ClinOrchestra: A Task-Agnostic Neuro-Symbolic AI Orchestration Platform

Project Name: ClinOrchestra

Lead/Mentor: Frederick Gyasi (gyasi@musc.edu) **Institution:** Medical University of South Carolina, Biomedical Informatics Center **Version:** 1.0.0

Brief Summary

ClinOrchestra is a **Task-Agnostic Neuro-Symbolic AI Orchestration Platform** that combines the reasoning capabilities of large language models (LLMs) with deterministic symbolic computation and knowledge retrieval systems. The platform implements a novel **Neuro-Symbolic Feedback Loop** architecture that can support **any task** as long as the output schema is properly defined.

Key Design Principle: ClinOrchestra is **task-agnostic and prompt-agnostic**. While initially developed for clinical NLP use cases, the architecture makes no assumptions about the domain, task type, or prompt structure. Users define their output schema, register relevant functions, configure domain-specific RAG sources, and add task-appropriate extras—the platform orchestrates the rest.

The core innovation lies in augmenting LLM-based processing with three complementary knowledge sources: 1. **Custom Functions** - Deterministic computations (any callable Python function) 2. **RAG (Retrieval Augmented Generation)** - Domain-specific document retrieval 3. **Extras** - Task-specific hints and domain knowledge patterns

---

Important Research Questions

1. **Can neuro-symbolic integration improve task accuracy** compared to pure LLM approaches by grounding neural reasoning in symbolic domain knowledge?

2. **How does the Neuro-Symbolic Feedback Loop** (Functions + RAG + Extras) enhance LLM performance on complex tasks requiring domain expertise?

3. **What is the optimal balance** between neural (LLM reasoning) and symbolic (deterministic computation) components for different task types?

Core Architecture: The Neuro-Symbolic Feedback Loop

Architectural Philosophy

ClinOrchestra implements a **hybrid neuro-symbolic architecture** where: - **Neural Component**: LLMs (Claude, GPT-4, Llama) handle natural language understanding, context interpretation, and synthesis - **Symbolic Component**: Functions, RAG, and Extras provide deterministic computation, factual grounding, and domain expertise

The feedback loop ensures that LLM outputs are informed by and validated against symbolic knowledge sources. **Critically, this architecture is domain-agnostic**—it can be applied to any task where structured output is desired, from clinical NLP to legal document analysis, financial data extraction, scientific literature processing, or any custom domain.

Two Execution Modes

Mode	Implementation	Characteristics	Best Use Case
STRUCTURED	`core/agent_system.py`	4-stage deterministic pipeline, predictable, reproducible	Production workloads
ADAPTIVE	`core/agentic_agent.py`	ReAct-style iterative agent, self-directed tool selection	Exploratory tasks

---

STRUCTURED Mode: 4-Stage Pipeline

The STRUCTURED mode (`core/agent_system.py:61-1200`) implements a deterministic 4-stage extraction pipeline:

Stage 1: Task Understanding (Lines 373-513)

Purpose: LLM analyzes input text + output schema to plan tool usage

```
Input Text + Schema → LLM → Task Understanding Plan
                                ├── functions_needed[]
                                ├── rag_queries[]
                                └── extras_keywords[]
```

Key Implementation: - `_execute_stage1_understanding()` (line 373) - `_convert_task_understanding_to_tool_requests()` (line 514) - LLM determines which functions to call, what RAG queries to execute, and which extras keywords to match

Stage 2: Tool Execution (Lines 699-820)

Purpose: Execute all planned tool requests with parallel async processing

```
Tool Requests → Parallel Execution → Tool Results
                ├── Functions (deterministic)
                ├── RAG queries (vector search)
                └── Extras matching (keyword-based)
```

Key Features: - **Parallel Async Execution** (`_execute_stage2_tools_async()`, line 784): 60-75% performance improvement - **Tool Deduplication** (`_deduplicate_tool_requests()`, line 550): Prevents repeated identical calls - **Parameter Validation** (`_validate_tool_parameters()`, line 597): Universal validation for any function - **Function Chaining** (`_detect_tool_dependencies()`, line 821): Supports `$call_X` syntax for referencing outputs - **Topological Sort** (`_topological_sort_tools()`, line 876): Correct execution order for dependencies

