s²:innovation
Software solutions

AI-driven Tango Device Generator

12 months later. What changed, what we learned, what's next.

Łukasz Żytniak
on behalf of the S2Innovation Team

s²innovation
Software solutions

AI-driven Tango device server generator
The goal of the project

Web-based application that **automates the generation** of Tango Controls device servers using Large Language Models (LLMs).

This application **surpasses the functionality of existing tools** (like POGO) by also implementing device-specific logic based on documentation and not just generating templates.

The popular AI tools are only as good as the person using it.

s²:nnovation
Software solutions

The new architecture

RAG pipeline

Semantic search

Extract

Docling parses the PDF (OCR when needed).
A HybridChunker splits it into chunks of up to 1024 tokens.

Embed & index

Each chunk is embedded with BAAI/bge-m3 and stored in a FAISS vector index for fast similarity search.

Retrieve & generate

Adaptive retrieval (similarity threshold plus percentage top-k) feeds the best snippets into the prompt → device-server code.

AI-driven Tango device server generator – Model comparison

	GPT-4o	GPT-5-mini (new)	Gemini 3 Flash (new)	Claude 3 Opus
Code quality	Very high	High, fast	Good	Good
Cost	$$	$	$	$$$
Context	128k	400k	1M	200k
Best for	Production	Quick iteration	Long manuals	Complex logic

The system is model-agnostics. New models are added through configuration, not by changing code.

s²:innovation
Software solutions

From proof-of-concept to deployable application

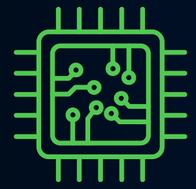
- **Authentication** - Keycloak OIDC in production, or dev mode
- **Project persistence** - MongoDB stores chat history, parameters and files per project
- **Async file processing** - the UI shows progress and never blocks
- **Auto-generated API client** - frontend built from the OpenAPI spec
- **Multi-user ready** - session isolation per project and user

Real-world tests

- **Original five:** Danfysik 8000, Metrolab PT2026, HPLC pump / PI 872, Eurotherm 3508, Andor Newton CCD
- **New from integration tests:** Keithley 2400 and Lakeshore
- **~20% time saving** was the baseline last year
- **Target ~34%** with automatic context extraction, for devices that ship PDF documentation

