Python LLM DSLs

Simplifying LLM Integration for SLAC Scientific Computing

Your Name

March 16, 2025

The Problem: LLM Integration is Needlessly Complex

Standard Approach (LangChain):

```
from langchain_anthropic import ChatAnthropic
from langchain.prompts import PromptTemplate
from langchain.chains import LLMChain

llm = ChatAnthropic(anthropic_api_key=api_key,
                    model="claude-3-opus-20240229")
template = """Answer with JSON {'answer': true/false
    }:
              Is this EPICS PV in a valid state?"""
prompt = PromptTemplate(input_variables=[], template=
    template)
chain = LLMChain(llm=llm, prompt=prompt)
```

The Problem: LLM Integration is Needlessly Complex (cont.)

```python
result = chain.run({})  # Parse JSON manually
if json.loads(result)["answer"]:
    print("EPICS PV state is valid")
```

Why So Verbose?

- **Wrong abstractions:** Chain-based design doesn't fit LCLS real-time needs
- **Mixed concerns:** Prompt engineering, API calls, and parsing all exposed
- **Mental overhead:** Too cumbersome for beamline scientists to use on the fly

One Solution: Context-Sensitive Semantic DSL

```python
# Our DSL approach for LCLS configuration checking
from lcls_llm import check_configuration

if check_configuration("Is PV:UNDULATOR:GAP in valid
    range?"):
     proceed_with_experiment()
else:
     alert_operator("Configuration issue detected")
```

Key Innovation: Semantic extension based on context

- Same function behaves differently in different contexts
- LCLS-specific implementation details hidden from users
- Code reads naturally for scientists and operators

How It Works: Context-Sensitive Dispatch

```
# In a conditional context (fault detection):
if check_configuration("Is timing system properly
    configured?"):
     # Uses a JSON-based boolean template
     # Returns True/False for decision making
     start_beam_delivery()
```

```
# In a variable assignment (knowledge assistant):
explanation = check_configuration("Explain timing
    system status")
# Uses a text-based template for detailed information
# Returns full explanation for operators
log_and_display(explanation)
```

The DSL detects the syntactic context and adapts behavior transparently.

Vision: SLAC Facility-Specific LLM Primitives

Implementing SLAC's roadmap with purpose-built DSLs:

```
from slac_llm import check_epics, predict_anomaly,
    assist_timing

# LCLS configuration expert system
if check_epics("PV:UNDULATOR:CURRENT", target_range
    =(100, 120)):
     confirm_experiment_ready()

# S3DF anomaly detection
anomalies = predict_anomaly(beam_data, sensitivity
    =0.85)
if anomalies:
     recommend_actions(anomalies)

# Timing system helper
new_config = assist_timing.create_readout_group(
     existing_groups=[group1, group2],
```

Alignment with SLAC LLM Roadmap

Addresses Key Roadmap Categories:

- **Knowledge Assistants:** DSLs for RAG-based information retrieval from eLog, Slack
- **LCLS-specific Use Cases:** EPICS interpretation, timing system, configuration
- **S3DF Use Cases:** Anomaly detection, fault prediction, troubleshooting
- **Coding Assistants:** Analysis script generation for experimental data

Implementation Benefits:

- **Reduced learning curve:** Scientists use domain terminology, not LLM jargon
- **Integration with SLAC systems:** Works with existing EPICS, data acquisition
- **ROI enhancement:** Lower barriers to adoption accelerates value realization

Implementation Considerations for SLAC

1. **Resource Requirements**
   - DSLs need domain-specific training/fine-tuning
   - Integration with secure SLAC systems requires careful planning
2. **Development Priorities**
   - Which roadmap items would benefit most from DSL approach?
   - Begin with high-impact use cases (configuration, anomaly detection)
3. **The Path Forward**
   - Create working group with LCLS, S3DF representation
   - Develop prototype for one key use case (e.g., EPICS assistant)
   - Expand based on user feedback and roadmap priorities

Live Demo

Let's see how this works in practice...

[JUPYTER NOTEBOOK DEMO]