Stage 3: Synthesis (Lines 1050-1150)

Purpose: LLM combines tool results with domain understanding to produce structured output

```
Tool Results + Original Text + Schema → LLM → Structured JSON Output
```

Key Implementation: - `_execute_stage3_synthesis()` - Constructs enhanced prompt with all tool results - LLM synthesizes final extraction following schema

Stage 4: Validation (Lines 1150-1250)

Purpose: Validate output against schema with optional self-critique

```
JSON Output → Schema Validation → Optional Self-Critique → Final Output
```

Key Features: - Pydantic-based schema validation - Optional iterative self-critique for quality improvement - Error recovery and re-extraction attempts

---

The Three Knowledge Sources (Symbolic Components)

1. Custom Functions (`core/function_registry.py`)

Purpose: Register any deterministic Python functions for domain-specific computations

Architecture (Lines 1-350):

```
class FunctionRegistry:
    def register_function(name, callable, parameters, description)
    def execute_function(name, parameters) -> result
    def get_function_info(name) -> function_definition
```

Key Features: - **Parameter Validation** (line 180-260): Type checking, required/optional parameters - **Parameter Transformation** (line 89-150): Maps schema fields to function parameters - **Flexible Registration**: Any Python callable can be registered

Example Function Registration:

```
registry.register_function(
    name="creatinine_clearance",
    callable=calculate_creatinine_clearance,
    parameters={
        "age": {"type": "number", "required": True},
        "weight": {"type": "number", "required": True},
        "creatinine": {"type": "number", "required": True},
        "sex": {"type": "string", "required": True}
    },
    description="Calculate creatinine clearance using Cockcroft-Gault"
)
```

Example Use Cases (Domain-Specific): - **Clinical:** Scoring systems (SOFA, APACHE), pharmacokinetic calculations, risk calculators - **Financial:** Interest calculations, risk scoring, currency conversions - **Legal:** Statute lookup, deadline calculations, jurisdiction determination - **Scientific:** Unit conversions, statistical computations, formula evaluations - **General:** Any deterministic computation that should not be left to LLM approximation

2. RAG Engine (`core/rag_engine.py`)

Purpose: Retrieval Augmented Generation for domain-specific knowledge retrieval

Architecture (Lines 1-926):

```
Documents → Chunking → Embedding → FAISS Index → Vector Search → Relevant
```

Core Components:

Component	Class	Purpose
Document Loading	`DocumentLoader` (line 50-230)	Load from URLs, PDFs, HTML, TXT files
Chunking	`DocumentChunker` (line 322-358)	Sliding window chunking (default 512 chars, 50 overlap)
Embedding	`EmbeddingGenerator` (line 232-320)	SentenceTransformer embeddings with caching
Vector Store	`VectorStore` (line 360-646)	FAISS with cosine similarity, GPU acceleration

Component	Class	Purpose
RAG Engine	`RAGEngine` (line 647-926)	Orchestrates retrieval pipeline

Key Features:

1. **Batch Embedding Generation** (line 256-316): 25-40% faster with configurable batch size
2. **Query with Variations** (line 841-901): Improved recall through term expansion

   ```
   results = engine.query_with_variations(
       primary_query="diagnostic criteria",
       variations=["diagnosis", "clinical criteria", "assessment"]
   )
   ```

3. **GPU Acceleration** (line 375-462): FAISS GPU mode for 10-90x faster searches
4. **Persistent Caching** (line 473-590): SQLite-based embedding and document cache
5. **Query Result Caching** (line 804-837): Avoid redundant vector searches

Supported Document Formats: - PDF (with encryption handling) - HTML (with script/style stripping) - Plain text and Markdown - Remote URLs with automatic caching

3. Extras Manager (`core/extras_manager.py`)

Purpose: Task-specific supplementary hints to help LLM understand domain patterns

Architecture (Lines 1-643):

```
class ExtrasManager:
    def add_extra(type, content, metadata, name)
    def match_extras_by_keywords(keywords) -> matched_extras
    def match_extras_with_variations(core_keywords, variations_map)
```

Critical Design Principle (Lines 20-33):

> Extras are NOT for matching against input text - they are supplementary knowledge that helps the LLM better understand how to approach the extraction task.

