ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-WEPD060

AI-DRIVEN DEVICE DRIVER GENERATOR*

K. Fugiel, K. Klimczyk, Sz. Kupiecki, T. Madej, L. Zytniak†
S2Innovation, Krakow, Poland

Abstract

Developing device drivers for control systems can be a time-consuming task, often requiring specialists to write boilerplate code and implement device-specific logic by hand. In frameworks like TANGO Controls, tools such as POGO exist to auto-generate skeleton code for new devices, but these produce only generic templates that still demand significant manual coding to implement the actual device logic. The rise of large language models (LLMs) presents an opportunity to automate and accelerate this process. Recent work has shown that LLMs like GPT-4 can produce working code for instrument control by interpreting natural language descriptions or documentation. In the embedded systems domain, for example, new AI-driven tools can convert hardware datasheets into digital models and automatically generate low-level driver code, greatly reducing the need for hand-coding and speeding up development. Building on these advances, we introduce an AI-driven device driver generator that integrates LLMs into the development of TANGO device servers. This approach aims to go beyond template generation by using device documentation to implement device-specific functionality, thereby producing more complete drivers with minimal human intervention.

RELATED WORK AND EXISTING SOLUTIONS

Traditional code generation tools for control system drivers have largely been template-based. In the TANGO Controls ecosystem, the POGO code generator provides a graphical interface for defining a device class model and outputs a device server skeleton in languages like C++, Java or Python [1]. POGO simplifies initial class creation but does not fill in the device-specific behaviour – developers must manually code the logic for each attribute and command based on the device's manual. Similarly, in other environments (e.g. EPICS or vendor-specific frameworks), one can find wizards or templates to scaffold driver code, yet the heavy lifting of implementing protocol specifics remains manual.

More recently, attention has turned to using AI to assist or automate driver development. Developers have begun experimenting with general-purpose AI coding assistants (e.g. OpenAI's Codex/ChatGPT or Anthropic Claude) to generate driver code from datasheets or problem descriptions. In one reported case, an engineer used an AI assistant to help write a QNX driver for a thermal sensor, finding that the AI could suggest register operations and fix errors through iterative prompts [2, 3]. While such ad-hoc use of chatbots shows promise, it still requires the developer to guide the AI and verify the code through multiple rounds of Q&A.

Dedicated solutions are emerging to make this process more systematic. Embedd.it, for example, has demonstrated a tool that parses component datasheets into a structured digital twin model and then generates production-ready driver libraries "at the click of a button," using deterministic code generation to avoid AI hallucinations [4]. This approach has been shown to quickly produce drivers for sensors and peripherals and even supports frameworks like Zephyr RTOS, substantially accelerating R&D workflows [5]. Likewise, researchers in materials science reported using ChatGPT-4 to create a complete control software module (including a GUI) for a Keithley 2400 SourceMeter instrument solely from the natural-language instrument manual, with only minimal human corrections needed [6]. These examples underline a trend: by leveraging AI with domain knowledge (e.g. device documentation), one can automate much of the coding for device control.

However, a gap remains in applying such AI-driven code generation specifically to TANGO Controls device servers, which have a particular structure (attributes, commands, properties) and rely on high-level device classes rather than low-level register tweaking. The tool presented in this paper addresses this gap by marrying the template-based convenience of existing tools with the intelligence of LLMs to implement device-specific logic.

BENEFITS OF AN AI-ASSISTED DRIVER GENERATOR

Adopting an AI-driven device driver generator yields several benefits over traditional development or template-only approaches:

- Accelerated Development: By automatically writing the device-specific code, the tool can shorten the driver development cycle from weeks to hours. Early tests showed that for devices like power supplies and teslameters, the generated TANGO server code required only minor edits before it was functional, indicating a dramatic speed-up in getting a working driver. This allows faster integration of new instruments into the control system.
- Reduced Manual Coding and Errors: Automating boilerplate and parsing device documentation spares developers from writing repetitive code and reduces the chance of human error. The generator reads the official device manual (if provided) and uses it to produce correct command sequences and data conversions, which means fewer omissions or mistakes compared to manually coding from a spec sheet. It effectively acts as an expert assistant that doesn't "forget"

* Work supported by S2Innovation
† lukasz.zytniak@s2innovation.com

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-WEPD060

implementation details spelled out in the documentation.

