SagaLLM: Context Management, Validation, and Transaction Guarantees for Multi-Agent LLM Planning

Edward Y. Chang
Stanford University
echang@cs.stanford.edu

Longling Geng
Stanford University
gll2027@stanford.edu

ABSTRACT

This paper introduces SagaLLM, a structured multi-agent architecture designed to address four foundational limitations of current LLM-based planning systems: unreliable self-validation, context loss, lack of transactional safeguards, and insufficient inter-agent coordination. While recent frameworks leverage LLMs for task decomposition and multi-agent communication, they often fail to ensure consistency, rollback, or constraint satisfaction across distributed workflows. SagaLLM bridges this gap by integrating the Saga transactional pattern with persistent memory, automated compensation, and independent validation agents. It leverages LLMs' generative reasoning to automate key tasks traditionally requiring hand-coded coordination logic, including state tracking, dependency analysis, log schema generation, and recovery orchestration. Although SagaLLM relaxes strict ACID guarantees, it ensures workflow-wide consistency and recovery through modular checkpointing and compensable execution. Empirical evaluations across planning domains demonstrate that standalone LLMs frequently violate interdependent constraints or fail to recover from disruptions. In contrast, SagaLLM achieves significant improvements in consistency, validation accuracy, and adaptive coordination under uncertainty—establishing a robust foundation for real-world, scalable LLM-based multi-agent systems.

PVLDB Reference Format:
Edward Y. Chang and Longling Geng. SagaLLM: Context Management, Validation, and Transaction Guarantees for Multi-Agent LLM Planning. PVLDB, 18(12): 4874 - 4886, 2025.
doi:10.14778/3750601.3750611

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/genglongling/SagaLLM.

1 INTRODUCTION

Multi-Agent Systems (MAS) have long been a cornerstone of distributed computing and database systems [15, 26, 29]. Over the past few decades, their development has followed two primary trajectories. In the database community, MAS traditionally integrated the foundational transaction processing principles, particularly ACID properties [22, 44], to ensure consistency and reliability for complex multi-step operations. For long-lived, distributed, or loosely-coupled tasks, MAS also adopted more flexible transactional models (e.g., Saga [18]) to maintain robustness while relaxing strict atomicity or isolation constraints.

Parallel to these database-oriented approaches, distributed systems research emphasized coordination protocols and flexible collaboration mechanisms [16, 45], enabling scalable multi-agent interactions without the overhead of strict locking or heavyweight transactional guarantees. These complementary development paths have resulted in frameworks optimized for different priorities: *transactional integrity* versus *adaptive coordination*, highlighting the fundamental trade-off between strong consistency and flexible execution in practical systems.

Recent advances in Large Language Models (LLMs) [12, 41, 51] have revitalized MAS as a paradigm for sophisticated reasoning and multi-agent collaboration [7, 8, 14]. Frameworks such as AutoGen [46], LangGraph [28], and CAMEL [30] demonstrate how LLM-based agents can decompose tasks, interact between modalities, and coordinate to solve complex problems. However, this resurgence often neglects the foundational transaction guarantees essential to reliable multi-agent workflows, particularly in domains requiring robust state management.

Unlike traditional MAS, LLM-based systems often lack mechanisms for maintaining strong consistency, failure recovery, and rollback handling, leading to inconsistent states, partial failures, and unreliable execution in real-world applications. These limitations stem from several fundamental challenges: LLMs struggle with *internal validation* due to inherent limitations highlighted by Gödel's incompleteness theorems [21], making them unreliable for detecting and correcting their own errors. Furthermore, *context loss* in long conversations [33, 35, 47] can cause LLMs to forget earlier steps, leading to contradictory decisions. When tasks are distributed across multiple agents, these problems are compounded, as no built-in supervisory mechanism exists to reconcile state changes or validate constraint satisfaction across agents.