Matching Algorithm (Lines 105-170):

```
Schema Fields → Keywords → Keyword Matching → Relevance Scoring → Top Extr
```

Relevance Scoring (Lines 301-353): - Exact keyword match in content: +1.0 - Partial keyword match: +0.5 - Keyword in type field: +0.3 - Keyword in metadata: +0.2 - Multi-keyword bonus: ×1.2

Key Features:

1. **Keyword-Based Matching** (line 105): Matches against schema fields, NOT input text
2. **Variations Support** (line 172-220): Term expansion for better recall
3. **Enable/Disable** (line 463-483): Toggle individual extras
4. **Import/Export** (line 543-593): YAML/JSON support for extras management
5. **Result Caching** (line 127-138): Avoid redundant matching computations

Example Extras:

```
- type: "diagnostic_criteria"
  content: "Assessment requires presence of at least 3 of 5 criteria..."

- type: "assessment_methodology"
  content: "When evaluating severity, consider temporal progression..."

- type: "domain_pattern"
  content: "In clinical notes, abbreviations commonly used include..."
```

ADAPTIVE Mode: ReAct-Style Agent (`core/agentic_agent.py`)

Purpose: Self-directed iterative agent for exploratory or complex tasks

Architecture (Lines 1-800):

```
Observe → Think → Act → Observe → Think → Act → ... → Final Output
```

Key Features: - **ReAct Pattern**: Iterative reasoning with tool selection - **Self-Directed**: Agent decides which tools to call - **Multiple Iterations**: Continues until extraction complete or max iterations - **Dynamic Tool Selection**: Chooses between Functions, RAG, Extras based on need

Configuration:

```
AgenticExtractionAgent(
    max_iterations=10,
    enable_self_critique=True,
    think_aloud=True
)
```

Use Cases: - Complex extraction requiring exploration - Tasks with unclear requirements - Research and development

---

Application State Management (`core/app_state.py`)

Purpose: Centralized state management for the orchestration session

Key Components (Lines 1-200): - `AppState`: Holds all session configuration - `ExtractionContext`: Per-extraction state (tool requests, results) - `OptimizationConfig`: GPU settings, caching, performance tuning

---

Prompt Engineering (`core/prompt_templates.py`)

Purpose: Structured prompts for each pipeline stage

Templates: - `STAGE1_TASK_UNDERSTANDING_PROMPT`: Plan tool usage - `STAGE2_TOOL_EXECUTION_PROMPT`: Execute tools (internal) - `STAGE3_SYNTHESIS_PROMPT`: Combine results into output - `STAGE4_VALIDATION_PROMPT`: Self-critique and validation

---

Web UI: Configuration via YAML/JSON

ClinOrchestra provides a **Gradio-based web interface** where all components can be configured through structured file uploads (YAML or JSON) or interactive forms. This enables **no-code task configuration**—users define their task entirely through the UI without writing Python code.

Prompt & Schema Configuration (`ui/prompt_tab.py`)

The Prompt tab allows users to: - Define **main prompts** (primary task instructions) - Create **minimal prompts** (fallback for context window issues) - Configure **RAG refinement prompts** (evidence-based output refinement) - Define **JSON output schema** via interactive field builder or file upload

Schema File Upload (YAML or JSON):

```
# schema.yaml - Simplified format
diagnosis:
  type: string
  description: "Primary diagnosis from the note"
  required: true
severity:
  type: string
  description: "Severity classification"
  required: true
confidence:
  type: number
```

```
  description: "Confidence score 0-1"
  required: false
```

```
// schema.json - JSON Schema format also supported
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "type": "object",
  "properties": {
    "diagnosis": {"type": "string", "description": "Primary diagnosis"},
    "severity": {"type": "string"}
  },
  "required": ["diagnosis", "severity"]
}
```

Functions Configuration (`ui/functions_tab.py`)

Register custom Python functions via the UI:

Functions File Schema (YAML):

```
functions:
  - name: calculate_bmi
    description: "Calculate Body Mass Index"
    code: |
      def calculate_bmi(weight_kg, height_m):
          if height_m <= 0:
              return None
          return round(weight_kg / (height_m ** 2), 2)
    parameters:
      weight_kg:
        type: number
        description: "Weight in kilograms"
      height_m:
        type: number
        description: "Height in meters"
    returns: "BMI value (number)"
```

Functions File Schema (JSON):

```
{
  "functions": [
    {
      "name": "calculate_bmi",
      "description": "Calculate Body Mass Index",
      "code": "def calculate_bmi(weight_kg, height_m):\n    if height_m <=
      "parameters": {
        "weight_kg": {"type": "number", "description": "Weight in kilogram
        "height_m": {"type": "number", "description": "Height in meters"}
      },
      "returns": "BMI value (number)"
    }
```

```
  ]
}
```

UI Features: - Interactive code editor with Python syntax highlighting - Parameter builder with type selection (string, number, boolean) - Test function execution with JSON arguments - Enable/disable individual functions - Import/export all functions

Extras Configuration (`ui/extras_tab.py`)

Add task-specific hints and domain knowledge:

Extras File Schema (YAML):

```
extras:
  - name: "WHO Malnutrition Criteria"
    type: definition
    content: "BMI categories - Underweight: <18.5, Normal: 18.5-24.9, Over
    metadata:
      category: clinical
      priority: high

  - name: "Lab Value Pattern"
    type: pattern
    content: "Lab values format - [test name]: [value] [unit]. Example: Gl
    metadata:
      category: medical
```

Extras File Schema (JSON):

```
{
  "extras": [
    {
      "name": "WHO Malnutrition Criteria",
      "type": "definition",
      "content": "BMI categories - Underweight: <18.5, Normal: 18.5-24.9",
      "metadata": {"category": "clinical", "priority": "high"}
    }
  ]
}
```

Extra Types: `pattern`, `definition`, `guideline`, `example`, `reference`, `criteria`, `tip`

UI Features: - Add/edit extras with name, type, content, and metadata - Enable/disable individual extras via checkboxes - Cache management (bypass or clear) - Import/export functionality

RAG Configuration (`ui/rag_tab.py`)

Configure document sources and retrieval settings:

Document Sources (via UI): - **URLs tab**: Paste document URLs (PDFs, HTML, TXT) one per line - **File Paths tab**: Enter local file paths one per line - **Upload tab**: Drag-and-drop document upload (.pdf, .txt, .md, .html)

Embedding & Retrieval Settings: | Setting | Default | Description | |---------|---------|-------------| | Embedding Model | `all-mpnet-base-v2` | SentenceTransformer model for embeddings | | Chunk Size | 512 | Characters per text chunk | | Chunk Overlap | 50 | Overlap between chunks | | K Value | 5 | Number of chunks retrieved per query |

Cache Controls: - Clear embedding cache (when configuration changes) - Bypass query cache (force fresh retrieval) - Clear query cache

Complete Task Configuration Workflow

1. **Prompt Tab**: Define task prompt + JSON output schema (upload YAML/JSON or use builder)
2. **Functions Tab**: Upload functions YAML/JSON or define interactively
3. **Extras Tab**: Upload extras YAML/JSON or add manually
4. **RAG Tab**: Add document sources + configure retrieval settings + initialize
5. **Processing Tab**: Upload data, run extraction, view results

