
GRENOBLE | MODANE

# Leveraging Python & LLMs in Control-Command Systems

A new era for system maintainability ?

Thomas GEMOND

# The Dual Challenge: Complexity & Fragmentation

## Engineering Hurdles

- **Complexity Trap:** C-C systems are inherently complex, blending real-time logic, safety-critical operations, and decades of legacy code. This creates a high-stakes environment where small changes can have cascading, unpredictable effects.
- **Maintainability Bottleneck:** Plagued by obscure legacy logic, "documentation drift" where docs no longer match code, and critical "tribal knowledge" held by a few senior engineers.

## Organizational Hurdles

- **Reinventing the Wheel:** Multiple labs solve identical problems (logging, simulation) from scratch with incompatible tools.
- **Deep Knowledge Silos:** Solutions are often lab-specific and poorly documented, preventing the sharing of best practices or lessons learned.
- **Wasted Resources:** Fragmentation leads to significant duplicated effort, higher maintenance loads, and slower innovation across the board.

# Building Consistent Local Ecosystems

## Standardization Starts Locally

- **Unify tools and practices** within your own teams.
- Reduce internal fragmentation, speed up onboarding, and simplify local maintenance.
- Create a reusable 'toolkit' for your lab, making new projects faster to deploy.

## Mitigating High Turnover

- Complex, bespoke ecosystems create a steep learning curve, hindering newcomers.
- Standardized practices leverage common market skills.
- Allows new hires to become effective immediately, mitigating turnover impact.

# The Candidate: Why Python?

*Python provides the mature, flexible foundation required for this common ecosystem.*

## The Candidate: Why Python?

### The Ultimate "Glue"

Python acts as a universal binding layer. It wraps high-performance C/C++ drivers, allowing you to control low-level hardware with high-level, readable syntax.

### High-Level Sequencing

It excels at managing state machines and complex workflows. It governs the *sequence* of operations, delegating real-time execution to the device layer.

### The Data Bridge

It is the only language that natively connects industrial hardware (OPC-UA, Serial) directly to modern data science stacks (NumPy, PyTorch) and web APIs.

# Python's Role: Orchestrator vs. Real-Time

## What Python IS For (Supervisory)

- **Orchestration:** Managing high-level sequences and state machines (e.g., experimental plans, state machines).
- **Data Handling:** Logging, databases, and cloud connectivity (e.g., Pandas data processing).
- **User Interfaces:** HMIs and operator dashboards (e.g., PyDM).
- **Soft Real-Time:** Tasks where timing > 10–20ms is acceptable (e.g., slower monitoring loops).

## What Python IS NOT For (Control)

- **Hard Real-Time:** Sub-millisecond safety loops.
- **Deterministic Control:** The Garbage Collector (GC) creates unpredictable jitter.
- **Safety Interlocks:** Never use Python for personnel or machine safety.
- **Motion Control:** High-speed servo loops (use FPGA/PLC).

# Where Python is Deployed

## 1. In Scientific Research (The "Framework" Model)

Facilities are replacing legacy, inflexible systems (e.g., Diamond's Java-based GDA, ESRF's SPEC) with modern, Python-based frameworks.

**Key Frameworks:**

- **Bluesky / Ophyd (NSLS-II, Diamond):** The dominant open-source stack for experiment orchestration. Ophyd provides a crucial hardware abstraction layer, and Bluesky runs the "plans".
- **BLISS (ESRF):** A complete, Python-native control system chosen to unify experiment control with the scientific Python ecosystem (NumPy, SciPy).
- **Pytac (Diamond):** A high-level library that provides a physics-oriented interface to the accelerator, abstracting the complex EPICS backend.

## 2. In Industry (The "Integration & AI" Model)

**Embedded Scripting:** Modern SCADA platforms (e.g., Ignition) embed Python (specifically Jython) as the primary language for all internal customization, logic, and event handling.

**Data Liberation (Legacy SCADA):** Python is the external tool used to pull data from "brownfield" systems (e.g., Siemens, AVEVA) via open protocols like OPC UA, MQTT, and SQL.

**AI, ML & Edge:** This is the biggest driver. Python is the de facto language for AI. It is used to analyze SCADA data for predictive maintenance. This logic is now being deployed on industrial "Edge AI" devices (e.g., Revolution Pi) right on the factory floor.

# The Next Leap: Large Language Models

*Beyond executing code, LLMs understand context. How does this apply to engineering?*

# LLMs: A Quick Primer (Nov 2025)

### Contextual Reasoning

Not just text predictors. They are "reasoning engines" that understand context, logic, and technical nuance in documentation.

### Efficient & Domain-Specific

The trend has moved from giant models to efficient, Small Language Models (SLMs) trained specifically on engineering domains.

### On-Premise & Secure

Models are now small and fast enough to run "air-gapped" or on-premise, ensuring security for sensitive C-C data.

### Natively Multimodal

Modern models can interpret charts, schematics, code, and raw sensor log data simultaneously.

# Dual Impact: Accelerating Dev & Ops

### For Developers (Offline)

- **Full Repo Context:** LLMs understand the entire codebase structure via Git integration.
- **Rapid Code Gen:** Generate boilerplate, write unit tests, and refactor legacy modules instantly.
- **Intelligent Debugging:** Analyze stack traces and propose fixes via Pull Requests.

### For Operators (In-System)

- **System Analysis:** Fuse disparate data (logs, tickets, PDFs) to explain complex anomalies.
- **Root Cause Analysis:** Move beyond simple fault codes to deep, contextual explanations.
- **Co-Pilot Interface:** A natural language assistant for ad-hoc troubleshooting queries.

# The "ControlAgent" Paradigm

This paradigm is ideal for **ad-hoc tasks or queries where a dedicated feature hasn't been implemented.** It provides ultimate flexibility by separating the "brain" from the "hands" to get the best of both worlds.

- **LLM Agents (The Brain):** Act as the coordinator. They interpret high-level, natural language goals from the user.
- **Python Computation Agent (The Hands):** A deterministic Python script that performs the *actual* task and returns raw data.

```
# User: "Avg pressure on Valve 3 over the last 2h ?"
[LLM] -> Calls: get_avg_pressure('V3', 2)
[Python] -> data = scada.history('V3', '2h')
[Python] -> avg = calculate_mean(data)
[LLM] -> "The average pressure was 102.1 kPa."
```

# Critical Challenges & Hurdles

### Hallucinations

Critical systems cannot tolerate probabilistic answers. The LLM **cannot** guess.

### New Failure Modes

LLM-generated code can introduce subtle, plausible errors. Code must be human-verified.

### Security & Data

Sensitive SCADA logs cannot go to public APIs. Requires on-premise models.

### Real-Time Performance

LLMs are slow. This limits them to an **advisory** (offline) role, not real-time control loops.

# A Path Forward: Proposed Next Steps

### Form a Working Group

Establish an inter-lab committee to define common standards for simulation, testing, and data logging.

### Identify a Pilot Project

Select one common problem (e.g., a unified diagnostic logger) to co-develop as a shared Python-based tool.

### Start a Shared Repository

Create a central, internal repository for common Python utilities, control scripts, and best practices.

### Explore Secure LLMs

Begin R&D on a secure, on-premise SLM/RAG system for code documentation and diagnostics.

# Questions?

Thank you for your attention.

Leveraging Python & LLMs for System Maintainability