- Ease of Use through GUI: The presence of an intuitive web-based GUI means even non-experts can define a new device server by filling out forms for attributes and commands, without dealing with development environment setup. This lowers the barrier to entry for adding devices. There is no need to install a heavy IDE or even the POGO application; everything is accessible via a browser. The tool thus broadens participation in driver development to domain scientists or engineers who may not be seasoned programmers.
- Multi-Model and Open AI Support: The backend can leverage multiple LLMs (such as OpenAI GPT-4, Anthropic Claude, Google Gemini, etc.) [3]. In practice this means the generation can be performed with different AI engines to improve reliability or suit various deployment preferences (including potential use of open-source models on-premises). The ability to easily switch or compare models ensures that the best or most compliant code result can be chosen, and it future-proofs the system as new models become available.

In summary, the AI-driven generator combines the strengths of template-based tools and AI's ability to digest natural language, yielding faster development with fewer mistakes and a more accessible workflow.

AI-DRIVEN DEVICE DRIVER GENERATOR: SYSTEM OVERVIEW

System Architecture

The solution presented by S2Innovation is a web-based application that automates the creation of TANGO device servers using LLMs [2]. The user interacts with a front-end graphical interface to specify the device class details. This includes defining the device's Attributes (readable/writable parameters exposed by the device), Commands (actions or procedures the device server can execute on the hardware), and Properties (configuration values). The interface is designed to be user-friendly, often using form fields and dropdown menus, so that the user can describe the device in a structured way without writing any code. In addition, the user has the option to upload the device's documentation (e.g. a PDF manual or datasheet) into the system.

Behind the scenes, the application uses a retrieval-augmented generation (RAG) pipeline to incorporate documentation knowledge. When the documentation PDF is provided, the system parses it and indexes it in a vector database. Relevant sections of the manual – for instance, the description of each command or register – are retrieved based on the context of the driver being generated [2]. These relevant text snippets are then fed into the prompt for the LLM alongside a predefined code generation prompt template. In essence, the LLM is instructed on the general task of creating a TANGO device class (with proper class structure, attribute read/write methods, command methods, etc.) and is given the specific device information (from the GUI input) plus related documentation excerpts. This allows the AI to produce code that not only has the correct shape of a TANGO device server but also includes device-specific logic (like correct command syntax, unit conversions, ranges, error messages from the manual, etc.).

Technologies Used

The backend of the application is built with FastAPI (a Python web framework) and uses LangChain to manage interactions with multiple LLM providers [3]. FastAPI handles the web server aspect, serving the GUI and API endpoints, while LangChain provides an abstraction to easily call different LLMs and handle prompt assembly, history, and post-processing. The use of LangChain enables integration with popular large-language-model APIs (such as OpenAI GPT-4) as well as the flexibility to include local or open-source models in the future. The system currently supports models like GPT-4, Claude 2, and Google's Gemini 1.5, among others [3]. This multi-model support was actually one of the design objectives – to evaluate and compare the quality of code generated by different LLMs and not be tied to a single vendor. The generated code is typically output in Python, leveraging the PyTango library, since Python is widely used for TANGO device servers and allows quick iteration. The choice of Python also aligns with the goal of accessibility (many scientists are familiar with Python).

Once the LLM returns the generated code, the system may apply some post-processing or validation. For example, it can run a syntax check or minimal static analysis on the code. In the future, a CatScan-like tool could be integrated to catch common issues. The output is then presented to the user, who can review the code in the browser. The user can copy the code or download it as a module, ready to be deployed as a TANGO device server. In tests with real devices, the generated code has been able to run with only minor modifications – e.g. small corrections in command names or adjustments in logic – demonstrating the effectiveness of the approach.

