cu (/opencyc_contrib/modules/cudaarithm/src/cuda/polar_cart.cu)	axivion	required	xaver		
SV814422	ax000	Kernel parameters passed by value should be trivial	angle	polar_cart.cu (/opencyc_contrib/modules/cudaarithm/src/cuda/polar_cart.cu)	axivion	required	xaver		
SV814423	ax000	Kernel parameters passed by value should be trivial	x	polar_cart.cu (/opencyc_contrib/modules/cudaarithm/src/cuda/polar_cart.cu)	axivion	required	xaver		
SV814424	ax000	Kernel parameters passed by value should be trivial	y	polar_cart.cu (/opencyc_contrib/modules/cudaarithm/src/cuda/polar_cart.cu)	axivion	required	xaver		
SV814425	ax000	Kernel parameters passed by value should be trivial	mag	polar_cart.cu (/opencyc_contrib/modules/cudaarithm/src/cuda/polar_cart.cu)	axivion	required	xaver		
SV814426	ax000	Kernel parameters passed by value should be trivial	angle	polar_cart.cu (/opencyc_contrib/modules/cudaarithm/src/cuda/polar_cart.cu)	axivion	required	xaver		
SV814427	ax000	Kernel parameters passed by value should be trivial	x	polar_cart.cu (/opencyc_contrib/modules/cudaarithm/src/cuda/polar_cart.cu)	axivion	required	xaver		
SV814428	ax000	Kernel parameters passed by value should be trivial	y	polar_cart.cu (/opencyc_contrib/modules/cudaarithm/src/cuda/polar_cart.cu)	axivion	required	xaver		
SV814429	ax000	Kernel parameters passed by value should be trivial	mag	polar_cart.cu (/opencyc_contrib/modules/cudaarithm/src/cuda/polar_cart.cu)	axivion	required	xaver		
SV814430	ax000	Kernel parameters passed by value should be trivial	angle	polar_cart.cu (/opencyc_contrib/modules/cudaarithm/src/cuda/polar_cart.cu)	axivion	required	xaver		
SV814431	ax000	Kernel parameters passed by value should be trivial	xy	polar_cart.cu (/opencyc_contrib/modules/cudaarithm/src/cuda/polar_cart.cu)	axivion	required	xaver		
SV814432	ax000	Kernel parameters passed by value should be trivial	xy	polar_cart.cu (/opencyc_contrib/modules/cudaarithm/src/cuda/polar_cart.cu)	axivion	required	xaver		
SV814433	ax000	Kernel parameters passed by value should be trivial	magAngle	polar_cart.cu (/opencyc_contrib/modules/cudaarithm/src/cuda/polar_cart.cu)	axivion	required	xaver		
SV814434	ax000	Missing check for errors (by calling cudaGetLastError) after launching a kernel	mag_withoutLearning	mag.cu (/opencyc_contrib/modules/cudaarithm/src/cuda/mag.cu)	axivion	advisory	xaver		



Issue Table

2025-11-03 09:49:56 +01:00

Show suppressed issues

```
opencyc_contrib/modules/cudaarithm/src/cuda/polar_cart.cu (843b6ede)
2290
2291
2292 const T scale = angleInDegrees ? static_cast<T>(CV_PI / 180.0) : static_cast<T>(1.0);
2293
2294 if (mag.empty())
2295     polarToCartImpl_1<T, false> <<<grid, block, 0, stream>>>(mag, angle, x, y, scale);
2296 else
2297     polarToCartImpl_1<T, true> <<<grid, block, 0, stream>>>(mag, angle, x, y, scale);
2298
2299
2300 template <typename T>
2301 void polarToCartInterleavedImpl(const GpuMat& mag, const GpuMat& angle, GpuMat& xy, bool angleInDegrees, cudaStream_t stream)
2302 {
2303     typedef typename MakeVec<T, 2>::type T2;
2304
2305     const dim3 block(32, 3);
2306     const dim3 grid(divUp(angle.cols, block.x), divUp(angle.rows, block.y));
2307
2308     const T scale = angleInDegrees ? static_cast<T>(CV_PI / 180.0) : static_cast<T>(1.0);
2309
2310     if (mag.empty())
2311         polarToCartInterleavedImpl_1<T, false> <<<grid, block, 0, stream>>>(mag, angle, xy, scale);
2312     else
2313         polarToCartInterleavedImpl_1<T, true> <<<grid, block, 0, stream>>>(mag, angle, xy, scale);
2314
2315     template <typename T>
2316     void polarToCartInterleavedImpl(const GpuMat& magAngle, GpuMat& xy, bool angleInDegrees, cudaStream_t stream)
2317     {
2318         typedef typename MakeVec<T, 2>::type T2;
2319
2320         const dim3 block(32, 3);
2321         const dim3 grid(divUp(angle.cols, block.x), divUp(angle.rows, block.y));
2322     }
2323 }
```

Details @ 2025-11-03 09:49:56 +01:00

Id: CUDA-opencyc:SV814417

Owners: xaver

Justification:

Location: /opencyc_contrib/modules/cudaarithm/src/cuda/polar_cart.cu:308:83

Provider: axivion

Severity: advisory

Error Number: CUDA-1.4

Message: Missing check for errors (by calling cudaGetLastError) after launching a kernel

Entity: polarToCartInterleavedImpl_1

Template: polarToCartInterleavedImpl-double<T>

Instantiation: polarToCartInterleavedImpl-Float<T>

Rule @ 2025-11-03 09:49:56 +01:00

CUDA 1.4 (user kernel launch) Check for errors after launching a kernel by calling cudaGetLastError

after launching a kernel with the CUDA C++ kernel launch syntax, call cudaGetLastError and check whether its return value indicates that a synchronous error occurred before the next CUDA API call

Scope: Host, Device

Audience: CUDA C++

Category: Advisory

Hardware Applicability: All Compute Capabilities

Rationale: CUDA C++ kernel launches return void, but reports errors by setting the CUDA global error state, which can be checked with cudaGetLastError. There are two classes of errors that an interface can return:

From the dashboard developers can access the issue table with more details and from there directly access the code for developers to fix the issue in their local IDE. They also get more details about the rule to understand why this is an issue.