For example, in a travel booking scenario, an LLM-based MAS can independently issue flight and hotel reservations without ensuring their coordinated success. If the flight is later canceled, the system may not recognize the inconsistency, leaving the hotel reservation active. Such scenarios illustrate the critical need for transactional frameworks that preserve the intelligence and adaptability of LLM-based MAS while ensuring consistency and reliability in long-running, interdependent workflows.

To address these limitations, we propose SagaLLM, a multi-agent transactional system that extends the Saga pattern, a transactional model originally developed to manage complex, long-lived transactions by decomposing them into smaller, independently validated, committed, and compensable units. By integrating transactional logic, compensatory rollback mechanisms, and *persistent memory*

Proceedings of the VLDB Endowment, Vol. 18, No. 12 ISSN 2150-8097.
doi:10.14778/3750601.3750611

into LLM-based MAS, SagaLLM ensures that each individual operation within a workflow is reliably validated and committed, with clearly defined compensating transactions that restore system-wide consistency in case of failure.

Crucially, SagaLLM leverages the reasoning and coding capabilities of LLMs to automate core aspects of transaction orchestration that previously required scenario-specific manual programming. SagaLLM enables LLMs to:

1. Identify the persistent states to track,
2. Validate constraints and inter-agent dependencies,
3. Design logging schemas to capture workflow transitions,
4. Develop compensatory logic for failure recovery, and
5. Implement communication protocols among agents to coordinate these behaviors.

Traditionally, each of these components required custom implementation by system developers, tailored to individual applications. In contrast, SagaLLM employs LLM as intelligent agents that automatically infer, generate, and coordinate these mechanisms, greatly improving scalability and reducing development overhead.

This hybrid approach—integrating transactional processing with adaptive multi-agent intelligence—makes SagaLLM particularly effective for real-world applications that are complex and demand reliability and safety in e.g., healthcare management, supply chain management, and emergency response. These capabilities collectively enable SagaLLM to overcome foundational limitations in LLM-based MAS, such as unreliable state coordination, lack of rollback support, and limited context retention, thereby supporting robust, scalable reasoning across distributed agents. We summarize our key contributions below:

1. **Transactional Consistency via Persistent Memory and Compensation**: SagaLLM introduces transactional safeguards and persistent-memory-based compensatory mechanisms to LLM-based MAS, ensuring reliable consistency and coherent state recovery across multi-agent workflows.
2. **Robust Constraint and Dependency Validation**: SagaLLM incorporates temporal-spatial context tracking and external verification mechanisms to validate inter-agent dependencies and prevent inconsistencies, addressing the fundamental limits of LLM self-verification.
3. **LLM-Orchestrated Intelligence**: SagaLLM automates key components of multi-agent planning—state tracking, constraint checking, log schema design, compensation logic, and coordination protocols—through the generative reasoning and coding capabilities of LLMs.

The remainder of this paper covers: related work (Section 2), problem definition (Section 3), architecture (Section 4), evaluation (Section 5), and conclusions (Section 6).

2 RELATED WORK

We review three strands of related work: (1) the evolution of transactional management, particularly the Saga pattern; (2) cognitive limitations of LLMs motivating the requirements for transactional integrity, independent validation, and context preservation; and (3) multi-agent LLM frameworks and recent attempts to integrate transactional safeguards.

2.1 Transaction Management Systems

Transactional models have evolved significantly since Gray introduced the ACID properties [22]. In distributed settings, strict ACID guarantees became impractical, prompting models such as BASE [37] and long-lived transaction patterns.

The Saga pattern by Garcia-Molina and Salem [18] decomposes long-lived transactions into smaller, locally atomic sub-transactions with compensating steps for failure recovery. This has influenced modern microservice architectures and workflow engines [38].

Systems such as YAWL [40], AWS Step Functions [1], and Azure Logic Apps [2] embed Saga-style workflows but remain rigid and manually defined, lacking dynamic adaptability. These foundational principles inform SagaLLM's adaptive, LLM-driven extensions.