Example Use Case

Suppose we need a device server for a programmable power supply. Using the tool, we would input the attributes (like voltage, current readings, setpoints) and commands (e.g. enable/disable output, set certain modes) via the GUI. We attach the manufacturer's PDF manual for that power supply. The generator will extract details such as the SCPI commands or protocol bytes needed to read the voltage, the valid range of current, error codes, etc., from the manual. It then produces a Python class inheriting from PyTango. Device, with all attributes and commands defined. The read/write methods for attributes and the command implementations will include calls to the power supply using the correct syntax (for instance, sending the SCPI command "VOLT 5" to set voltage). Internally, if the manual indicated any required conversion (say the device expects a value in tenths of volts), the code will include that math. This is a stark improvement over a tool like POGO, which would have only generated empty method stubs – here the

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-WEPD060

methods are populated with working code. In short, the AI-driven generator can read the manual so you don't have to. It effectively encodes the manual into the driver software.

Importantly, the tool also addresses some limitations observed in initial trials. One issue with LLM-generated code is that it can occasionally misinterpret or hallucinate commands if the retrieval step misses relevant info or if the prompt is ambiguous. In our application, this manifested as occasional command mismatches (e.g. using an incorrect command keyword). We mitigate this by refining the prompt and ensuring the retrieval grabs sufficient context (for example, always including any command reference tables from the documentation). Another limitation was handling complex devices where the documentation is large – we found the need for more advanced document parsing (for example, combining information spread across different sections of a long manual). These insights are guiding ongoing improvements to the system's prompt engineering and retrieval algorithms.

Despite these challenges, the prototype has clearly demonstrated its value: even for fairly complex instruments, the bulk of the device server code can be generated in one shot, with the developer only doing light touch-ups. This suggests a paradigm shift in how control system software can be developed, with AI tools taking over the heavy coding and humans focusing on verification and high-level design.

ROADMAP AND FUTURE WORK

The AI-driven device driver generator is an evolving project. We are thinking of several extensions to broaden its capabilities and usefulness.

Automated Unit Test Generation

A planned feature is to have the tool generate a suite of unit tests alongside the driver code (Fig. 1). These tests would validate the behaviour of the generated device server (for example, simulating device responses to ensure the parsing logic in read methods is correct, or checking that commands produce expected state changes). An automated test generator would increase confidence in the AI-produced code and catch any mistakes early. Given the structured nature of TANGO and the availability of the device's specification, the system can create dummy device stubs or use known good values from the documentation to form test cases. This addition would move the tool closer to a one-click solution for not only code but also its verification.

Digital Twin Generator

We plan to extend the application to create a digital twin of the device – essentially, an emulator or simulation model of the hardware. A digital twin would allow developers and users to run the device server without the physical device, which is invaluable for testing in CI pipelines or developing higher-level control logic before hardware is available. This feature builds on the same documentation parsing: the system can generate a simulated device class that mimics the real device's responses (using timing constraints, state machines, and data from the manual). The concept of deriving digital twins from documentation is already gaining traction in the industry [4], as it enables rapid prototyping and hardware-in-the-loop testing. By incorporating this, our tool would not only write the driver but also a virtual device that speaks the same protocol – effectively closing the loop for a full virtual testing environment.

Cross-Language Driver Conversion

Currently, the tool outputs Python code, but many existing device servers (and developer preferences) are in C++ or Java. On the roadmap is a driver conversion capability, where an existing driver implemented in C++ or Java can be ingested (along with its device description) and the tool will produce an equivalent Python implementation (or later vice versa). This could be done by using the LLM to understand the logic of the original code and re-create it in the target language, while ensuring it adheres to TANGO patterns. Such a feature would be highly useful for migrating legacy codebases to newer languages or unifying a facility's software stack. It also demonstrates the generality of the approach – since the core logic is derived from device behaviour, it should be reproducible in any supported language. Multi-language support was already envisioned in the initial design, and this will bring it to fruition.