The results of the analysis are displayed in a customizable dashboard and, after the second code check, include a delta analysis. The delta analysis tracks the number of issues over time and shows you the changes vs. the last check.

Axivion for CUDA product tour ➔

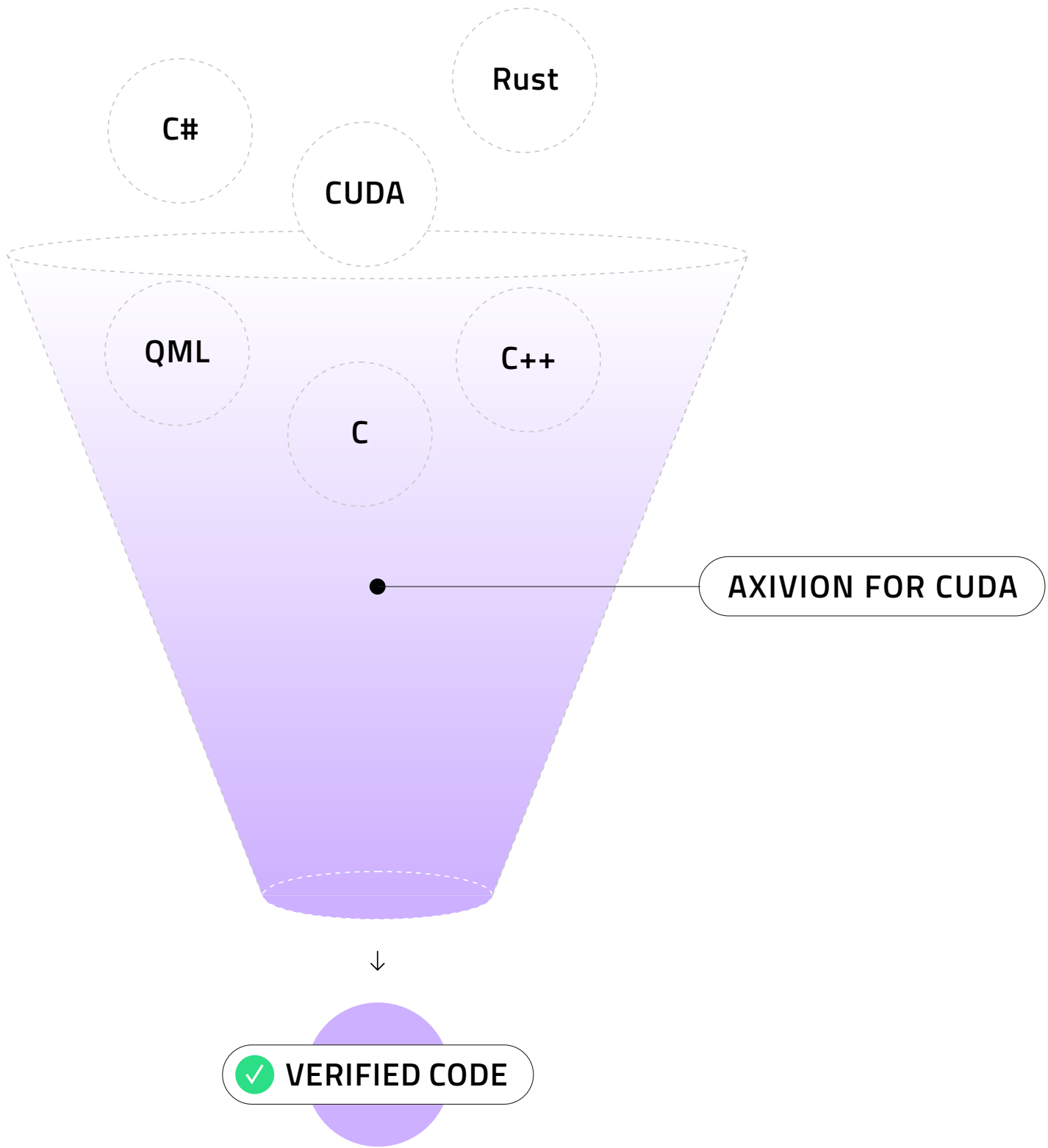
Beyond Static Code Analysis

Axivion for CUDA does more than just detects bugs. It ensures the entire code fulfills the requirements set by leading standards and guidelines (MISRA, CUDA C++ Guidelines for Robust and Safety-Critical Programming, ISO 26262 etc.) → Chapter 5.

Comprehensive reports give clear overview of the status of your software and allow you to provide the evidence that the development process. They are generated and sent out automatically and can be adapted to your needs.

The **Tool Qualification Kit** supports classification and qualification of your tool chain and can be integrated into your CI process. The technical automation of the validation tests allows you to repeat them efficiently after changes have been made, for instance in the form of updates or upgrades, in order to benefit from improvements and innovations, thus also allowing the suitability check to be repeated. The provided test driver carries out one or more tests for the relevant rule sets and reports the status of these tests.

Axivion’s **multi-language support** ensures that your entire code is checked without the need to run several separate analyses or even do a manual review. Take checking MISRA rules as an example. Most tools simply ignore the CUDA code or flag it as “false”. This bit of the code then needs to be reviewed separately. Not so with Axivion. The MISRA checker will analyze the entire code so confirm compliance. And as Axivion can analyze multiple languages simultaneously, it can also verify other potential issues in multi-language code, such as if cross-calling from one language to the other is correct and as intended.



3

Axivion Architecture Verification for CUDA Projects

Architecture Verification: The Secret to High- Performance Computing

Why Verify Your Architecture?

Software architecture and design need to match with the code for you to be sure you can use the software architecture as a guide and baseline for discussing the impact of new features.

Only then is a long-term targeted and planned development of your products possible. Axivion for CUDA ensures your code complies with your architecture. In addition to the functional architecture, the tool also reviews and checks safety and security architecture specifications for compliance, e.g. Freedom from Interference.

The Importance of Architecture Verification for CUDA Projects



Software architecture refers to the core concepts and characteristics of a software system within its surroundings, expressed through its components, connections, and the guiding principles of its creation and development.

