ChatGPT

Symbolic Delimiters and Tokenization Strategies

Modern LLM prompting increasingly employs **symbolic delimiters** – e.g. special tokens, emojis, or rare Unicode characters – to structure instructions and signal boundaries between tasks or phases. For example, Luo *et al.* (2024) propose appending a sentinel token `<SR>` at the end of each text chunk, and modifying attention masks so the model "takes a deep breath" and **aggregates** chunked information via that token [1]. Similarly, frameworks like ARTIST use XML-like tags (`<think>…</think>`, `<tool_name>…</tool_name>`, `<output>…</output>`) to explicitly mark **reasoning**, **tool-call**, and **result** phases within a prompt [2]. These tokens remain stable under tokenization and serve as clear anchors for structured prompting (e.g. nesting tasks or demarcating input/output of tools).

- **Chunking Sentinels:** In the *"Taking a Deep Breath"* method, each segment of context is terminated by `<SR>`, which the model learns as a summary node. This ensures that entire spans compress into a single token-sensitive pointer [1].
- **Phase Markers:** In ARTIST's agentic reasoning, bespoke tags like `<think>` and `<output>` appear in the model's chain-of-thought, enforcing a "tool-augmented CoT" flow [2]. By tokenizing `<tool_name>` and `<output>` as literal tokens, the LLM reliably switches modes (from planning to execution) without confusion.
- **Unicode/Grapheme Delimiters:** Researchers have noted that even **emojis or rare glyphs** can radically alter tokenization. For instance, injecting emoji characters can fragment words into new sub-tokens and confuse downstream evaluators [3]. While this poses a security risk (e.g. in "Emoji Attack"), it also implies that carefully-chosen Unicode symbols can serve as *unique, non-overlapping markers* in prompts (so long as their tokenization effects are understood). In practice, one might use a highly-unlikely symbol (, , μ, etc.) as a **block delimiter** to guarantee a single token stable signal.

By choosing sentinel tokens outside normal vocabulary (like XML tags or rare symbols), prompt authors create **token-stable signals** that survive Byte-Pair Encoding intact and guide the LLM's processing. This enables reliable parsing of complex prompts: e.g. a trigger emoji can universally denote "memory recall" or a bracket symbol can denote "start subtask." These strategies are documented as empirically improving LLM behavior in tasks requiring long or nested context [1] [3].

Prompt Decomposition and Multi-Phase Tool Calling

Recent work reveals that **decomposing prompts into structured phases** greatly aids complex multi-step reasoning. Instead of a monolithic "chain-of-thought," next-gen methods break tasks into sub-instructions and interleave **tool calls** or subroutine executions. ARTIST (2025) provides a vivid example: the model's reasoning alternates between segments like

```
<think>…</think>  ⇒  <python>…</python>  ⇒  <output>…</output>
```

where `<python>…</python>` is treated as a tool invocation and `<output>` as its result [2]. Each tagged segment is a phase delimiter, ensuring that code generation and tool outputs stay clearly separated from

free-form reasoning. This *hybrid CoT* framework ("Reason-Thinking Tool-Use Reason-Output") enforces correctness at each step: the model must conclude a `<think>` block before calling a tool, and cannot proceed until an `<output>` is returned.

Similarly, **tool-augmented pipelines** often inject function-call tokens or DSL blocks. For example, in multi-step math solving one might prompt:

```
<solve> Calculate ∫ x dx </solve>
<python>import sympy as sp; print(sp.integrate(x, x))</python>
<result>1/2*x**2</result>
```

This structured flow guarantees that each subtask has clear boundaries. Ye *et al.* (2025) demonstrate a *Task Memory Engine* where a hierarchical Task Memory Tree (TMT) is constructed: each node represents a sub-step, and the prompt synthesizer uses the **active path** in the TMT to generate the next instruction [4] [5]. In effect, the agent creates prompts like `Perform [Subtask X] given [memory of previous steps]`. By treating each node as a symbolic frame (with input, output, status), the LLM gets **structured cues** for decomposition.