---

Example Usage Patterns

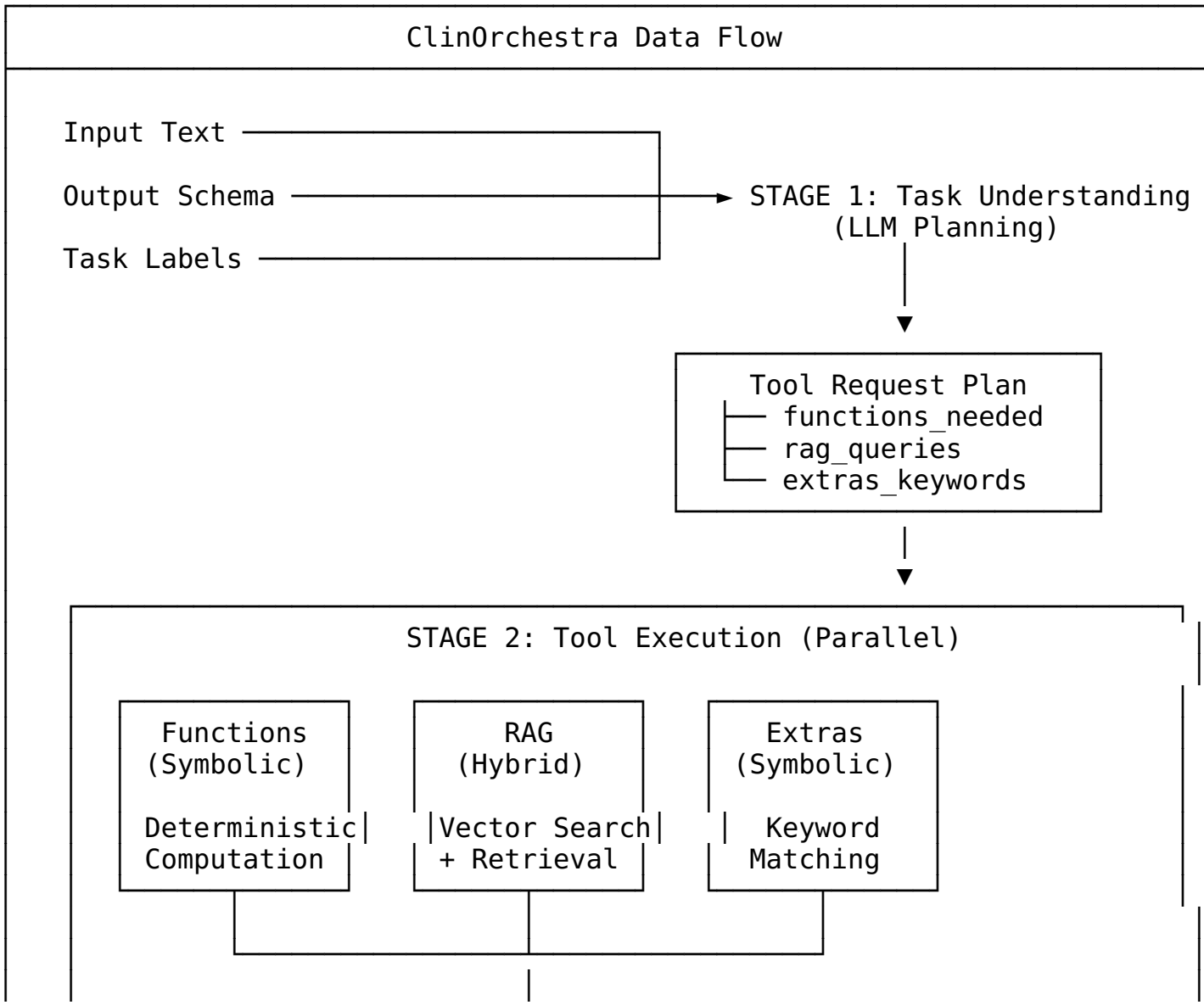
SDK Integration (from `examples/`)