The term “architecture” in software development was initially borrowed from the construction industry due to the similarities between the two fields. In the past, the waterfall methodology was widely used in software development, so detailed plans were required before writing any code. This approach was similar to the planning required in construction, where the architectural design must be finalized before construction begins. But things have changed. Modern software development methodologies are designed to be flexible and easy to change over time, so there’s less need for strict planning at the beginning. Nevertheless, defining a software architecture remains essential. Standards such as ASPICE refer to the V-Model, which expects detailed plans.

The software architecture is like a map showing how the software works and fits into the world around it. It’s important to make sure the software works well in the environment in which it is used because it minimizes potential disruptions and maximizes the software’s longevity and effectiveness in serving its purpose.

To help everyone understand the software architecture, a description is created that explains how it works and solves the problems it was created for. This description is like a guidebook that helps people see how the software will meet their needs.

Different people care about different things when it comes to software. To ensure everyone can understand the software architecture, it not only requires good documentation of what the software should look like, you must make sure the implementation does not deviate from the plan.

How Architecture Verification Works With Axivion

Setting up Axivion to verify your architecture can be done in four simple steps:

Step 1
Create your architecture model

For this simply import any existing machine-readable model of your architecture via the interface or use the integrated modeler.

Step 2
Create your code model

Extract your code model from your source code. The code model represents entities (e.g. source files) and classes or functions as well as the dependencies between them. Axivion can automatically construct your code model by analyzing your source code project.

Step 3
Map the code to the architecture

Now establish how your code elements correspond to architecture components, by assigning the code elements to the architecture elements. Depending on your product structure and architecture model, this can be done:

manually (e.g. with the Gravis modeler)

automatically with Axivion by naming convention and hierarchical information

or by using information given in the model (e.g. tagged values)

Step 4
Interpret the dependencies

During this step you interpret the dependencies within the architecture model and specify what they mean; in other words, how to interpret architecture dependencies to match code dependencies. A simple example would be: Component A is allowed to call functions in component B. Of course, especially for UML models, far more advanced interpretations of dependencies (required/provided interfaces) are also possible.



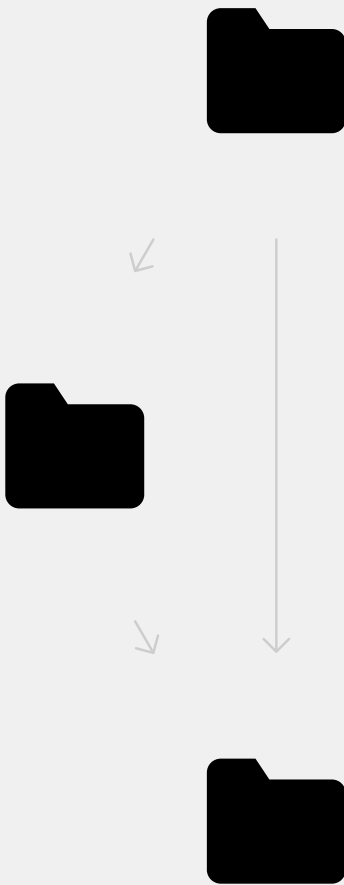
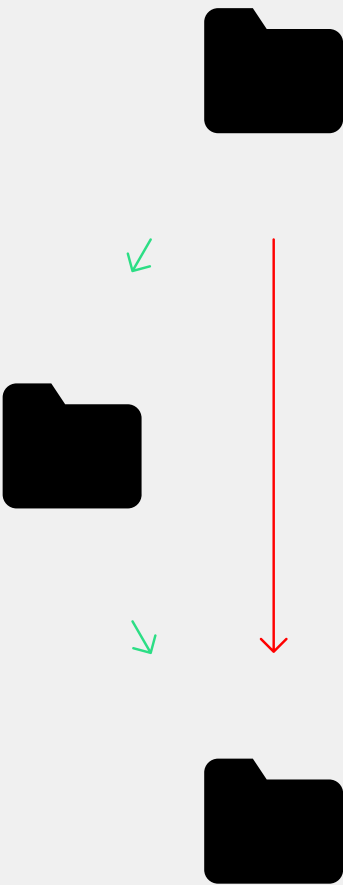
Once the initial setup is done, Axivion for CUDA can verify your architecture and will identify:

Convergences

Divergences

Absences

This automated verification delivers exhaustive and reliable results and you can now decide how to react to these findings.

	Correct the deviation in the source code	Update the architecture specifications	Accept the deviation temporarily
<p>It is important to note that this is not a one-time verification for a specific moment in time. On the contrary – what makes Axivion for CUDA stand out from competitors is the ongoing architecture verification during the software development process. Before code is pushed or committed, developers will be able to see if their code matches the intended architecture and can keep the code clean.</p>	<p>If this deviation is a mistake, the code should be corrected so it matches the architecture again.</p>	<p>If this deviation actually makes sense, you should update the architecture accordingly.</p>	<p>If this deviation is something you can accept temporarily, leave it and deal with it later.</p>
			

Architecture Checks

Axivion is designed to be part of your continuous software analysis and development process. Once set up, it easily integrates into the daily review process. This replaces error-prone manual reviews with exhaustive and reliable, automated checks. The results are summarized in a dashboard and linked to the source code and can also be directly presented in the developer's IDE. Developers can look at the deviations and can directly make the necessary changes as needed.

Axivion's delta analysis focuses on changes in the codebase. Instead of needing to inspect all issues every time, the developer can focus on only those caused by the latest code, allowing for faster and more focused reviews. For this a baseline is established (usually the previous check) and when new code is committed or pushed Axivion identifies the differences (delta) between the current and previous version.

Developers get immediate insight into the changes (e.g. between commits, builds or branches) and don't have to wait for a full analysis to run. They can focus on fixing only new issues and save a significant amount of time spent on code reviews and debugging while at the same time maintaining architectural consistency.

The result: A reliable software architecture for longer lifecycles.

Architecture Recovery

There are various reasons for not having a clear overview of the architecture of a software. It might be that over the years different people updated different parts of the documentation and now there's no clear picture of what the architecture looks like. Or, the project started so small that documenting the architecture was not seen as a priority. But after adding several new features, it has become difficult to understand the software - especially for new people joining the team.

