woven by TOYOTA

LLVM in the Automotive Industry

Bringing Functional Safety to Open Source

June 10th, 2025

Wendi Urribarri

Functional Safety Engineer
Global Safety & Quality LoB
wendi.urribarri@woven.toyota

woven by TOYOTA

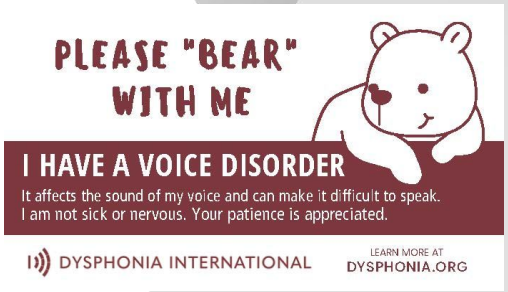
LLVM in the Automotive Industry

Bringing Functional Safety to Open Source

June 10th, 2025

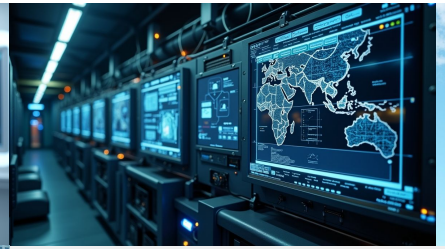
Wendi Urribarri

Functional Safety Engineer
Global Safety & Quality LoB
wendi.urribarri@woven.toyota



Safety-critical industries

Safety-critical industries

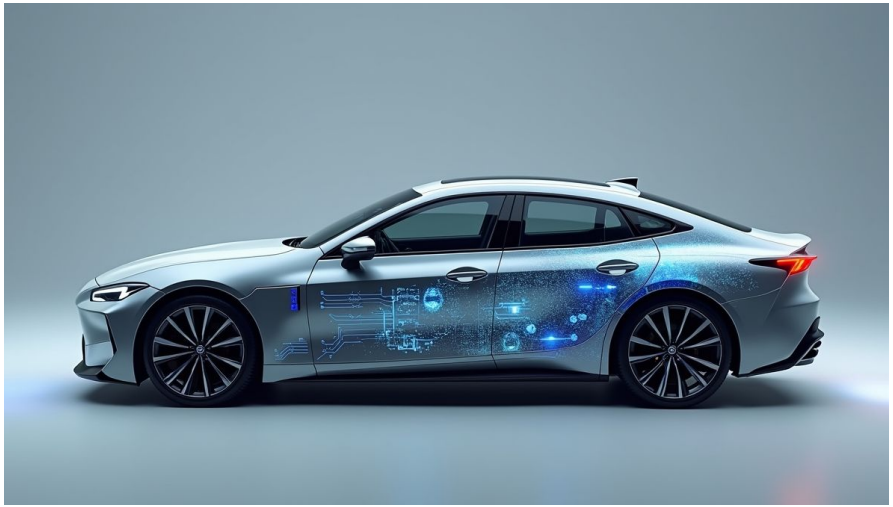




Certain failures could cause serious harm, injuries, damage, or loss of life





Automotive Industry



e.g. Advanced Driver Assistance Systems (ADAS), Autonomous Driving (AD), Software Defined Vehicle (SDV)

Increase of **software components** in road vehicles

Increase in significance of **Functional Safety**

Functional Safety

Making sure systems behave *safely* even when something goes wrong, by *preventing*, *detecting*, *handling* e.g.,

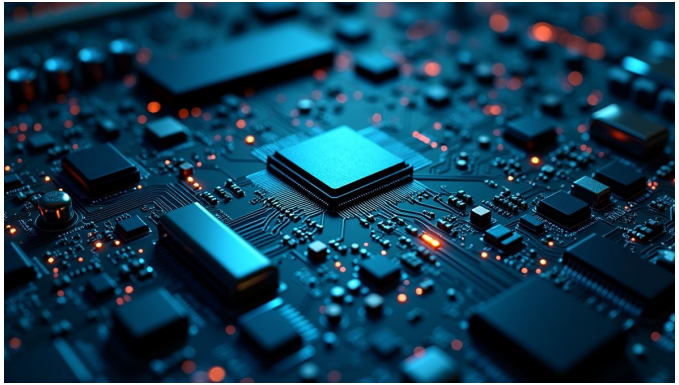
- Hardware defects
  - Steering angle sensor stuck at zero → Steering locked while driving
- Software bugs
  - Buffer overflow in the perception stack → Incorrect detection or classification of other vehicles and pedestrians

