March 17, 2025

Talk Overview: Two Perspectives on LLM Integration

Part 1: Conceptual DSL Design

- LangChain et al. are clunky
- Context-aware procedures can help (toy example: a DSL embedded in Python)
- Mapping onto scientific use cases

Part 2: Case Study

- NumPy $\rightarrow$ PyTorch physics model port
- Current Claude + Aider workflow
- Friction points and opportunities

Connecting thread: How better LLM abstractions could help scientific uses of LLMs

The Problem: LLM Libraries Have Excessive Boilerplate

Example: Using LangChain for a simple boolean check:

```
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}: Do the
    BPMs indicate beam mis-steering?"""
prompt = PromptTemplate(input_variables=[], template=template)
chain = LLMChain(llm=llm, prompt=prompt)
result = chain.run({})  # Parse JSON manually
if json.loads(result)["answer"]:
    print("Beam position monitor diagnostics are valid")
```

Common Library Issues:

- **Excessive boilerplate:** Library interfaces force verbose patterns for simple operations. It's difficult to factor out API-level overabstractions
- **Manual parsing:** Output processing burden falls on the user

DSL Solution: Context-Sensitive LLM Functions

Our approach:

```
# As a boolean:
if llm("Do the BPMs indicate beam mis-steering?"):
    print("Need to correct steering")

# As text:
response = llm("Do the BPMs indicate beam mis-steering?")
print(response)  # "I reviewed the logged values of BPMS:
    LI21:233:X and ..."
```

Idea: The DSL detects how you're using the result and automatically:

- Uses the appropriate prompt template
- Handles the parsing for you
- Returns the right data type based on context

[JUPYTER NOTEBOOK DEMO]

Example: LLMs for beam steering

```
from slac_llm import llm, recommend_actions

# Hypothetical example of beam drift correction
if llm("Do the BPMs indicate beam mis-steering?"):
    recommend_actions("Review BPM diagnostic logs and
    advise how to recover")
else:
    print(llm("Summarize beam drift metrics from the last 5
     minutes"))
```

Why do this at SLAC?

- **Integration:** Freely mix LLM capability with existing Python constructs
- **Knowledge capture:** Encode expert knowledge in domain-specific primitives

Context Management: Why Current Approaches Fall Short

Another challenge: How do we handle context for LLM operations?

RAG Limitations in Current Frameworks:

- Vector embeddings lose semantics; RAG has lower recall than in-context retrieval

Better Approach:

- Task-specific full-text contexts
- Near-perfect recall in SOTA models

Case Study: NumPy to PyTorch Scientific Model Port

Project: Convert diffuse scattering simulator from NumPy to PyTorch

- Physics-based model for crystallographic diffuse scattering
- Tightly-coupled numerical components
- Specialized math operations requiring gradient flow

Traditional Approach: 6+ weeks of manual conversion

Result with LLMs: 1.5 weeks (75

Two-Tool LLM Workflow

The Process:

```
# Planning in Claude Web UI
- Map component interactions, save as architecture.md
- Draft phased implementation plan (iterate many times)
- For each step of the plan, ask LLM to draft a spec prompt

# Implementation in Aider:
- Select context, paste in spec prompt
- Aider auto-runs new unit tests and reads the output
- If needed: iterate in Aider until tests pass
```

LLM Workflow Friction Points

Key Friction Points:

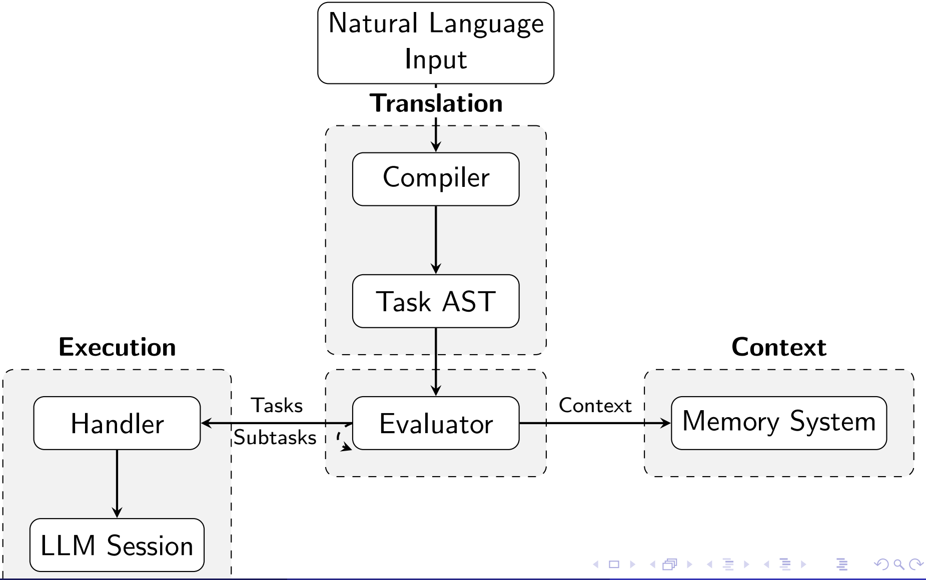
- Manual context management
- Copy/paste overhead
- **Tool-imposed limitations:**
  - Claude: Good for planning but lacks context managment and file system / shell integration
  - Aider: Good for code editing but its system prompt is counterproductive for general reasoning and planning tasks

```
# Claude session for high-level planning
claude> Follow bootstrap.xml. Then write
     the spec for phase 4.
[2000 token response with new spec]

# Copy-paste to Aider for implementation
aider> /add torch_utils.py, etc.
aider> Implement this spec: ...
[Aider usually succeeds but sometimes
     shoots you in the foot]

# Back to Claude for debugging
claude> Here's failing test output ...
[More context switching and copy-pasting]
```

'Agentic' frameworks solve some of these issues but don't give the necessary level of control, especially around context management

Future Direction: LLM Operating System

Long-term approach: A general task coordination framework



Potential Next Steps

1. **Start small:** Prototype one-off tools for specific SLAC use cases
2. **Explore context management:** Data handling choices – such as RAG vs. direct context management – become important at large scales
3. **Consider domain-specific libraries:** Would building a library of SLAC-specific LLM primitives be valuable?