Most often, however, architecture recovery is needed when embedding code from 3rd parties, but never received the required documentation or only incomplete and outdated documentation. Nevertheless, to use this code efficiently and to ensure it meets the highest quality standards, it is important to understand how the software was designed and how the different parts relate to each other. This is where Axivion's architecture recovery can help.

Based on whatever fragments and assumptions exist within the team, an architectural hypothesis is formed. Axivion then validates this initial hypothesis by comparing it with the implementation and deviations are resolved in an iterative process. This process uses the same technology and functionality as the architecture checks.

What if absolutely no documentation is available?

Axivion can help with this, too, with Architecture Archaeology.

Architecture Archaeology

Architecture archaeology is an extreme form of architecture recovery. With Axivion completing a seemingly unmanageable task is in fact possible: documenting a software project long after it was launched.

With no documentation to rely on, the first step is to describe the hypothesis and define the mapping. Since there is no documentation to import, the included Axivion editor can be used to draw this first hypothesis.

Then the automated checks are performed to find deviations from the hypothesis in the code.

The third step involves assessing the deviations by prioritizing the architecture violations, making the necessary adjustments and if needed refining the hypothesis.

These steps two and three, the automated checks and assessment of the deviations, are repeated as often as needed until the architecture is accurate.

Once the architecture has been verified as correct, it can be used in the daily architecture checks to prevent architectural drift and thus software erosion.



Common Architectural Pitfalls in CUDA Development

In CUDA source code, logic is divided into host code (running on the CPU) and device code (executing on the GPU). An important concept in CUDA is the kernel, a function executed on the GPU and launched from the host.

Calling a kernel involves specifying grid and block dimensions, parameters that define how threads are launched and distributed across GPU hardware. This invocation protocol must be carefully configured based on the target architecture and the kernel’s logic.

Improper configuration may lead to:

- **poor performance** (e.g., due to underutilized parallelism)
- **runtime errors such as out-of-bounds accesses**

Another key consideration is CUDA’s device memory hierarchy. Shared memory (accessible by all threads within a block) offers lower latency than global memory, but all data transferred from the host to the device initially resides in global memory.

A common pattern is to:

1. transfer data from host to device (global memory)

2. copy it into shared memory at the beginning of a kernel

3. perform computation

4. and copy results back to global memory

A well-designed architecture can support this by enforcing kernel wrapper functions or pre-processing steps (e.g. copyToShared()) before launching compute-intensive logic.

Developers often rely on established CUDA libraries to improve productivity and performance.

These include:

CUDA C++ Core Libraries (CCCL), which provide abstractions and algorithmic utilities similar to the C++ STL

cuBLAS, offering optimized linear algebra operations

cuDNN, designed for deep neural networks and inference tasks

As with any external dependency, it is important to explicitly model the use of these libraries in the project architecture. A common approach is to define clear software layers, such as:

Kernel Layer
Contains device code (CUDA kernels) and may use CCCL utilities.

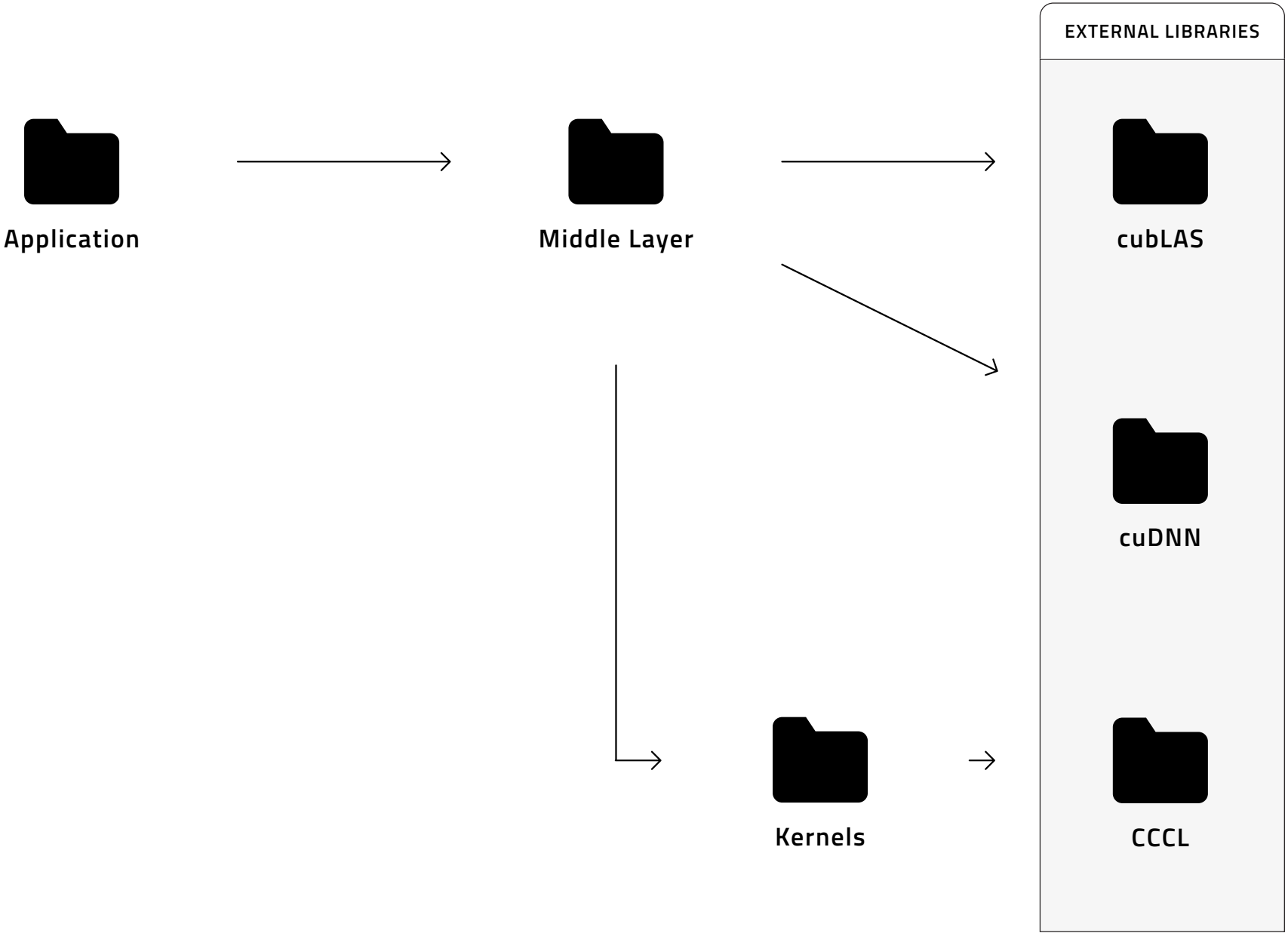
External Libraries Layer
Contains external libraries like CCCL and cuBLAS.