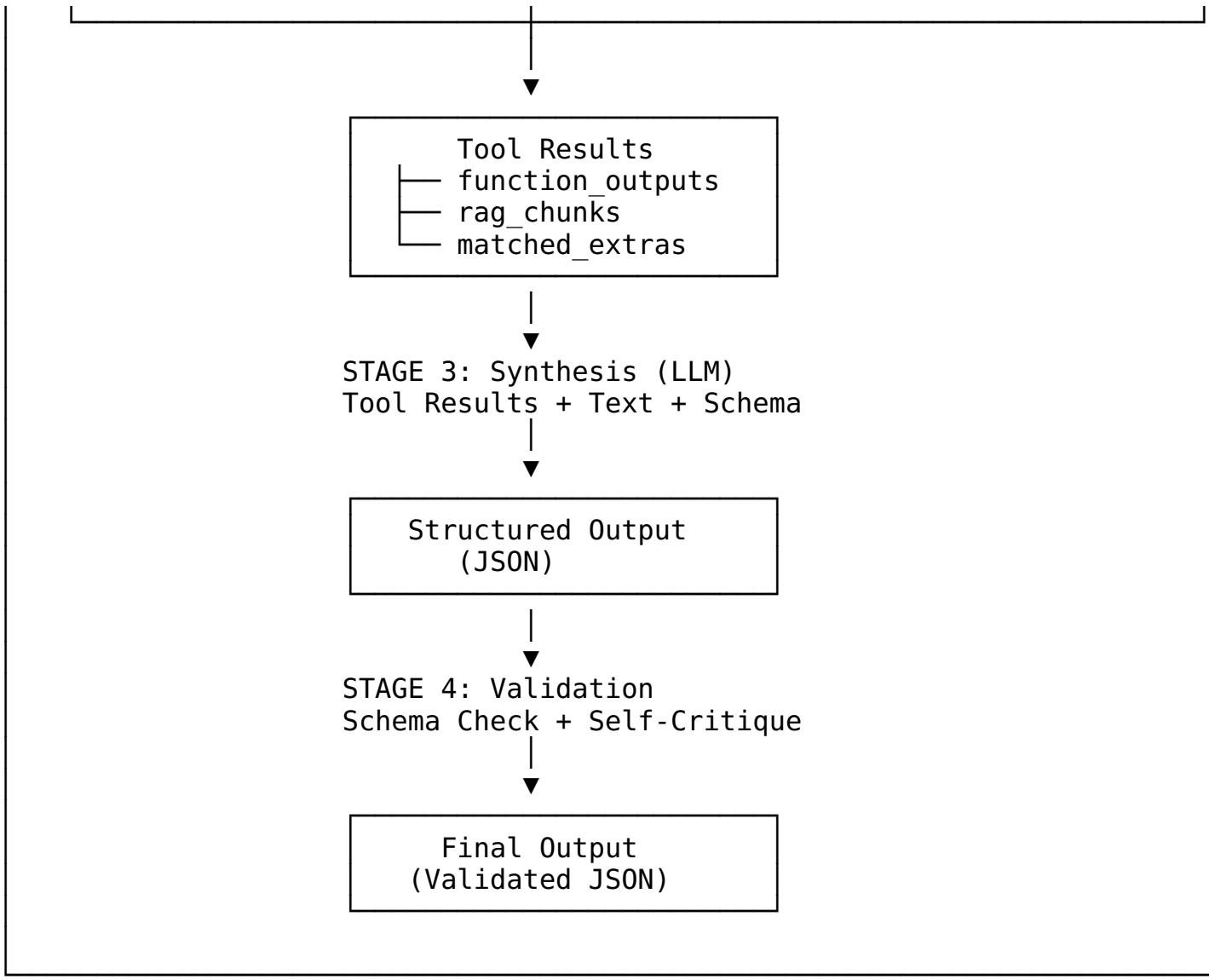
Basic Configuration:

```
from core.agent_system import StructuredExtractionAgent
from core.function_registry import FunctionRegistry
from core.rag_engine import RAGEngine
from core.extras_manager import ExtrasManager

# 1. Define output schema
schema = {
    "type": "object",
    "properties": {
        "diagnosis": {"type": "string"},
        "severity": {"type": "string", "enum": ["mild", "moderate", "sever
        "confidence": {"type": "number"}
    }
}

# 2. Register custom functions
registry = FunctionRegistry()
registry.register_function("calculate_score", score_function, params, desc

# 3. Initialize RAG with domain documents
rag_config = {"sources": ["guidelines.pdf", "criteria.html"]}
```

```
rag_engine = RAGEngine(rag_config)

# 4. Load task-specific extras
extras_manager = ExtrasManager("./extras")
extras_manager.add_extra("hint", "Look for specific criteria...", {})

# 5. Create agent and extract
agent = StructuredExtractionAgent(
    function_registry=registry,
    rag_engine=rag_engine,
    extras_manager=extras_manager,
    schema=schema
)

result = agent.extract(clinical_text)
```