Hybrid numeric-symbolic calls also follow this pattern. In advanced math and planning tasks, LLMs are guided to produce code or formal language, then verify or refine: e.g. an LLM might output a PDDL plan which a symbolic solver checks and returns feedback [6] [7]. This creates a *feedback loop* (LLM → symbolic solver → LLM) that is effectively a multi-phase pipeline. The use of formal grammar (PDDL tokens) and solver responses adds structure and reliability that purely free-text CoT lacks [7] [6].

Agent-to-Agent Routing and Symbolic MoEs

LLM-based agents are often deployed in **multi-agent systems (MAS)**, where specialized models or "experts" collaborate on parts of a task. These systems rely on symbolic cues for routing and coordination. By definition, an LLM-agent has a prompt state and uses tools as actions [8], and an MAS is "a collection of agents designed to interact through orchestration," enabling task decomposition and parallelism [8]. In practice, frameworks like Symbolic Mixture-of-Experts (Symbolic-MoE, 2025) implement this by inferring symbolic **skills** or keywords for each query. The system matches these skills against model profiles and *textually recruits* the best experts. For instance, to solve a math problem about algebra and calculus, the system might annotate skills `{Algebra, Integration}` and route the query to models known to excel in those areas. Each expert then works independently, and an **aggregator agent** (selected by its symbolic ability) synthesizes their textual outputs into a final answer [9] [10].

- **Skill-Based Routing:** Symbolic-MoE first derives a small set of discrete skill tokens for the task, then uses these tokens to match and activate a sparse set of LLM experts [9]. The communication is entirely text-based, but the **symbolic labels** (e.g. "Probability", "Algebra") define the routing.
- **Modular Roles:** Multi-agent studies (e.g. *ChatDev*, *AutoGen*, etc.) show that explicitly specifying agent roles or using hierarchies improves success. Interventions like clear role prompts and coordination graphs have been shown to partially mitigate failures [11] [12]. For example, adding a `Supervisor` role or using a `CEO-Agent` and `Engineer-Agent` with distinct emojis/headers can help isolate subtasks. Though many MAS still fail due to misalignment, research emphasizes that giving each

agent a narrow symbolic specification (with sentinel flags and termination markers) is key to reliability [11] [8].

Overall, agent routing often embeds **symbolic instructions** in the prompts. An orchestrator might send "*WorkerAgent*: Analyze data; *RouterAgent*: Combine worker outputs" where emojis or unique labels (, `RouterAgent:`) ensure stable splitting. The agents in the MAS treat these tokens literally, so the communication channel remains noise-free and machine-oriented.

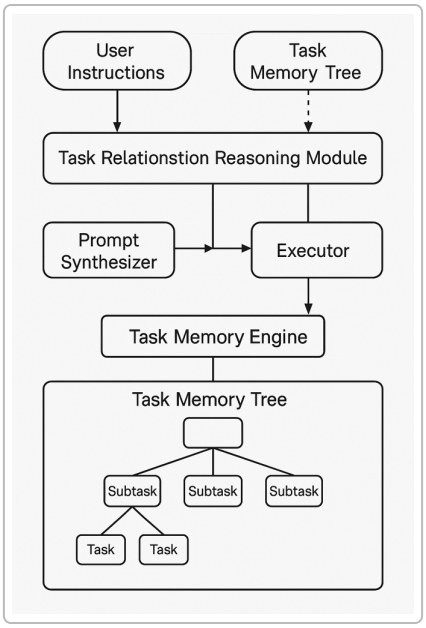
Hybrid Symbolic-Numeric Pipelines

Across domains, cutting-edge pipelines interleave neural generation with formal symbolic reasoning. In geometry proof generation, for instance, the workflow is: **(1) Retrieval** – rewrite the problem abstractly (replacing specific numeric values with placeholders) and fetch an analogous solved problem [13]; **(2) LLM Proofing** – generate a candidate proof in formal steps; **(3) Symbolic Verification** – use a theorem prover to check validity and give structured error feedback [13]. This neuro-symbolic loop (LLM ↔ solver) greatly boosts correctness: Sultan *et al.* report proof accuracy surging by ~58–70% when combining analogical prompts with a formal verifier [13]. The placeholders and retrieved proof act as *symbolic scaffolding*, while the LLM fills in the reasoning.