Middle Layer
Offers higher-level computations based on kernels or external libraries like cuBLAS.

Application Layer
Contains business or domain logic and uses only the middle layer.

This separation improves modularity and supports portability across platforms. It can also help simplify testing and maintenance.

HIGH-LEVEL LAYERED ARCHITECTURE FOR A CUDA-BASED PROJECT



4

Software Quality as Economic Advantage

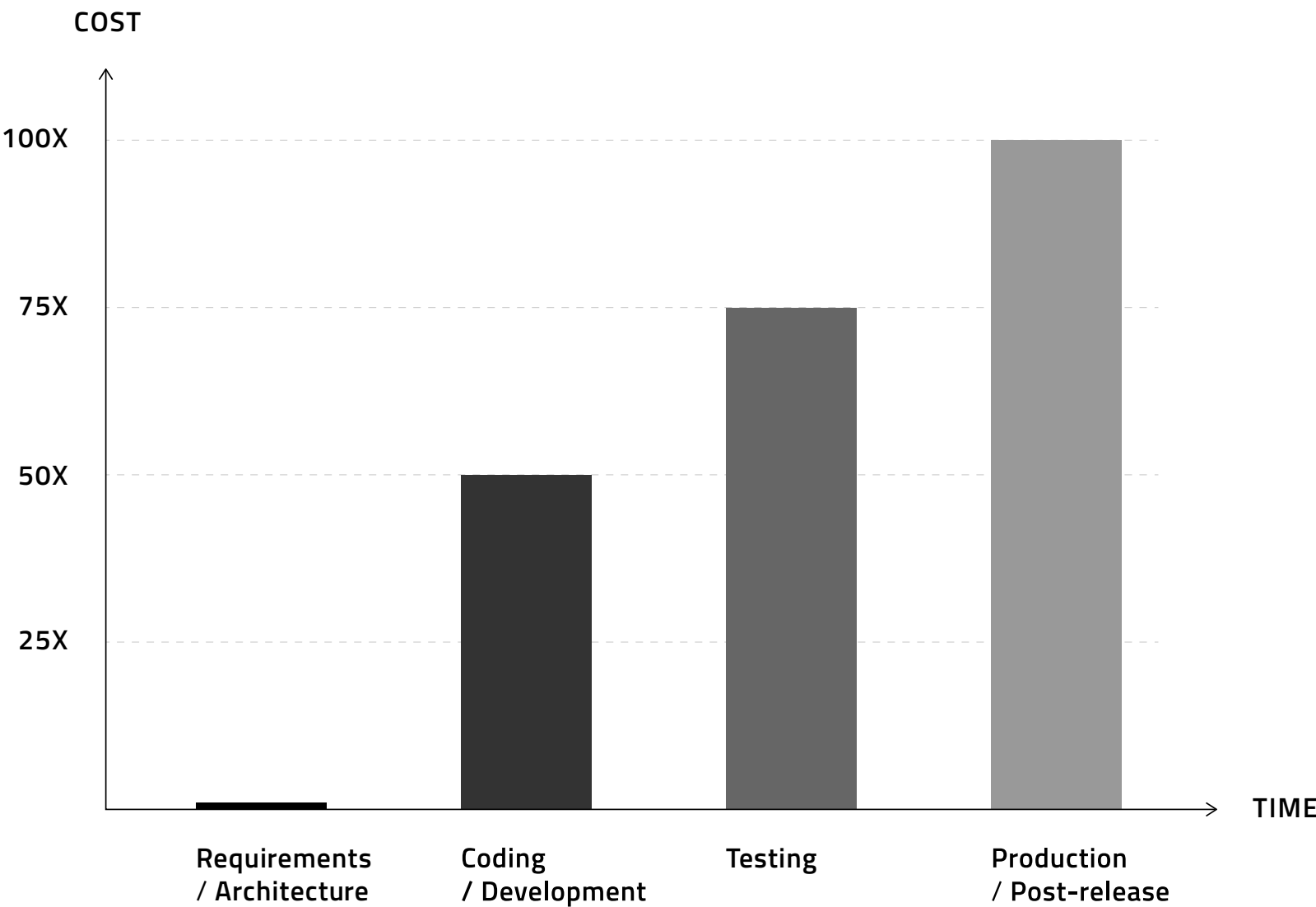
Software Quality as Economic Advantage

Besides improving software quality by tackling software erosion as early as possible (→ Chapter 1), there are significant economic benefits, too.

The expression *time is money* dates back to the 18th century but still holds true over 250 years later. The time needed to understand the architecture or to fix a bug can be converted into the amount you have to pay your employees per hour.

As bugs detected early require less time to fix than those found later during the development cycle or even during post release, they cost less to fix. According to studies, the cost to fix a bug found during implementation is about 6 times higher than if it were caught during design, and up to 100 times more if found in production. This so-called shift-left means using the time to innovate instead of finding and fixing bugs, which is far more productive and economical.

RELATIVE COST TO FIX BUGS, BASED ON TIME OF DETECTION



This can also be applied to understanding the structural aspects of a software’s architecture and software maintenance in general. Reliable software documentation not only helps understand the architecture faster – again, money saved – it also helps plan new features.

Knowing – as opposed to assuming – the effect any changes will have, will ensure there are no unpleasant surprises at a later stage. Noticing that a change made to one part of the code leads to failure of the software in other parts, means hours spent on connecting the two incidents, discussion remedies, implementing and testing these. Meeting deadlines and finishing the project within the set budget becomes impossible; in some cases, the entire project plan may even need to be rewritten.

Having a reliable project plan tight from the start means faster time to market so you can start earning money with your new or improved product. It also gives you a competitive edge if you can hit the market with the latest updates faster than your competitors.