Safety Standards

Rigorous development processes

Prescribed requirements, guidelines, practices to ensure safety at each level (systems, hardware, software)

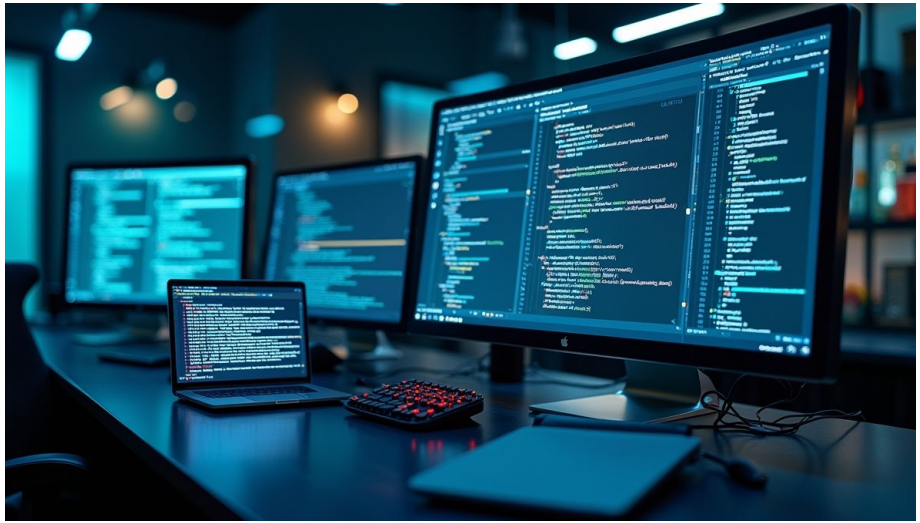
Sub-processes: specification, design, implementation, testing, safety analyses, configuration, calibration, tracking of changes, versioning, documentation, release…



Functional Safety standard in the automotive industry:
ISO 26262

Software development tools are part of the safety equation because they support functional safety

Software tools can introduce or fail to detect errors, and undermine even the best designed system



Confidence in the use of software tools

- **Follow a structured, evidence-based process for tool development**
- **Evaluate the level of “confidence” that the software tool can be used “properly” in a safety-related context**
- **When higher confidence is necessary, ensure its reliability and correct functioning**

1 - Tool Definition — Identification and specification

- Intended usage/Use cases
- Inputs and Outputs
- Functions/Features
- Execution environment
- Configuration options
- Expected behavior under anomalous conditions
- Known Issues and their workarounds/countermeasures
- Versioning and Change tracking
- Pre-determined max ASIL of usage

2 - Tool Evaluation

Identify:

- Potential malfunctions and their impact
- Measures that can be applied to prevent or detect them
- Determine the *Tool Confidence Level*

“Simplified dysfunctional analysis”

3 - Tool Qualification

When tool confidence level is insufficient, provide QA evidence of the tool itself

This can involve:

- Introducing new prevention, detection, or error handling mechanisms inside the tool
  - e.g., adding an option to use Alive2 to analyse and verify LLVM code and transformations
- Checks:
  - Verification and Validation
  - Development process assessment

4 - Means for verification of appropriate usage

Provide means to the final users for verification of appropriate usage of the tool

Summary of use constraints, conditions, restrictions, limitations, resulting from previous steps

Usually given in a manual

Can we create a shared, open qualification path for LLVM that benefits both industry and community?

Envision the possibility

Could we explore the interest in qualifying LLVM-based compilers upstream?

If so, could this evolve into a meaningful, collaborative effort, useful and understandable to LLVM contributors, companies, and tool vendors, beyond just safety compliance?