Data Flow Diagram

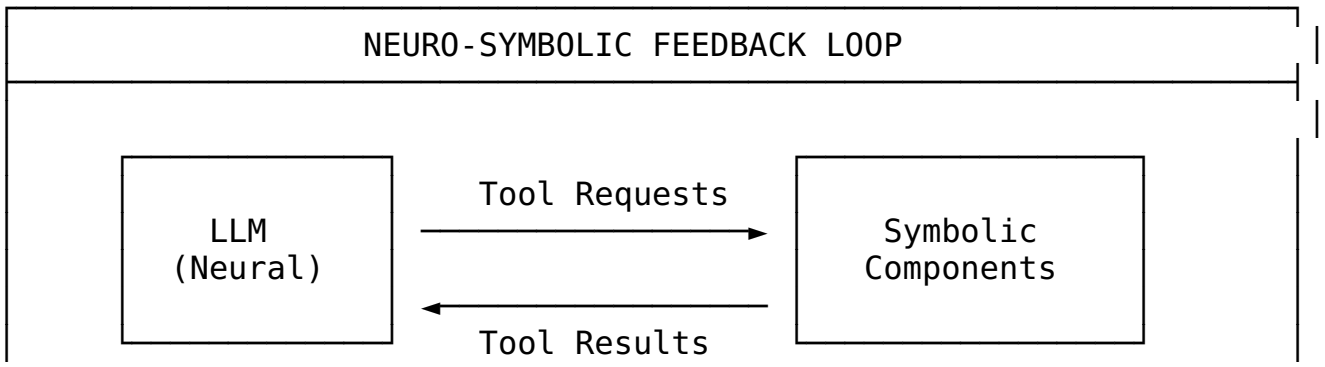


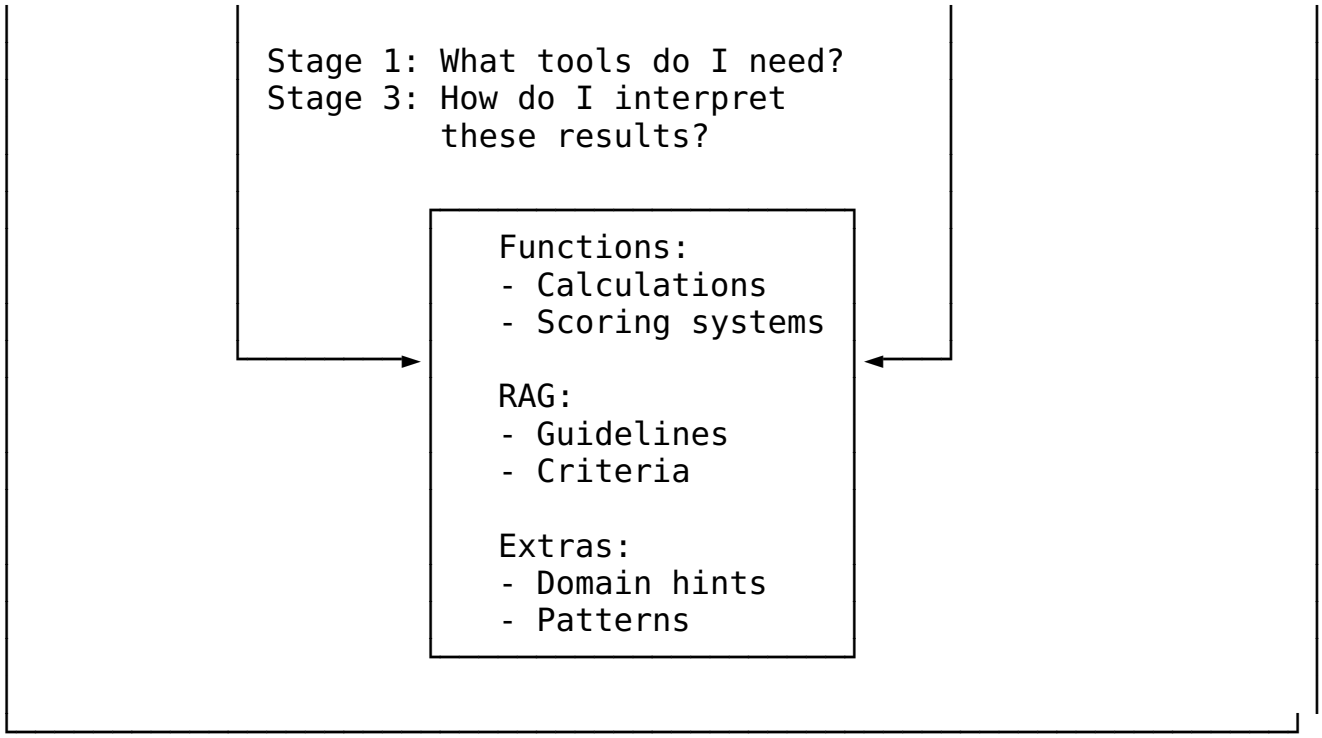


Neuro-Symbolic Feedback Loop: Technical Details

How the Loop Works

The neuro-symbolic feedback loop operates through structured interaction between neural (LLM) and symbolic (Functions/RAG/Extras) components:





Benefits of the Approach

1. **Grounded Reasoning**: LLM outputs are informed by factual, retrievable knowledge
2. **Deterministic Computation**: Clinical calculations are exact, not approximated
3. **Domain Expertise**: Task-specific hints guide LLM interpretation
4. **Transparency**: Tool calls and results are logged and traceable
5. **Reproducibility**: Same inputs produce same outputs (STRUCTURED mode)

Performance Optimizations

Implemented Optimizations (v1.0.0)

Optimization	Location	Improvement
Parallel Tool Execution	`agent_system.py:784`	60-75% faster
Batch Embedding	`rag_engine.py:256`	25-40% faster
Query Result Caching	`rag_engine.py:804`	Avoid redundant searches
Embedding Caching	`rag_engine.py:277`	In-memory cache
Document Caching	`rag_engine.py:59`	SQLite persistent cache
Extras Result Caching	`extras_manager.py:127`	Avoid redundant matching
GPU FAISS Acceleration	`rag_engine.py:375`	10-90x faster vector search
Tool Deduplication	`agent_system.py:550`	Prevent repeated calls

Enabling GPU Acceleration

```
# Environment variable
export CLINORCHESTRA_ENABLE_GPU=1

# Or in config
rag_config = {
    "use_gpu_faiss": True,
    "gpu_device": 0  # Which GPU to use
}
```

Resources

Code Repository: GitHub (private/institutional)