2.2 LLM Limitations Necessitating SagaLLM's Key Requirements

To realize this complementary relationship between AI and workflow systems, we must first address the specific limitations that prevent current LLMs from functioning effectively in transactional workflows. In addition to what has been said about hallucinations [25], our analysis reveals three fundamental limitations that directly motivate SagaLLM's core requirements: transactional integrity, independent validation, and strategic context preservation.

Self-Validation Gap Necessitating Independent Validation. LLMs inherently lack robust self-validation mechanisms, a limitation originating from intrinsic boundaries identified by Gödel's incompleteness theorems, demonstrating fundamental constraints on a system's ability to verify its own reasoning [11, 21]. Recent research confirms that self-refinement techniques [27, 31, 34], while iterative and beneficial, are unable to surpass inherent capability ceilings to reliably correct deeper logical errors [24]. In transactional scenarios, these validation gaps manifest as factual inconsistencies, invalid operations, and unreliable plan feasibility assessments [48]. Thus, SagaLLM incorporates an independent validation framework to mitigate these inherent limitations.

Statelessness Necessitating Transactional Integrity. LLMs process each interaction independently, lacking native mechanisms to maintain the state across sequential interactions. This fundamental statelessness necessitates explicit transactional integrity management to maintain coherent operation sequences and ensure robust failure recovery. Without systematic transaction management, LLM-based systems risk state inconsistency, operation losses, and incoherent recovery procedures.

Context Limitations and Strategic Preservation. LLMs rely on self-attention mechanisms that prioritize recent tokens, leading to significant degradation in context retention over long sequences. Empirical studies reveal sharp drops in recall beyond token limits [35, 47], especially for mid-context information [23, 33]. Chain-of-thought heuristics [5] further exacerbate this by lacking mechanisms to manage or pass context reliably across steps. These limitations hinder multi-step reasoning [39, 43], as earlier outputs are frequently lost. SagaLLM addresses this by explicitly preserving vulnerable context elements—goals, justifications, and dependencies—through structured memory and persistent tracking.

Collectively, these limitations provide strong motivation to address all three key requirements within SagaLLM. Comprehensive transaction management, independent validation, and strategic context preservation are essential for reliably deploying LLM-based multi-agent systems (MAS) in critical real-world applications.

2.3 Multi-Agent LLM Frameworks and Transaction Limitations

Frameworks such as AutoGen [46], LangGraph [28], and CAMEL [30] advance multi-agent LLM coordination but fall short in addressing SagaLLM's three core requirements: transactional integrity, independent validation, and context preservation.

Missing Transaction Semantics. LangGraph and AutoGen enable structured workflows and agent interactions, but lack built-in atomicity guarantees, compensation logic, or robust failure recovery. AgentScope [17] and AFlow [50] introduce limited rollback mechanisms, but do not generalize across workflows.

Validation Gaps. Most frameworks are based on LLM self-validation, exposing them to reasoning errors and hallucinations. Systems such as PLASMA [4] improve reliability but omit transaction-level rollback. LLM-MCTS [52] and Tree-of-Thought [49] emphasize pre-execution reasoning without runtime consistency checks.

Limited Context Preservation. CAMEL preserves dialogue history but lacks mechanisms for tracking state transitions, inter-agent dependencies, or compensatory paths. Broader planning systems [42] do not offer persistent context retention strategies.

Unlike these systems, SagaLLM treats compensation, validation, and context tracking as first-class design goals, and ensures reliable, recoverable, and intelligent coordination across complex multi-agent workflows.

3 SYSTEM REQUIREMENTS

Building on the limitations identified in Section 2, this section formally defines SagaLLM's requirements. While SagaLLM inherits the core transactional semantics of classical Saga patterns [18], significant adaptations are required for LLM-based multi-agent systems, as summarized in Table 1.