Challenges

01
Evolving compiler infrastructure

LLVM is large, evolving, and highly configurable

02
Missing structured quality evidence

OSS workflows and quality evidence would need to become more comprehensive, documented

03
Fragmented qualification landscape

Today, most tool qualification is proprietary, expensive, and duplicated across companies

Many vendors have independently and privately qualified their LLVM-based compilers

Vision of a Safety Group

Complementary role to that of LLVM's Security Group, focused on systematic assurance and tool reliability, e.g.,

- Participate in quality of each release
- Discuss safety-relevant issues (e.g., risk analysis, known bugs analysis)
- Provide specifications
- Identify missing tests and resolve the gaps
- Encourage and review upstream changes that could aid qualification

Vision of an Open Qualification

A collaborative effort to qualifying LLVM-based compilers in open source, scaling up incrementally, e.g.,

- Develop reusable quality arguments, test suites, docs…
- Shared validation & audit strategies
- Documentation & Traceability

Precedents exist: Linux kernel safety efforts by ELISA project & Linux Foundation, CompCert, Ferrocene…

Qualification isn’t just about passing audits. It’s about building deeper trust in the compiler’s behavior.

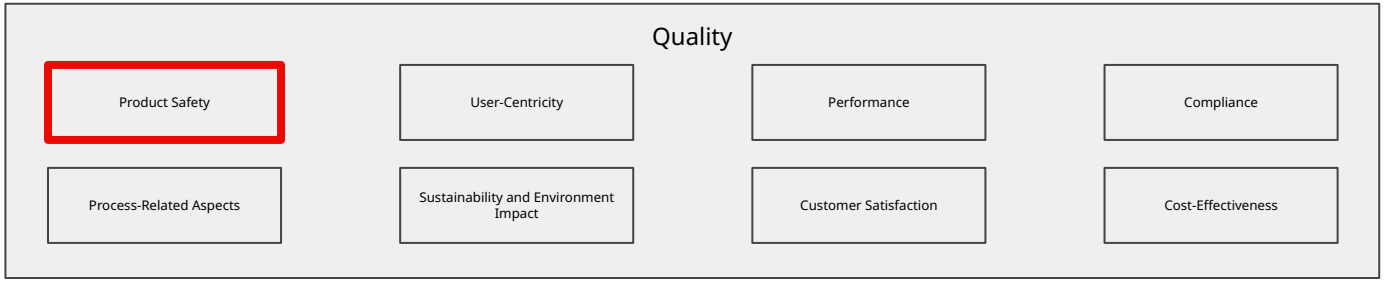
The same practices that help us meet compliance to safety standards can make LLVM more reliable, transparent, and robust for all users.