In knowledge-intensive QA, similar pipelines use **ontologies** or knowledge graphs. Zhao *et al.* (2025) describe a system where LLM statements are translated into a formal OWL ontology and checked for consistency with a reasoner; any contradictions are explained and fed back as prompt hints [14]. In the SymAgent framework, a planner agent extracts "symbolic rules" from a knowledge graph to decompose a question, then an executor agent uses tools (graph queries, retrieval) to assemble the answer [15]. These stages are delineated by custom tokens or query languages: for example, the KG might be queried with SPARQL sentences or action tokens, which the LLM learns to output as part of the prompt.

Another example is planning with formal languages: Kagitha *et al.* (2025) show that prompting an LLM to generate PDDL (a symbolic planning DSL) and then using a classical planner with feedback rounds dramatically outperforms free-text planning. They found that *iteratively revising* the LLM output with solver feedback more than **doubles** performance [7]. In these pipelines, the **PDDL tokens** act as a stable DSL front-end – grammar-constrained and validator-checked – while the LLM still provides the underlying logical structure.

Figure:



Task Memory Engine architecture: user instructions feed into the Task Relationship Reasoning Module (top), which consults a hierarchical Task Memory Tree (bottom) to generate prompts via the Prompt Synthesizer. This structure enables nested, symbolic planning [4] [5].

Examples of Symbolic DSL Structures

Throughout these systems, one sees recurring DSL patterns in prompts. Common examples include:

- **Task Dispatch Blocks:** Prompts often start with a clear *dispatch* line, e.g. `▷ START_TASK: CalculateVolume`. This single-token bullet (▷) could be a Unicode arrow that marks the task's beginning. By defining a fixed symbol, the model learns to parse "ACTION:..." reliably. Likewise, subtasks might be bracketed: `[Outline]… [/Outline]`, using a notepad emoji to enclose an outline sub-instruction.
- **Tool Call Markers:** Custom brackets or emojis can denote I/O interfaces. For example, use a wrench emoji 🔧 to prefix tool invocations (`🔧 Python: print(2+2)`) and a chart emoji 📊 to indicate results. In ARTIST-style prompts, one could write `<tool>…</tool>` literally, or use `【⚙】…【/⚙】` in plain text. The key is consistency: the model treats these pairs as atomic symbols around a code snippet.
- **Memory Access Tokens:** If a prompt engine supports external memory, special tokens can query it. For instance, `FETCH[UserHistory]` might fetch past conversation. In a symbolic DSL, this is just another sentinel: the LLM sees `` and knows to treat the bracketed part as a memory key. This ensures memory calls never merge with normal text.
- **Phase Delimiters:** In multi-phase workflows, authors often write something like `Phase 1: Research` as part of the prompt, with different colored squares to mark transitions. Or use rare separators like `§§§` around a new phase. These tokens must be chosen so that the tokenizer treats them as single units.

By rigidly enforcing these symbols, prompt engineers create a **mini-language** on top of the LLM. Every task has a templated structure: agent header, task ID, input section, output section, etc., each bounded by non-

ambiguous markers. This yields robust pipelines: the LLM cannot misunderstand phase changes or intermix tasks, because each segment is encapsulated in a token-stable DSL element.

Symbolic DSL Implementation (TypeScript Example)

