

S U R E S O F T
Intelligence for Safe World

CT — AI-Powered Test Generation

Structure-based & measurable · Embedded C · Requirements-based · Coding-agent native

Suresofttech · H1 2026

Four gaps CT's AI closes in verification

AI layered on top of CT's precise code analysis — at a level teams actually use in the field. Delivered two ways: **CT ALIRA AI** (built into CT) and **CT DVERA** (inside your coding agent).

① Structure-based generation

Precise analysis + multi-agent → high statement coverage

82–98% of a frontier model even on an air-gapped, self-hosted LLM — proven quantitatively.

② Embedded C

C · C++ support, tuned for C

Supports both C and C++, with dedicated tuning for **embedded C** — the case AI finds hardest.

③ Requirements & spec-based

V-SPICE · requirements traceability

Starts from requirements/specs, not code → **requirement↔test traceability** (certification evidence).

④ Coding-agent integration

6 agents native · CT DVERA

Across **6 coding agents**, uses development context to **generate, run, and review tests** directly.

Structure-based generation — quantitative proof via precise analysis + multi-agent

Clear, provable numbers = structural (code-based) coverage · powered by CT ALIRA AI

Technical pillar 1

Precise code-analysis grounding

CT injects its precise analysis — prototypes, types, symbols, call graphs — straight into the agent → **targets uncovered paths** for generation.

Information a pure AI agent simply doesn't have → high, stable statement coverage.

Technical pillar 2

Multi-agent orchestration

Planning, generation, fixing, and analysis are split across **four specialist agents** · stage transitions are governed by the engine (deterministic code), not the LLM.

→ Reaches frontier-level statement coverage even on a **mid-size self-hosted LLM**.

Self-hosted LLM = 82–98% of frontier-model statement coverage

Runs on air-gapped, in-house LLMs with no source code leaving your network

4 open-source projects — a self-hosted LLM approaches the frontier model

Values = statement coverage (%) · CT baseline / frontier LLM / self-hosted LLM = absolute, self-hosted÷frontier = ratio

Language	Project	Complexity	CT baseline	Frontier LLM	Self-hosted LLM	Self-hosted÷Frontier
C	cJSON	Med	48	77.4	68.6	88.6%
C	openAUTOSAR	High	38	75.4	69.2	91.8%
C++	tinyxml2	Low	59	97.7	94.7	96.9%
C++	ninja	High	29	82.2	67.8	82.5%

Frontier LLM = the top-performing model (measured: Sonnet 4.6, cloud-served) · Self-hosted LLM = a 27B-class on-prem open model (measured: Qwen 3.6-27B) · statement coverage · as of 2026.06.

Customer production code – C, proven at 97.9% of frontier

Applied to a real customer project · self-hosted absolute coverage 81.6% = 97.9% of frontier (83.3%) · values = statement coverage (%)

Customer	Language	Complexity	CT baseline	Frontier LLM	Self-hosted LLM	Self-hosted÷Frontier
Automotive parts supplier (Company A)	C	Med	64.9	83.3	81.6	97.9%

Proven live through an automotive OEM's in-house LLM gateway – CT's AI test generation was verified running in the customer's own environment via their internal LLM gateway. Concrete evidence for air-gapped, in-house-LLM deployments.

CT baseline 64.9 → 81.6 with AI (self-hosted)

97.9% of frontier – minimal loss in an air-gapped setting

Embedded C specialization — a C-dedicated agent

Safety and embedded still run on C — yet C is actually harder for AI

The challenge

No classes — must infer stubs and linkage

Global symbols and headers are tangled, and correct stubs require inferring the **build/link structure**. Macros, function pointers, void*, global-variable initialization — many C-specific hurdles.

AI training and benchmarks are C++-centric, so C is comparatively under-validated.

The solution

AUTOSAR-based tuning + C-dedicated agent

A **C-dedicated generation workflow** redesigned on the automotive embedded standard (AUTOSAR Classic), plus a dedicated agent for C (macros).

Built-in embedded-C corrections: void*, global-variable tracking, stubs, nested structs.

Two C projects (cJSON · automotive-embedded openAUTOSAR) proven quantitatively on a self-hosted model — approaching the frontier model

Requirements- & spec-based generation — through to certification traceability

Two generation paths that start from requirements/specs, not code

01

Requirements-based test generation

Auto-generate tests for a target function via requirement **@-mentions**, attached spec files, or slash commands · auto-links **requirement↔test traceability** (certification evidence).

02

V-SPIICE spec-based generation

Combines V-SPIICE function specs + test-case (TC) values with CT's analysis · AI **maps values to code symbols** · eliminates manual CSV import.

AI Auto-Fix — one flow from generation to maintenance

Source changes, compiles fail — AI handles it. AI-generated tests are fixed by AI · powered by CT ALIRA AI

01

Auto-fix compile errors

A dedicated agent in the generation loop analyzes and auto-fixes compile failures — retrying in a **Closed-loop**.

02

Follows source changes

When function names, parameters, or globals change, it **updates only the affected tests in bulk** — asset continuity, not regeneration.

03

Practical outputs

CSV export of test data · stub support · no loss of data or results when code changes.

CT DVERA — a verification agent inside your coding agent

Call CT directly from 6 coding agents · native MCP + Skill

6 coding agents

Tailored to each agent

Claude Code · Cursor · Codex · Windsurf · GitHub Copilot · Gemini — connected the way each agent expects (MCP, Claude Code/Codex CLI Skill), with **no code changes**.

21 CT MCP tools

From setup to query, end to end

From **CT's initial environment setup and project creation** through analysis, test generation, execution, and coverage query — **the whole flow, in natural language**.

What sets it apart

Leverages development context

Captures and uses the coding agent's **development context for verification** · structure- and requirements-based generation.

Deployment

From air-gapped on-prem to coding agents

On-prem · air-gapped

Runs without source leaving your network

Run AI test generation on in-house / self-hosted LLMs with **no source code leaving your network**.

LLM integration

Bring Your Own Model (BYOM)

Direct connectors for OpenAI · Azure OpenAI · Anthropic · Google, plus **LiteLLM compatibility** for any in-house / self-hosted LLM · **even models outside the list via custom integration**.

Coding agents

Native connection · natural-language

Use CT in natural language from **6 agents (Claude Code · Codex, etc.)** via each agent's native method (MCP · Skill).

Metric: statement coverage · Self-hosted LLM = 27B-class on-prem open model (measured: Qwen 3.6-27B) · Frontier LLM = top-performing model (measured: Sonnet 4.6, as of 2026.06).

SURESOFT

Intelligence for Safe World

Thank You