s²:innovation
Software solutions

What's next



Open-source LLMs

Auto test generation

Migration of C++ / Java device servers

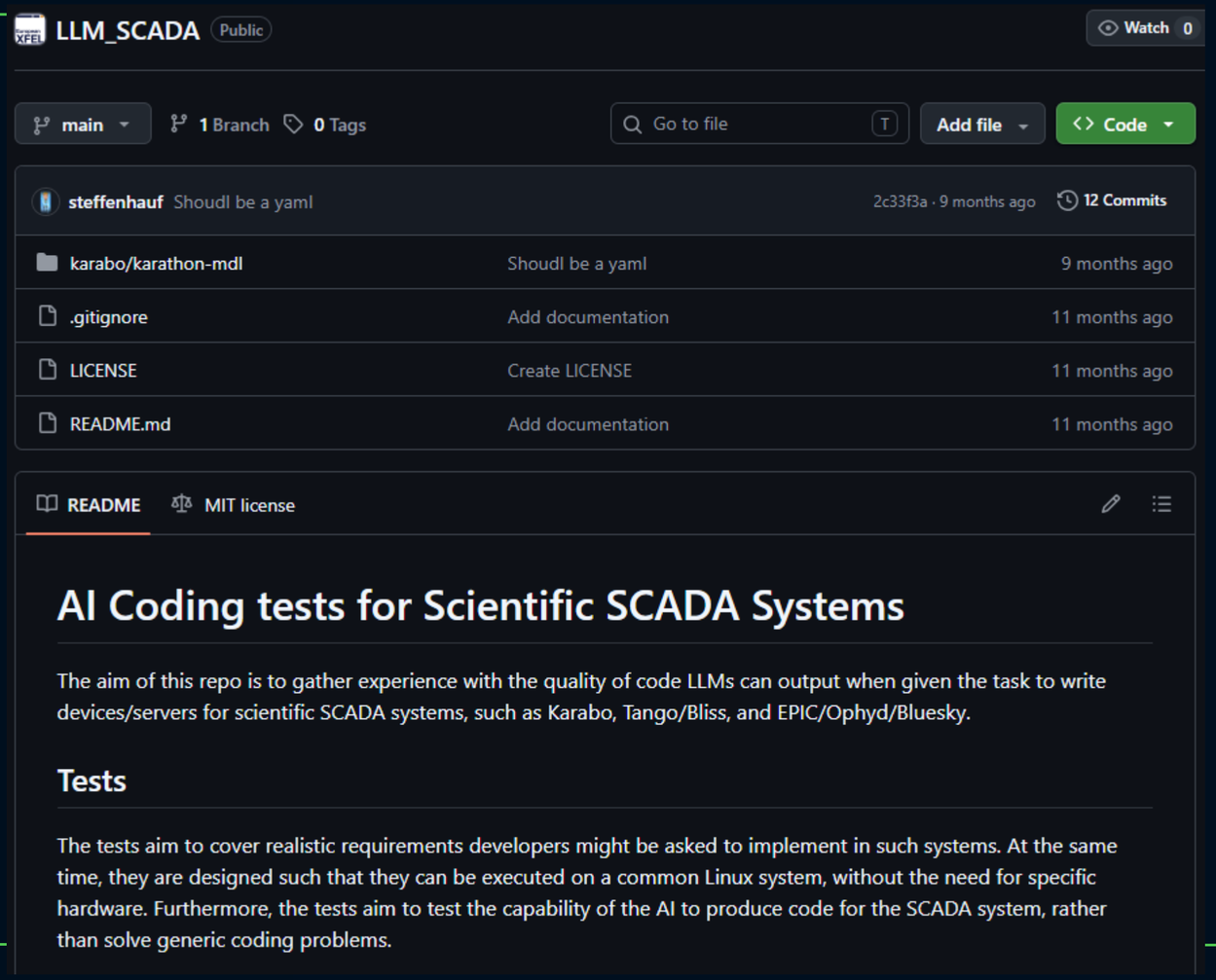
Per-register indexing

We are looking for partners for further testing.

EuXFEL Benchmarking AI for Scientific SCADA Code

LLM_SCADA Analysis for AI Coding Tests in Scientific SCADA Systems is primarily a **benchmark and evaluation methodology**. Its purpose is to test how well LLMs generate scientific code, how much repair effort is required, and where models fail on framework-specific details. The initial tests were done in Karabo.

AI-Driven Device Driver Generator is a **product-oriented engineering tool**. Its purpose is to help engineers generate working Tango Controls device servers from a structured device definition and optional device documentation, using LLMs and RAG.



EuXFEL Benchmarking AI for Scientific SCADA Code

DS-Generator (S2Innovation)
Our Tango Device Generator powered by GPT-5-mini.

GPT-5-x (direct, CHAT GPT)
Same underlying model with no Tango specialization.

Three scenarios, increasing complexity

1 Monitoring Device

CPU & memory of the device's own process

2 Coordinated Motion Scan

Two motors, optimal path, full state machine

3 Image Processing

Center-of-mass, crosshair imprint, image pipeline

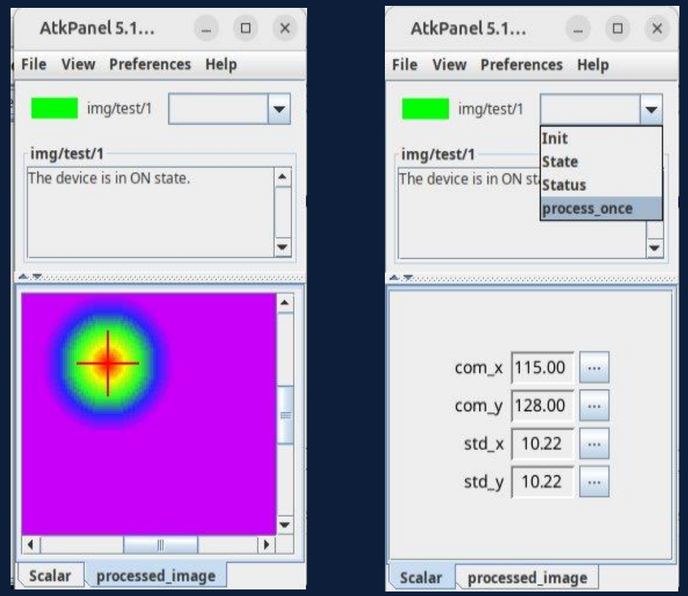
Measured: lines changed to reach a running, tested device server, plus test coverage %.