Below is a simplified TypeScript mini-DSL framework for orchestrating an LLM agent workflow. It uses Unicode emojis and symbols as sentinel tokens for tasks, agents, and phases. The `AgentRoute` enum assigns each agent a unique emoji router, `Instruction` encapsulates prompts, and the `DSL` class assembles a structured prompt. This illustrates best practices discussed above: distinct token anchors, clear phase divisions, and unambiguous instruction formatting.

```
enum AgentRoute {
  Router = ' ',          // Orchestration agent marker
  Planner = '🗺',        // High-level planning agent
  Executor = '⚙',       // Execution/tool-calling agent
  Verifier = ' ',       // Verification agent
  Memory = '🗄'          // Memory-access agent
}

enum Sentinel {
  TaskStart = '▸',       // Start-of-task marker
  TaskEnd   = '◂',       // End-of-task marker
  Subtask   = '≡',       // Nested subtask marker
  PhaseSep  = ' | '      // Phase delimiter
}

// Represents a single instruction or command in the DSL
interface Instruction {
  agent: AgentRoute;
  action: string;        // e.g. code snippet, question, or recall request
}

// DSL builder for LLM prompts
class DSL {
  static startTask(taskName: string): string {
    return `${Sentinel.TaskStart} Task: ${taskName}`;
  }
  static endTask(taskName: string): string {
    return `${Sentinel.TaskEnd} End Task: ${taskName}`;
  }
  static subtask(name: string): string {
    return `${Sentinel.Subtask} [${name}]`;
  }
  static joinPhase(parts: string[]): string {
    return parts.join(` ${Sentinel.PhaseSep} `);
  }
```

```
  static formatInstruction(instr: Instruction): string {
    // Prefix the agent emoji and colon, then the action
    return `${instr.agent} ${instr.action}`;
  }
  static buildPrompt(taskName: string, instructions: Instruction[]): string {
    // Compose full prompt with start, each step, and end markers
    const lines = [this.startTask(taskName)];
    instructions.forEach(inst => {
      lines.push(this.formatInstruction(inst));
    });
    lines.push(this.endTask(taskName));
    return lines.join('\n');
  }
}

// Example usage: dispatching a task to a team of agents
const instructions: Instruction[] = [
  {agent: AgentRoute.Router, action: 'Define sub-tasks for user query.'},
  {agent: AgentRoute.Planner, action: 'Generate high-level steps to solve each
sub-task.'},
  {agent: AgentRoute.Executor, action: 'Execute plan for sub-task 1: Compute
data analytics.'},
  {agent: AgentRoute.Memory, action: 'Retrieve user's previous preferences from
memory.'},
  {agent: AgentRoute.Verifier, action: 'Check solution consistency and loop if
needed.'}
];
const prompt = DSL.buildPrompt('AnalyzeDataQuery', instructions);

console.log(prompt);
```

This produces a structured prompt such as:

```
▸ Task: AnalyzeDataQuery
  Define sub-tasks for user query.
🗺 Generate high-level steps to solve each sub-task.
⚙ Execute plan for sub-task 1: Compute data analytics.
🗄 Retrieve user's previous preferences from memory.
  Check solution consistency and loop if needed.
◂ End Task: AnalyzeDataQuery
```

Each line starts with a **single-token agent emoji** (ensuring token-stable routing), and the task is bounded by `▸`/`◂`. The phases or subtasks can be further divided using the `Sentinel.PhaseSep` marker (not shown above). In practice, this DSL could be fed to an LLM that has been instructed that each emoji maps to a role, and each sentinel symbol is a literal token boundary. By following these best practices, the system

enforces a clear, nested conversational structure — enabling deep multi-turn reasoning, tool calls, and context management in a reliable, machine-friendly way.

Sources: Recent research demonstrates the efficacy of such structured tokens and symbolic pipelines: e.g. ARTIST's tagged reasoning chains [2], chunk-summing sentinels [1], neuro-symbolic proof generation [13], ontology-driven consistency checks [14], and planner DLSs like PDDL [7] [6]. All examples above are informed by methods developed August 2024–May 2025 to ensure modern implementability.

---

[1] [2406.10985] Taking a Deep Breath: Enhancing Language Modeling of Large Language Models with Sentinel Tokens
https://arxiv.org/abs/2406.10985

[2] arxiv.org
http://arxiv.org/pdf/2505.01441

[3] arxiv.org
https://arxiv.org/pdf/2411.01077

[4] [5] Task Memory Engine (TME): Enhancing State Awareness for Multi-Step LLM Agent Tasks
https://arxiv.org/html/2504.08525v1

[6] [7] Addressing the Challenges of Planning Language Generation
https://arxiv.org/html/2505.14763v1

[8] [11] [12] Why Do Multi-Agent LLM Systems Fail?
https://arxiv.org/html/2503.13657v1

[9] [10] Symbolic Mixture-of-Experts: Adaptive Skill-based Routing for Heterogeneous Reasoning
https://arxiv.org/html/2503.05641v1

[13] arxiv.org
https://www.arxiv.org/pdf/2505.14479

[14] Enhancing Large Language Models through Neuro-Symbolic Integration and Ontological Reasoning
https://arxiv.org/html/2504.07640v1

[15] SymAgent: A Neural-Symbolic Self-Learning Agent Framework for Complex Reasoning over Knowledge Graphs
https://arxiv.org/html/2502.03283v1