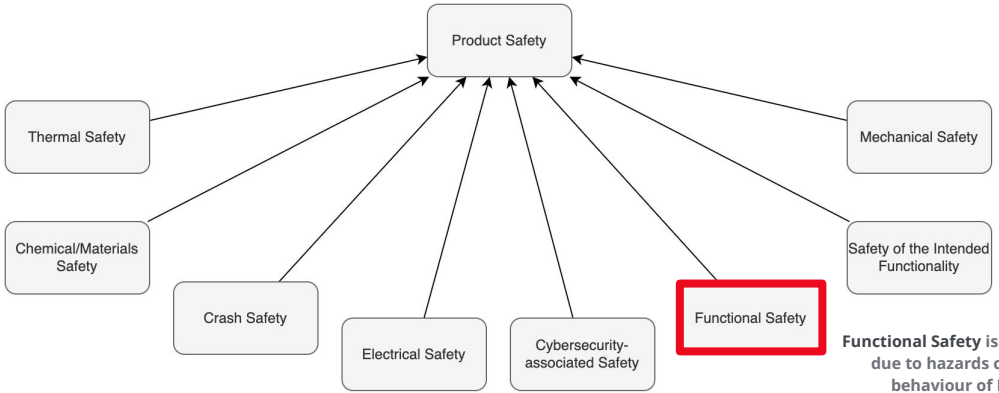
Thank you

My email: **wendi.urribarri@woven.toyota**

My LinkedIn profile: **www.linkedin.com/in/uwendi**

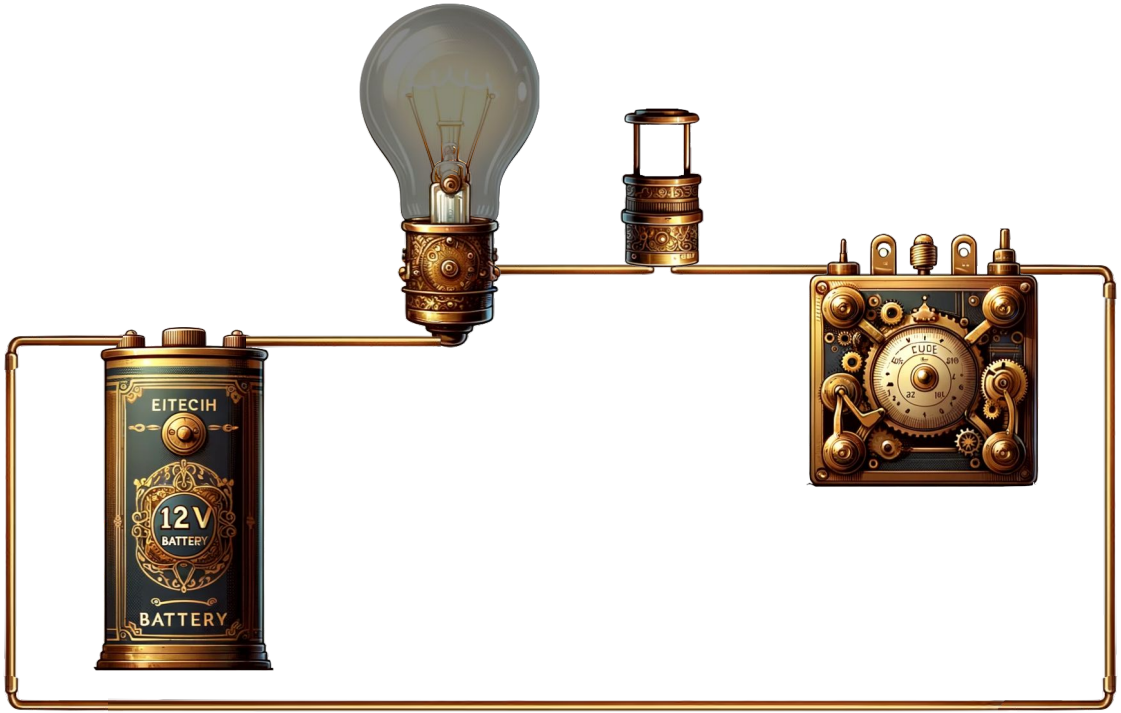
Quality, Product Safety, and Functional Safety