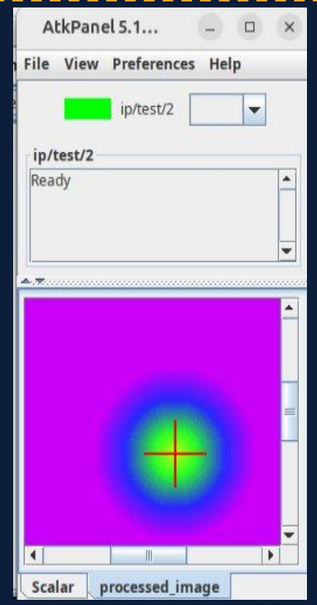
Both tools run the same prompt, verbatim the difference is the Tango-specific context DS-Generator provides.

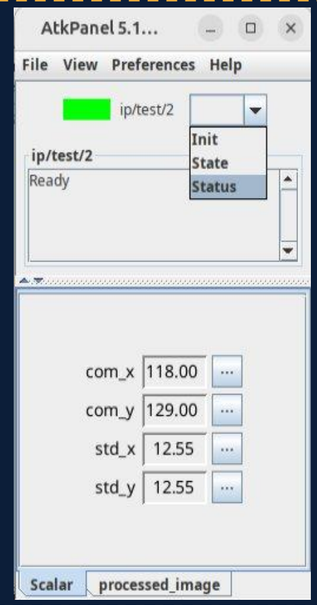
RESULTS

SCENARIO	MODEL	ADD. ITERATIONS	LINES FIXED (DEVICE)	LINES FIXED (TESTS)	TEST COVERAGE
1. Monitoring	DS-Generator	0	+3 / -2	+1 / -1	**100%**
	GPT 5.x	**0**	**0 / 0**	**0 / 0**	93%
2. Coordinated Motion	DS-Generator	0	+3 / -3	+8 / -8	86%
	GPT 5.x	1	+4 / -4	+2 / -2	**93%**
3. Image Processing	DS-Generator	1	**0 / 0**	**0 / 0**	**100%**
	GPT 5.x	1	+4 / -5	0 / 0	72%

Complexity of Fixes Matters, Not Just Line Count







DS-Generator · Scenario 3

0 lines changed

- Clean run after a single correction iteration.
- Correct Tango idioms: DevState.ON, property placement in MultiDeviceTestContext, fresh image fetch per attribute read.
- **100% test coverage out of the box.**

GPT-5 direct · Scenario 3

4 + 5 lines changed

- COM bias bug: intensities clipped to zero before the weighted moments center drifts off the true Gaussian peak.
- dformat=AttrDataFormat.IMAGE dropped on attributes silent failures in the test context.
- Coverage 72% happy path covered, error paths missed.
- Scenario 2: motor names hardcoded (not device properties); nearest-neighbour path replaced wholesale with Held-Karp DP; test rewritten.

What This Means in Practice

Where DS-Generator wins

- Tango-specific correctness state handling, attribute declarations, property placement.
- Higher coverage in the harder scenarios 100% vs 72% on image processing.
- Code closer to what a Tango developer would write and review.

Where GPT-5 is competitive

- Scenario 1 (simple monitoring): raw GPT-5 produced a perfect device, zero fixes.
- For simple, stateless devices, direct GPT-5 is a viable starting point.

Honest caveats

- Scenario 2 still required 8 + 8 lines of adjustment, not zero effort.
- One correction iteration for image processing (same as GPT-5).
- The advantage grows with Tango complexity; for simple devices the gap narrows.