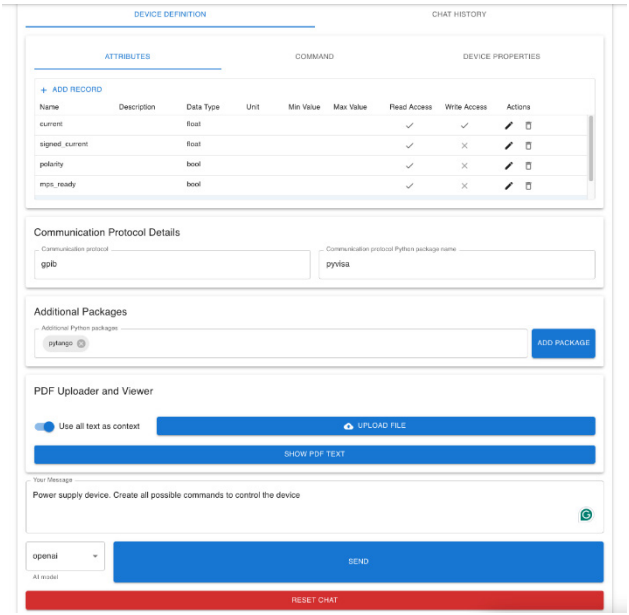

Figure 2: Web interface of the AI-driven Tango device driver generator.

Open-source LLMs

Furthermore, we are exploring the integration of open-source LLMs to alleviate concerns about cloud API costs and data confidentiality. Models like Llama2 or other locally hostable LLMs, possibly fine-tuned on code, could be plugged into the system so that institutes can run the generator entirely on-premise. This goes hand in hand with seeking improvements in the retrieval component (for example, using more advanced embeddings or even fine-tuning a model on a corpus of device manuals to better specialize it in the language of technical documentation).

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-WEPD060

Source Code Quality Metrics

Finally, an important aspect of future work is the rigorous evaluation of the generated code quality. We plan to quantitatively measure how much manual correction is needed for various device types, and to track improvements as the system evolves. Community feedback will be essential here – as more developers use the tool for different devices, we will gather data on its performance and any failure modes.

CONCLUSION

The AI-driven device driver generator represents a significant step towards automating control system software development. By leveraging LLMs within a structured framework, it can produce functional device server code faster and with less human effort than traditional methods. Our application essentially bridges the gap between a device's documentation and a ready-to-use driver, which has been a long-standing dream in the field of accelerator and experiment controls. Early results are promising, and as we enhance the system with testing and simulation capabilities, it stands to become an indispensable tool for developers.

We also recognize that this project opens many avenues for academic inquiry, from improving AI understanding of technical texts to formal verification of AI-written code. S2Innovation is actively seeking a scientific partner to collaborate on the next phases of this work. Such a partnership could explore advanced topics like co-simulation of AI-generated twins with real hardware or training specialized models on open data from control systems. We believe that with combined industry and research efforts, the vision of fully AI-assisted development for device drivers can be realized, benefiting the entire community.

REFERENCES

[1] TANGO POGO Code Generator – Pogo User's Guide, https://tango-controls.readthedocs.io/en/latest/tools/pogo.html

[2] Embedd.it – AI for Driver Development , https://www.electropages.com/the-hub/video/revolutionising-embedded-development-ai-driven-driver-code-generation

[3] D. M. Fébba *et al*., "From Text to Test: AI-Generated Control Software for Materials Science Instruments," Jun. 2024, arXiv:2406.16224. doi:10.48550/arXiv:2406.16224

[4] K. Klimczyk *et al*., "Automation of Device Server creation in TANGO Controls using Large Language Models," presented at the 5th ICFA Beam Dynamics Mini-Workshop on Machine Learning for Particle Accelerators, Geneva, Switzerland, Apr. 2025, contribution 29, unpublished.

[5] "AI-driven Tango device driver generator," presented at the 39th Tango Community meeting at INAF, Giulianova, Province of Teramo, Italy, May 2025. https://indico.tango-controls.org/event/422/contributions/888/attachments/600/811/2025-05-22%20S2I%20AI-driven%20Tango%20device%20driver%20generator.pdf

[6] Software Integration and Testing of Chips, https://embedd.it/.