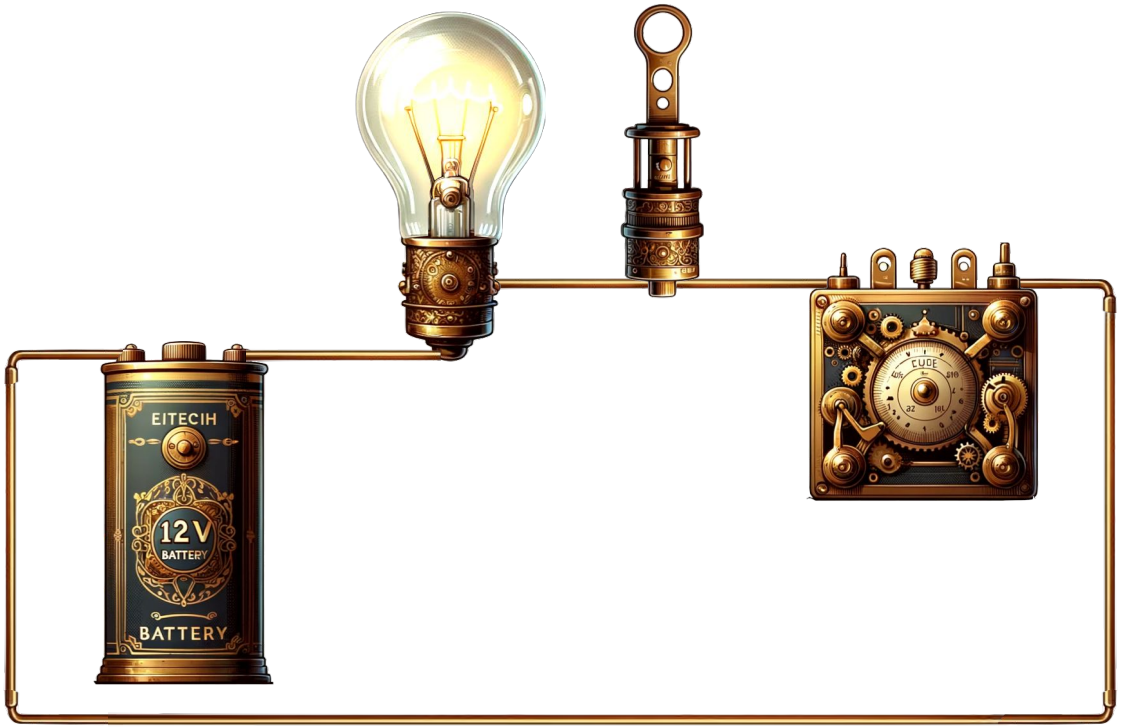


Functional Safety is **absence of *unreasonable risk*** due to hazards caused by malfunctioning behaviour of Electrical, Electronic or Programmable Electronic **(E/EE/PE) systems**