Bottom line – DS-Generator produces code that is correct by Tango conventions, with fewer structural surprises. GPT-5 direct is fast for simple cases but introduces domain-specific bugs that need a Tango expert to catch. For scientific SCADA, where correctness matters and hardware is expensive that difference is significant.

Update: Since gpt-5.3 and onwards EuXFEL usually achieve single shot coding for the Karabo Middlelayer API

s²innovation
Software solutions

Early stage of automating test generation for PyTango device servers

The problem:

- Control systems run hundreds of PyTango device servers, each talking to real hardware (serial, TCP, vendor drivers)
- Testing them means mocking that hardware so tests run offline
- In practice test coverage stays low

The goal:

- A tool that takes an existing device server as input and produces a working pytest test suite - hardware mocked, runs offline - keeping the human in the loop

s²:innovation
Software solutions

The approach

1. Device source
2. Analysis
3. Device profile
4. Test planning
5. AI-assisted generation
6. Validated pytest suit

- **Analysis** - extract the device structure from the source: attributes, commands, properties...
- **Device profile** - a structured model of the device's public interface, used as the single source of truth. Grounds every downstream step in what the device actually exposes
- **Test planning** - determine what to verify: happy paths, edge cases, error handling
- **AI-assisted generation** - a LLM creates the tests using DeviceTestContext, mocking all hardware, guided by the device profile and curated examples
- **Validation at every step** - generated code is automatically checked (syntax, static analysis, execution) → failures feed a correction loop

Current State

What works today:

1. End-to-end pipeline on a real reference device.
2. Automatic extraction of attributes, commands, properties and state machine
3. Tests run offline with hardware mocked, and pass automated quality and syntax checks
4. Failing tests are repaired automatically within a bounded correction loop

s²:innovation
Software solutions

Roadmap

Where it's going:

1. **Measuring test quality, not just pass/fail** - add code coverage and mutation testing
2. **A critic in the loop** - an agent reviews generated tests against the device interface, flags missing cases and weak assertions
3. **Human-in-the-loop checkpoints** - expose the test plan for review before code is developed, and flag low-confidence tests for inspection

s²innovation
Software solutions

Early stage of automating the migration of Tango Controls device servers

The problem:

- Control systems run hundreds of legacy C++/Java Tango device drivers

The goal:

- A tool that takes an existing device driver as input and produces a working PyTango equivalent - keeping the human in the loop

The approach

1. C++/Java source
2. Analysis
3. Intermediate Representation
4. Code generation
5. AI-assisted completion
6. PyTango driver

- **Analysis** - extract the device structure: attributes, commands, properties, state machine
- **Intermediate Representation** - a language-neutral model of the device. Decouples the source language from the target, so new inputs or outputs can be added independently
- **Generation** - develop a PyTango skeleton with the correct structure
- **AI-assisted completion** - a LLM fills in the implementation, guided by the original source and the device model.
- **Validation at every step** - generated code is automatically checked (syntax, static analysis, execution)

Current State

What works today:

1. End-to-end pipeline on a real reference device
2. Automatic extraction of attributes, commands, properties and state machine
3. Generated drivers pass automated quality and syntax checks

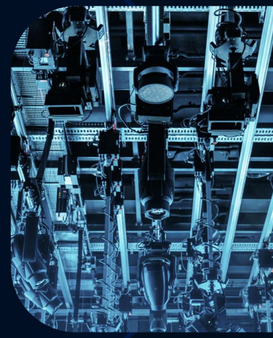
Roadmap

Where it's going:

1. **Knowledge-guided generation** – feed the model with PyTango examples for higher-quality output
2. **Multi-level correctness checks** – from syntax, through static analysis, to running the device in a test context
3. **Confidence flagging** – the tool marks uncertain code for human review instead of guessing silently

Thank You

For Your Attention



Łukasz Żytniak
+48 789 339 875
lukasz.zytniak@s2innovation.com

s²innovation
Software solutions

AI-driven Tango Device Server Generator