Key Dependencies: - `anthropic` / `openai` / `ollama` - LLM backends - `sentence-transformers` - Embedding generation - `faiss-cpu` / `faiss-gpu` - Vector similarity search - `gradio` - Web UI framework - `pydantic` - Schema validation

---

Acknowledgements

Author: Frederick Gyasi (gyasi@musc.edu) **Institution:** Medical University of South Carolina, Biomedical Informatics Center

---

Summary

ClinOrchestra is a **task-agnostic, prompt-agnostic orchestration platform** that implements a **Neuro-Symbolic Feedback Loop** combining:

1. **Neural reasoning** (LLMs) for natural language understanding and synthesis
2. **Symbolic computation** (Functions) for deterministic domain-specific calculations
3. **Knowledge retrieval** (RAG) for document grounding
4. **Task guidance** (Extras) for domain expertise hints

The platform can support any task as long as the output schema is properly defined. Whether the domain is clinical NLP, legal document processing, financial analysis, scientific literature extraction, or any custom application—ClinOrchestra orchestrates the neuro-symbolic workflow.

This hybrid architecture addresses key limitations of pure LLM approaches: - **Hallucination mitigation** through factual grounding - **Computational accuracy** through deterministic functions - **Domain expertise** through retrievable knowledge - **Reproducibility** through structured pipeline execution

The platform supports both **STRUCTURED** (production) and **ADAPTIVE** (exploratory) modes, making it suitable for research and deployment across any domain.