42%

Software
Maintenance

~33%

Increased
Working Time

Improved ROI/TCO: Get More for Less

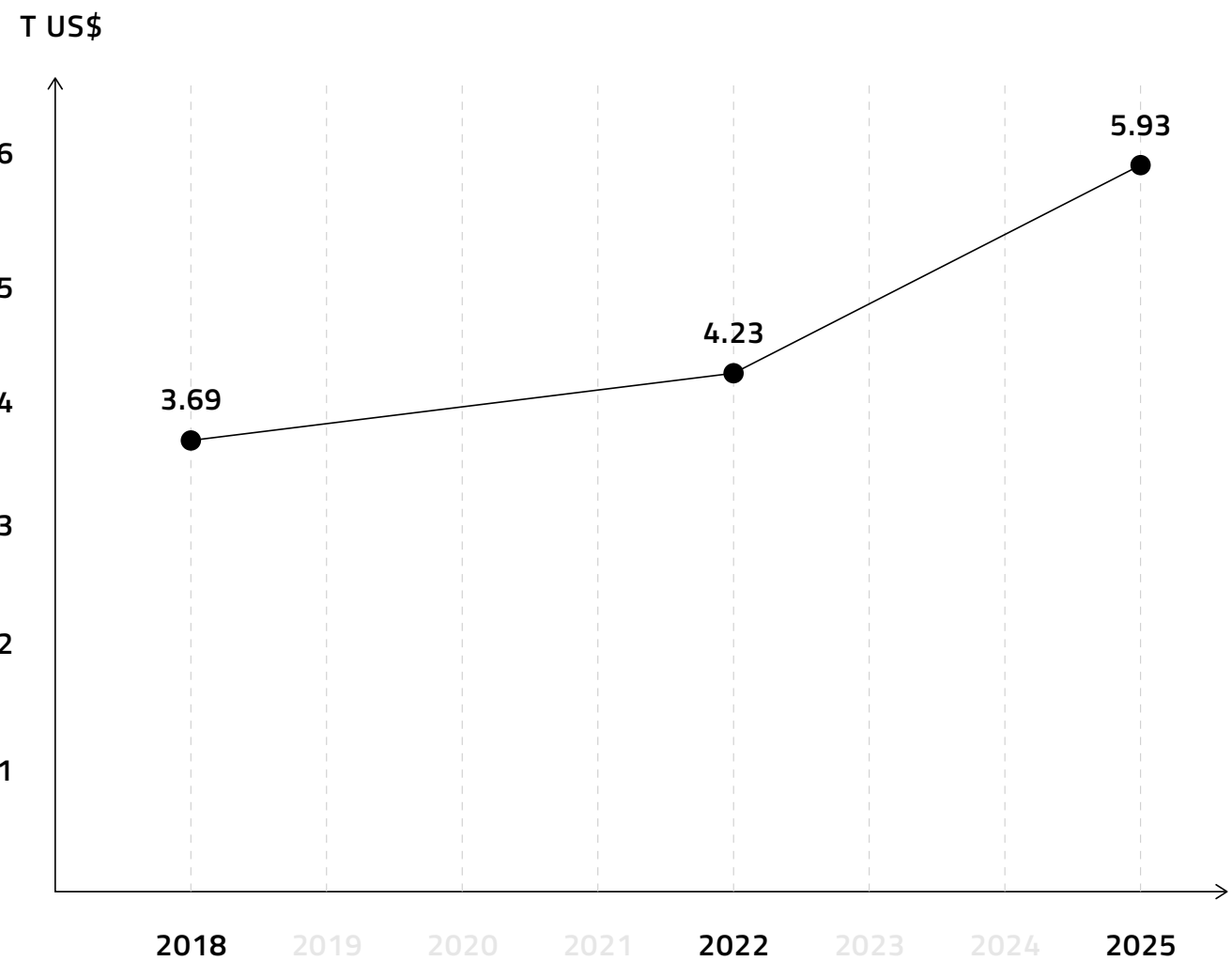
The above shows that investing in software quality improves ROI by ensuring that development resources are used efficiently. High-quality code is easier to maintain, extend, and refactor, which reduces the long-term cost of enhancements and updates.

The total cost of ownership (TCO) is also significantly impacted. Poor quality leads to higher maintenance costs, more frequent outages, and increased support needs. In contrast, quality-focused development results in more stable systems, reducing the need for reactive maintenance and freeing up teams to focus on innovation.

Projecting the numbers provided by the [CISQ Consortium for Information & Software Quality](#) in 2020, the costs resulting from technical debt in the US alone in 2025 would come to ~1.84 trillion USD. Looking at the numbers globally, that figure goes up to \$5.93 trillion!

This clearly shows that ignoring software erosion is something you can’t afford.

DEVELOPMENT 2018–2025



\$5.93 T

Poor software quality cost \$5.93 trillion worldwide.

Make Sure the Best of the Best Work for you

High-quality codebases are easier to understand, modify, and extend. This directly benefits onboarding, as new developers can become productive faster. Developers rank code quality and maintainability among the top factors influencing job satisfaction. Especially in times when finding skilled employees can be challenging, it is important to retain talent.

Working with high-quality code allows developers to spend less time firefighting production issues and more time on meaningful work. This leads to higher retention and a more motivated workforce. Also, companies with strong engineering cultures attract top talent, further reinforcing their competitive edge.

The result: Less disruptions due to constant employee fluctuation and lower HR costs.



Protect What you Have Created

The financial damage caused by a software error can be substantial, often involving direct costs such as lost revenue, compensation claims, regulatory fines, and the expense of fixing the issue itself. In many cases, these consequences are measurable and already severe enough to warrant serious concern. However, the impact doesn't stop there. The reputational fallout from such failures can be even more devastating and far harder to quantify.

Negative press coverage, social media backlash, and word-of-mouth criticism can quickly erode public trust. Once a company's image is tarnished, rebuilding it is a long and uncertain process. Some organizations spend years trying to regain credibility, investing heavily in PR campaigns, customer outreach, and brand rehabilitation. Others never recover at all. The damage can ripple outward: customers may lose confidence and stop buying your products, while business partners and suppliers may reconsider their relationships, terminate contracts, or impose stricter terms.