Concepts and ideas in ISO 26262

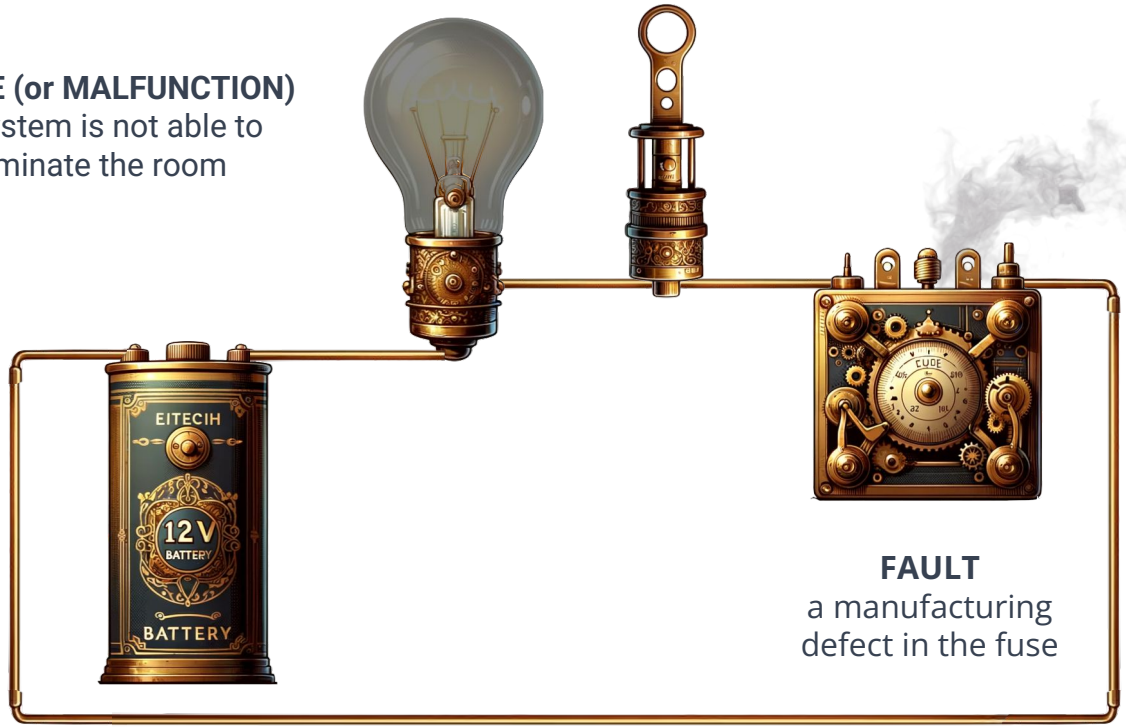


Concepts and ideas in ISO 26262



Concepts and ideas in ISO 26262

FAILURE (or MALFUNCTION)
the system is not able to illuminate the room



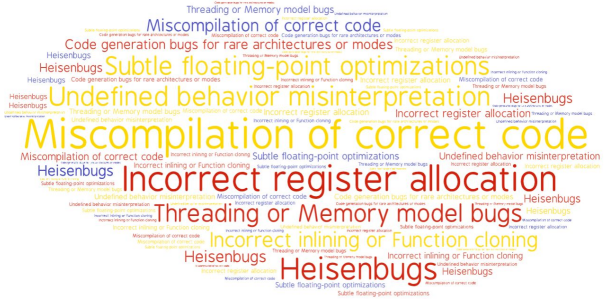
ERROR
electricity is not conducted properly

FAULT
a manufacturing defect in the fuse

Types of failures

Random	Appear in the hardware only. They occur due to the aging of HW parts.
Systematic	Appear in both software and hardware. They occur due to errors in design or production and are typically able to be reproduced.

Confidence in the use of compilers



Compilers can introduce bugs

AND

Some kinds of bugs can be hard to detect by only testing the output file

Coding guidelines do not protect from compiler bugs.
The compiler must be trustworthy.

Intended usage/Use cases

Usage scenarios inscribed in the intended purpose

- High level, general descriptions of the ways in which a user can interact with the software tool
- Written descriptions of “usage scenarios”, “situations” in which the tool may be useful, or “interactions” between the user and the software tool
- The “tasks” that the users can perform with it

A use case outlines the software tool's expected behavior and outputs as it responds to a “request” and inputs from the user's point of view

Inputs and Outputs

I/O can vary widely depending on the tool’s purpose and functionality, e.g.:

Inputs	Expected outputs
• Data inputs: raw data files, databases, streams... that the tool processes • User input through forms, CLI, GUI... • Configuration settings: parameters or options that customize how the tool operates • Commands or instructions: specific commands, queries, or scripts that direct the tool to perform certain tasks • External resources: libraries, APIs, or external services that the tool might access during its operation • License or authentication: credentials or keys that allow access to secure features or services	• Processed data: transformed, analyzed, or summarized data that reflects the tool's processing activities on the inputs • Reports or logs: detailed output in the form of reports, summaries, or logs indicating what actions were performed and any results • Alerts or notifications: messages or alerts signifying errors, updates, or completion of specific tasks • Visualizations: graphical representations of data or results, such as charts, graphs, dashboards... • Exported files or formats: files in specific formats generated as part of the tool's output

Functions/Features

- Features: specific attributes or capabilities that a software tool offers (often what attract users to a product)
- Functions: operations or tasks that a software tool can perform (core activities that the tool is designed to execute)
- Technical properties: underlying technical characteristics of the software tool that affect its performance and compatibility

Execution environment

Context in which the software tool operates, such as a specific minimum version of an operating system or a database management system

Configuration options

Parameters that modify the behavior of the software tool

- They can alter the fundamental ways in which the software tool works, affect the qualities of its performance, and potentially change outputs even when the same inputs are used
- They are helpful for customizing the tool’s usage to suit individual users, groups, or projects
- They typically come with predetermined “types”, and “default values” or “default arguments”
- In some scenarios, a software tool may not be configurable by the end user, but rather by an administrator or an individual with special privileges

Expected behaviour under anomalous conditions

Normal expected functioning of the tool in abnormal conditions

This includes the generation and display of:

- Error Messages: specific notifications indicating the occurrence of errors or malfunctions within the system
- Warning Messages: alerts that notify users of potential issues or deviations from expected performance, allowing for preventive measures
- Diagnostic Codes: coded information used to identify and diagnose the root cause of errors or malfunctions, facilitating troubleshooting and corrective actions

