Recursive Task Processing System Design Document

Status: Draft
Authors: [Authors]
Reviewers: [Reviewers]
Last Updated: 2024-11-13

Overview
This document proposes a system for breaking down complex AI tasks into manageable subtasks, processing them recursively, and composing the results into coherent outputs. The system addresses the fundamental challenge of managing large language model (LLM) token limitations while maintaining context and coherence across complex tasks.

Context
Current approaches to handling complex AI tasks often rely on increasingly sophisticated models or complex orchestration frameworks. These solutions typically trade external complexity for internal complexity, making systems harder to understand, maintain, and debug. This design takes the opposite approach: maintaining external simplicity through clean abstractions while allowing for sophisticated internal processing.

The system draws inspiration from classical software engineering principles, particularly:
- Recursive problem decomposition
- Clean separation of concerns
- Rational engineering (minimal necessary complexity)
- Composition of simple, well-defined interfaces

Goals
Primary Goals
- Create a system that can recursively break down complex tasks into token-limited subtasks
- Enable high-quality output composition through iterative refinement
- Maintain context and coherence across subtask boundaries
- Provide clear debugging and introspection capabilities
- Support multiple LLM backends through clean abstractions

Secondary Goals
- Enable easy testing through abstract interfaces
- Support different task decomposition strategies
- Allow for flexible refinement policies
- Facilitate research into AI task processing patterns

Non-Goals
- Creating a general-purpose workflow orchestration system
- Optimizing for maximum performance in initial implementation
- Supporting real-time or streaming processing
- Handling multi-modal inputs/outputs in initial version
- Managing distributed computation
- Providing built-in security features beyond basic validation
- Supporting concurrent task processing

Success Metrics
Functional
- Successfully processes complex tasks that exceed single-prompt token limits
- Maintains coherence and consistency across subtask boundaries
- Produces outputs that match or exceed quality of single-prompt attempts

Technical
- All components have clear, minimal interfaces
- Full system testable without LLM dependencies
- Debug output available at each processing stage
- Easy to extend with new decomposition/refinement strategies

Development
- Implementation complexity primarily in concrete classes, not interfaces
- Easy to understand and modify system behavior
- Clear separation between framework and implementation concerns

System Architecture

Core Components

Data Structures
```
TaskNode
 ├── prompt: str          # Task description/prompt
 ├── parent: TaskNode     # Parent task (null for root)
 ├── subtasks: [TaskNode]  # Child tasks from decomposition
 ├── result: str          # Execution/refinement result
 └── debug_str(): str     # Human-readable state representation
```

Primary System Interfaces

1. Understanding Phase
```
RequestTranslator (abstract)
 └── translate(request: str) → TaskNode
```
Converts human requests into system-processable task nodes. Acts as validation layer between external input and internal processing.

2. Planning Phase
```
TaskPlanner (abstract)
 ├── is_base_case(task: TaskNode) → bool
 └── decompose(task: TaskNode) → None

PlanningGraphTraversal (abstract)
 └── traverse(root: TaskNode) → Iterator[TaskNode]
```
Handles recursive task breakdown through clean separation of decomposition logic from traversal strategy.

3. Execution Phase
```
```

```
TaskExecutor (abstract)
└─── execute(task: TaskNode) → None

ExecutionGraphTraversal (abstract)
└─── traverse(root: TaskNode) → Iterator[TaskNode]
```
Manages initial execution of leaf tasks in planned sequence.

4. Refinement Phase
```
RefinementCheck (abstract)
└─── needs_refinement(task: TaskNode) → bool

ComposingRefinementStrategy (abstract)
└─── compose_and_refine(task: TaskNode) → None

RefinementCoordinator
└─── refine_tree(root: TaskNode) → None
```
Handles iterative improvement and composition of results.

System Orchestration
```
TaskOrchestrator
├─── translator: RequestTranslator
├─── planner: TaskPlanner
├─── planning_traversal: PlanningGraphTraversal
├─── executor: TaskExecutor
├─── execution_traversal: ExecutionGraphTraversal
├─── refinement_coordinator: RefinementCoordinator
└─── process(request: str) → str
```
Coordinates all phases while maintaining clean separation of concerns.

Design Considerations

Key Design Decisions

1. **Separation of Planning and Execution**
   - Decision: Complete task planning before any execution
   - Rationale: Enables global optimization of task breakdown
   - Alternative Considered: Interleaved planning/execution
   - Trade-off: Additional memory usage vs. potential early optimization

2. **Unified Composition and Refinement**
   - Decision: Combined ComposingRefinementStrategy interface
   - Rationale: Allows refinement decisions based on composition context
   - Alternative Considered: Separate composition and refinement phases

   - Trade-off: Simplified architecture vs. potentially more complex implementations

3. **Abstract Graph Traversal**
   - Decision: Separate traversal strategies for planning and execution
   - Rationale: Different optimal orders for different phases
   - Alternative Considered: Single traversal strategy
   - Trade-off: Interface complexity vs. flexibility

4. **In-Place Result Updates**
   - Decision: Modify TaskNode results directly
   - Rationale: Simplifies state management and debugging
   - Alternative Considered: Immutable nodes with result chains
   - Trade-off: Mutability vs. history tracking

Extensibility Points

1. **Task Planning**
   - Custom decomposition strategies
   - Different base case criteria
   - Domain-specific planning approaches

2. **Execution**
   - Multiple LLM backends
   - Different prompt strategies
   - Custom execution patterns

3. **Refinement**
   - Various quality metrics
   - Different composition strategies
   - Custom iteration policies

Design Principles Applied

1. **Rational Engineering**
   - Minimal necessary interfaces
   - Clear separation of concerns
   - Deferred complexity until needed
   - Built-in debugging capabilities

2. **Composability**
   - Abstract interfaces enable mix-and-match implementations
   - Clean boundaries between components
   - Consistent patterns across system

3. **Testability**
   - All key behaviors behind interfaces
   - No direct LLM dependencies in framework
   - Clear state inspection points

Open Questions

1. **State Management**
   - Should we track iteration history?
   - How to handle partial failures?

   - When to clear intermediate results?

2. **Context Propagation**
   - How to share context efficiently between phases?
   - What metadata should travel with tasks?
   - How to handle global constraints?

3. **Optimization Opportunities**
   - Where to introduce parallelism?
   - How to cache intermediate results?
   - When to prune unnecessary refinement?