In extreme cases, the reputational harm can trigger a downward spiral. Reduced sales lead to shrinking revenue, which in turn affects operations, staffing, and innovation. Investors may pull out, and stakeholders may demand leadership changes. What begins as a technical error can escalate into a full-blown crisis, potentially forcing a company to downsize, restructure, or even shut down entirely.

The cost of failure, therefore, is not just financial, it's existential.



5

Where it Matters Most: CUDA Applications in Safety-Critical Environments

CUDA Applications in Safety-Critical Environments

As already mentioned, CUDA's popularity is growing. This can be seen across multiple industries. Starting with the migration of CPU based high-performance computing code to GPU, GPUs today are no longer confined to graphics and high-performance computing or limited to entertainment and convenience. They now serve as accelerators enabling safety-critical systems such as ADAS (Advanced Driver Assistance Systems) and industrial and medical robotics. These are highly regulated industries with multiple, long-standing rules and standards, which need to be complied with.

As CUDA plays a bigger role than ever in these environments, the user community now also needs clear, enforceable coding rules to ensure not only performance and efficiency but reliability and safety as well.

Coding guidelines establish a standardized set of rules that facilitate the creation of maintainable, robust, and predictable code. These guidelines transcend mere stylistic recommendations. The consistency promoted and enforced by the guidelines mitigates misunderstandings among developers, facilitates the onboarding of new team members, and contributes to the sustained quality of code in large or distributed projects.

In the context of CUDA development, where performance and parallelism in particular increase intricacy, guidelines also help avoid subtle bugs that might only manifest at scale or under specific execution conditions.

Software Safety & Security

In software development, safety and security are critical but distinct technical disciplines, each addressing different classes of risks. Their relevance becomes especially pronounced in high-performance computing environments, such as those involving CUDA, where parallel execution on GPUs introduces unique challenges.

In general, **safety** refers to the system's ability to operate without causing unintended harm, particularly in the presence of faults. In technical terms, safety is about fault detection, containment, and recovery. For example, in an application controlling a robotic arm, a race condition or memory access violation could result in physical damage or injury. Safety engineering in such contexts involves:

- Static analysis to detect unsafe memory access.
- Runtime checks for out-of-bound errors.
- Redundancy in computation to validate critical outputs.
- Fail-safe mechanisms, such as watchdog timers or fallback routines.

Security, by contrast, focuses on protecting systems from malicious threats. This includes safeguarding GPU memory, preventing unauthorized kernel launches, and ensuring data confidentiality during computation. Security measures include:

- Memory isolation between host and device.
- Secure APIs to prevent buffer overflows or injection attacks.
- Encryption of data transferred over PCIe or stored in GPU memory.
- Access control to restrict execution privileges.

CUDA's low-level access to hardware makes it a potential target for side-channel attacks, especially in multi-tenant environments like cloud GPU clusters. Attackers might exploit timing variations or shared memory to infer sensitive data. Therefore, secure CUDA programming involves not only correct logic but also hardware-aware threat modeling.

Safety and security often intersect. A security breach—such as unauthorized access to a CUDA kernel controlling a medical device—can directly compromise safety. Conversely, a safety flaw (e.g., unchecked pointer dereference) might be exploited to execute arbitrary code.

So while safety ensures that software does not cause harm due to faults, security ensures it is not exploited to cause harm intentionally. In CUDA-based systems, both require specialized attention due to the complexity and performance-critical nature of GPU computing.

Coding guidelines and industry standards ensure both safety as well as security and it therefore is crucial to ensure that CUDA code complies with them just as much as the C or C++ code in the software.

Coding Guidelines are a Must

Rules serve a purpose. Things have changed: it is no longer sufficient for the QA function to identify bugs but rather developers nowadays are responsible for ensuring the absence of flaws. Additionally, safety must be considered even when the software is operating as intended without faults or failures. Coding guidelines are developed by and for industry experts to support the verification process and serve as an essential tool for achieving the required level of quality and transparency.

There are several well-established coding guideline sets available for C and C++ which form the basis of CUDA development:

MISRA C:2025 / MISRA C++:2023 have their origins in the automotive industry and are widely adopted not only in safety-related software.

CERT C / C++: Focused on security and provide rules to prevent software vulnerabilities.

CWE: Rules derived from the community-developed catalog of common software and hardware security weaknesses



These rulesets vary in scope, but they share the goal of preventing unsafe constructs and improving readability, reliability, maintainability and adaptability. A subtle point when applying the guidelines: developers not only have to consider the current version of the code but also need to think about future changes and how to minimize the risk introduced to seemingly unrelated parts of their systems.

Functional Safety in CUDA Applications

Functional safety refers to the part of a system's overall safety that depends on its correct functioning in response to inputs, including the detection of faults and the execution of appropriate safety measures. It ensures that safety-related systems (like those in automotive, industrial automation, or medical devices) operate correctly even when faults occur.

Functional safety is governed by standards such as:

ISO 26262 (Automotive)

IEC 61508 (General Industry)

IEC 26304 (Medical)

These standards define processes for hazard analysis, risk assessment, safety lifecycle management, and verification/validation of safety mechanisms. They are language-agnostic and therefore also apply to CUDA applications.

Rethinking Functional Safety ↗

Freedom from interference, also known as software segregation, is a key concept within functional safety, especially in systems with mixed criticality, meaning where safety-critical and non-safety-critical software components coexist.

It ensures that:

No unintended influence occurs between software components of different safety levels.

Data is protected from interference.

Safety mechanisms are not compromised by faults or behaviors in unrelated parts of the system.

Demonstrate Freedom from Interference by Using Architecture Analysis ↗

Axivion for CUDA supports the development of software which needs to apply functional safety.

The **Axivion Tool Qualification Kit** supports the classification and qualification of your tool chain. It comprises prefabricated test suites with execution and results evaluation processes that can be automated. This enables you to specifically check and verify the suitability of Axivion in environments with functional safety requirements.

NVIDIA's CUDA C++ Guidelines for Robust and Safety-Critical Programming

Applying coding guidelines when working with NVIDIA CUDA C++ is not just a matter of best practice—it’s a necessity for ensuring robust, secure, and maintainable software. Without clear rules and disciplined coding standards, developers can easily introduce subtle bugs, performance bottlenecks, or vulnerabilities that are difficult to detect and costly to fix.