They ensure that the software tool maintains a manageable operation, even in adverse situations, by informing users of discrepancies and guiding necessary interventions

Known issues

Resources for identifying reported abnormal functioning of the software tool under normal conditions (e.g., bugs)

- Type: classification of the issue, aiding in systematic tracking and resolution
- Description: explanation of the issue
- Status: current state of the issue (e.g., open, in progress, resolved)
- Release: tool version in which the issue was identified
- Severity: evaluation of the issue's potential impact on the system, hardware or software under development, or on the users, prioritizing resolution efforts
- Workarounds and/or Countermeasures: suggested solutions or temporary fixes to mitigate the issue until a permanent resolution is implemented

Versioning and Change tracking

Practices to manage and document updates, modifications, and iterations of software throughout its lifecycle

- Versioning: systematic assignment of unique identifiers (versions) to different iterations of the software
- Change tracking: documenting all modifications made to the software, including code changes, configuration adjustments, feature additions…

Pre-determined max ASIL of usage

Indicates the highest maximum "*Automotive Safety Integrity Level*" (ASIL) rating for the requirements allocated to the item or element that the tool is expected to support in its development

- It denotes the tool's capacity to maintain safety requirements across varying levels of criticality, thus enabling the reliable development of automotive solutions across a spectrum of safety concerns
- ASIL spans from ASIL A (basic level of safety integrity) to ASIL D (demanding the highest safety performance)

Software tool requirements

Specific expectations, criteria, and conditions that the software tool must meet to effectively perform its intended functions and satisfy user needs

They serve as a blueprint for the development, verification, and validation of the tool

Tool Confidence Level (TCL)

Measure of how much trust we need to place in a software tool used to develop safety-critical systems

- TCL1, TCL2, TCL3
- Reflects the level of assurance required to use the tool without introducing or missing dangerous defects
- It tells us how much evidence or rigor we need before we can rely on the tool
- The higher the TCL required, the more we need to show the tool works reliably and won’t cause undetected issues
- It helps us decide whether a tool can be used as is, or if it needs extra qualification

Simplified dysfunctional analysis

Lightweight version of a traditional hazard or failure modes analysis. We focus on the tool use cases and ask:

- *What could go wrong for the given use case?* (e.g., a miscompilation, a missed warning)
- *What impact would that have on the final system?* (e.g., incorrect behavior in a safety-critical feature)
- *Can we reduce or manage the risk?* (e.g., by a manual check of the inputs or the outputs)

The goal is NOT to prove that the tool is safe, but to understand and manage how defects in the tool could lead to defects in the product

Safety Group

Examples of what the group could do:

- Curate best practices for deterministic use, configuration, testing
- Participate in quality of each release
- Provide specifications for different parts, and tests of their behavior
- Identify missing tests and resolve the gaps
- Discuss known safety-relevant issues (e.g., known bugs, categories, and impacts)
- Align on a scope for future qualification efforts (e.g., a subset of LLVM, only certain frontends, certain targets)
- Encourage and review upstream changes that could aid qualification
- Serve as a contact point for tool vendors doing work related to safety compliance, e.g, to ISO 26262, EN 50128, etc.

Open Qualification

Many benefits, among others:

- Shared effort = Shared benefits
- Transparency = Better trust and reuse
- More robust compiler ecosystem
- Lower entry barriers to safety domains
- Cross-industry alignment

Useful beyond safety - Examples

01
Reinforced confidence

Detailed documentation of LLVM's quality, such as specifications, traceable assurance of how code is transformed (e.g., reqs ↔ tests), analysis of known limitations, etc.

02
Augmented bug detection

Catch hard-to-find, subtle bugs or regressions before they affect end users, by using innovative verification methods such as formal verification (e.g. Alive2), added test suites for LLVM IR, etc.

03
Viable in other high-assurance contexts

Expand LLVM's reach and impact, allowing it to compete with proprietary compilers in high-assurance contexts