

CT — AI-Powered Test Generation

CT layers AI on top of a deterministic test engine that analyzes code precisely, automating unit/integration test generation and coverage. It runs in air-gapped, in-house-LLM environments with no source code leaving your network, and can be driven directly from your coding agent — delivered as **CT ALIRA AI** (built into CT) and **CT DVERA** (inside your coding agent).

The core, provably measurable number is **structural (code-based) coverage**. All performance figures below are on a **statement-coverage** basis.

Key capabilities

① Structure-based AI generation — quantitative proof

CT injects its precise code analysis (prototypes, types, symbols, call graphs) straight into the agent to target uncovered paths, and splits planning, generation, fixing, and analysis across specialist agents (multi-agent orchestration). Stage transitions are governed by the engine (deterministic code), not the LLM — so even a mid-size self-hosted LLM reaches **82–98% statement coverage** vs. a frontier model.

② Requirements- & spec-based generation · traceability

Generates tests starting from requirements/specs (e.g., V-SPICE) rather than code, binding requirement IDs to inputs and expected values and linking them into a **requirement↔test traceability** report (certification evidence).

③ CT DVERA — coding-agent verification agent

Attaches natively to 6 coding agents — Claude Code · Cursor · Codex · Windsurf · GitHub Copilot · Gemini — and drives CT's analysis, test generation, execution, and coverage query in natural language via each agent's native method (21 MCP tools, Claude Code/Codex CLI Skills). It leverages the coding agent's development context for verification.

④ C · C++ support · embedded C specialization

CT supports both C and C++. C — the mainstay of safety and embedded — is actually harder for AI because it has no class boundaries (global symbols, macros, function pointers, void*, inferring the build/link structure); CT builds in embedded-C corrections via an AUTOSAR-based, C-dedicated generation workflow and a C-dedicated agent.

⑤ AI Auto-Fix — from generation through maintenance

A dedicated agent analyzes and auto-fixes compile failures (retrying in a **Closed-loop**); when the source changes, it updates only the affected tests in bulk.

Performance data

Values = statement coverage (%). Frontier LLM = top-performing model (measured: Sonnet 4.6); self-hosted LLM = 27B-class on-prem open model.

Open-source benchmarks (publicly verifiable)

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%

Customer production code

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%

The self-hosted LLM approaches the frontier model — little performance loss even in an air-gapped setting.

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

Deployment

On-prem · air-gapped — run AI test generation on in-house / self-hosted LLMs with no source code leaving your network

LLM integration (BYOM · Bring Your Own Model) — direct connectors for 4 clouds (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 — use CT in natural language from 6 agents via each agent's native method (MCP · Skill)