The guiding principle of safety and security engineering is “if it is available, you have to apply it”. If state-of-the-art methods are not applied, manufacturers may be held liable more easily in case of failure, since the omission of established preventive measures can be interpreted as negligence. This holds true not only for new but also existing projects. This principle reflects a proactive stance: if a safeguard or best practice exists, it should be used —not ignored. Applying the NVIDIA CUDA C++ coding guidelines aligns with this philosophy, ensuring that developers take full advantage of available tools and knowledge to build safer, more reliable. GPU-accelerated systems.

But just like for other coding guidelines it is necessary to check that the code complies with the rules. This is where Axivion for CUDA comes in. Axivion automates the compliance check for NVIDIA’s CUDA C++ Guidelines for Robust and Safety-Critical Programming. This allows developers to apply the new rules with confidence. Not only for current or future applications, but also for older software.

Complying with NVIDIA’s CUDA C++ Guidelines for **Robust and Safety-Critical Programming is easy.**

Product Tour: How to Check NVIDIA's CUDA C++ Guidelines ➤

Other Safety and Security Measures

Axivion also supports other measures associated with detecting vulnerabilities. The defect analysis of Axivion for CUDA checks your source code for potential runtime errors and includes scalable data and control flow checks. This includes:

Buffer Overflow Detection

Buffer overflows are also a safety issue and one of the most common code vulnerabilities. Axivion detects errors that make stack-based and heap-based attacks on the memory possible, allowing you to fix the issues and protect your application.

Race Conditions

Implementing measures to control how and when multiple processes or threads access the shared resources and to ensure only one process can access a resource at a time is just as important as avoiding deadlocks. Axivion for CUDA identifies any irregularities in the code to avoid these race conditions.

Taint Analysis

Axivion can help track the flow of data of particular interest through the program to ensure it is not mishandled. By integrating well-configured taint analyses into the development process, developers can preemptively prevent security risks.

Taint Analysis: Closing the Gaps Before Attackers Find Them ↗

6

Leveraging Code Analysis for CUDA

Leveraging Code Analysis for CUDA

CUDA applications are found in domains requiring massive parallel computation and complex mathematical analysis. These span across various different use cases, including Autonomous Driving, Artificial Intelligence (Deep Learning, Machine Learning), Data Science, scientific computing (climate modeling, computational chemistry, bioinformatics), image and signal processing, financial computing, medical imaging, media and entertainment, and quantum computing to name just a few.

All these areas benefit greatly from CUDA's ability to harness the parallel processing power of NVIDIA GPUs to accelerate computationally intensive tasks that would be too slow on traditional CPUs.

Autonomous Driving (AD/ADAS)

AD/ADAS have highest safety requirements and at the same time benefit from usage of CUDA. Consequently, to be road-worthy, safety-critical code assurance is mandatory. To achieve the same level of compliance as with ISO26262/MISRA C++ for example, code analysis detects logic flaws, undefined behavior, and many more issue types. Architecture verification helps with the integration of individual components and to maintain and exercise control in development teams distributed around the globe.

Artificial Intelligence (AI)

CUDA is fundamental for deep learning and machine learning, enabling rapid training of models by parallelizing the vast matrix and tensor operations involved. With code analysis, model correctness can be improved by eliminating errors during implementation and avoiding e.g. subtle float truncation bugs. In addition, memory and security issues can be addressed early, for example out-of-bounds indexing or buffer mishandling. Finally, applying code analysis enforces consistent development pipelines across teams. As code bases tend to become large and distributed, architecture verification helps to fence in dependencies and avoid architecture violations.

Scientific Simulations & High-Performance Computing (HPC)	Media & Entertainment	Data Science & Analytics
<p>Climate & Weather Modeling: Simulating complex climate systems benefits from the parallelization offered by GPUs.</p>	<p>High-performance tasks in content creation and processing, like rendering and video editing, can be accelerated. Code analysis gives hints and aid for detection of inefficiencies or problems with memory handling. Since the applications are often multi-platform, both code analysis and architecture verification help enforcing good coding practice to keep the application cross-platform and flawless.</p>	<p>Complex data analysis, numerical computations, and other data-intensive tasks see significant speed-ups, helping firms make better, faster decisions. Code analysis for CUDA helps developers to find incorrect type usage, incorrect type conversions and identifies missing null-handling in data digestion code. If configured correctly, it can also give performance guidance by identifying for example deeply nested loops. At the same time, architecture verification enables the projects to keep the high-level overview of the code base and keep maintainability at the best possible level.</p>
<p>Computational Chemistry: Tasks like protein docking and molecular dynamics simulations are accelerated, leading to faster discovery and analysis.</p>		
<p>Bioinformatics: Processing genomic data, sequencing, and gene simulations are intensive tasks where CUDA provides performance gains.</p>		
	Image & Signal Processing	Financial Computing
<p>In this domain, code analysis for CUDA contributes to thread safety and race detection by analyzing the massively parallel code constructs for race conditions and erroneous memory access. For example, by checking the correct use of data types and conversion, it can identify potential precision and data loss during low-level operations. Architecture Verification gives the software architects a powerful means of exercising control of the ever-growing application code base.</p>	<p>Real-time processing and reconstruction of images in medical imaging (MRI, CT) and signal processing are significantly faster with CUDA. By means of boundary checks and overflow analysis, code analysis makes sure that unsafe use of data is avoided. It can also flag unintended integer division or rounding errors that can lead to distorted results. Architecture Verification helps with keeping consistent processing pipelines.</p>	<p>CUDA accelerates complex risk analysis and other financial calculations, allowing for real-time data analytics crucial for trading and investment decisions. Code analysis helps developers to preserve precision and rounding control in their code and can raise alerts if mixed float and integer types are used. Architecture verification ensures the usage and provision of the correct API and the compliance of the code with its intended architectural structure.</p>

Want to Future-Proof Your CUDA Applications?

[Contact us ↗](#) to discover how your organization can master software quality, compliance and innovation at the same time.

Learn More

Visit our Resource Center

Request a Proof of Value Workshop