To address foundational limitations in LLM-based execution, SagaLLM is organized around three tightly interwoven yet conceptually distinct requirements that extend classical transaction processing into the realm of adaptive multi-agent intelligence:

1. **Transactional Integrity:** Ensures that agent operations transition the system through coherent, globally consistent states. This is achieved through structured rollback mechanisms, compensating actions, and invariant preservation across interdependent agents. It also requires reliable tracking of system state to detect and repair inconsistencies triggered by partial execution or disruptions.
2. **Independent Validation:** Addresses the known limitations of self-validation of LLM by introducing global and cross-agent and external validation layers. These mechanisms evaluate agent outputs and inter-agent inputs against constraints, schemas, and dependency graphs. Persistent validation histories are maintained to support rollback triggers and guard against hallucinations or invalid commitments.
3. **Context Management:** Maintains essential state and dependency information across long-horizon workflows. Unlike systems that rely on ephemeral context windows, SagaLLM persistently stores goals, state histories, decision justifications for transitions and constraint resolutions, and compensation plans in structured memory. This enables agents to reason over consistent histories and perform accurate recovery after failures.

Table 1: Classical Saga vs SagaLLM Framework Comparison

Aspect	**Classical Saga**	SagaLLM
Domain	Database transactions	Multi-agent LLM workflows
Compensation	Pre-defined rollback procedures	LLM-generated + validated compensation
Validation	Schema/constraint validation	Independent LLM output validation
Context	Stateless transactions	Strategic context preservation
Coordination	Simple sequential execution	Complex multi-agent dependency management
Intelligence	Rule-based workflows	Adaptive LLM reasoning w/ transaction guarantees

This integrated design contrasts with prior systems that handle these aspects in isolation or without formal guarantees. The mutual reinforcement among these three pillars reflects the central challenge of LLM-based workflows: effective orchestration under uncertainty, without sacrificing consistency and correctness.

3.1 Transactional Integrity Requirements

SagaLLM provides transactional guarantees tailored for multi-agent workflows, extending classical transaction semantics across autonomous agent boundaries. Sequences $O = \{o_1, o_2, ..., o_n\}$ are operations treated as a unit of logical cohesiveness, where each o_i is locally atomic. If any operation fails, SagaLLM initiates compensatory actions to restore global consistency.

Applying O to a system state S must yield either a fully committed state S', or trigger a coherent rollback that returns the system to S, thereby avoiding partial or inconsistent outcomes. To ensure this, SagaLLM enforces the following properties:

Consistency Preservation. SagaLLM ensures that all state transitions respect global invariants I. If $S \models I$, then any resulting $S' \models I$, even when execution spans multiple agents.

Isolation Guarantees. SagaLLM guarantees that concurrently executing agents produce final states equivalent to some serial order, regardless of autonomy or internal decision processes.

Durability Assurance. SagaLLM guarantees persistence of committed states by durably recording execution outcomes and metadata necessary for fault recovery and compensatory execution.

To enforce these guarantees, SagaLLM maps each o_i to a local transaction T_i, paired with a compensating transaction C_i. In case of failure at step T_j, compensating actions are invoked in reverse:

Table 2: Transaction State Management in SagaLLM

Mechanism	Information Recorded
Application State (S_A)	
Domain Entities	• Application-domain objects • Entity states and status • Checkpoints and snapshots
Operation State (S_O)	
Execution Logs	• Operation inputs and outputs • Timestamps and execution status • Completion indicators
Decision Reasoning	• LLM-generated reasoning chains • Justifications and alternatives
Compensation Metadata	• Inverse operations • Preconditions and recovery state
Dependency State (S_D)	
Causal Dependencies	• Inter-operation constraints • Data and resource flow mappings
Constraint Satisfaction	• Boolean condition checks • Satisfaction evidence and timestamps

$$\text{Saga } S = \{T_1, T_2, ..., T_n, C_n, ..., C_2, C_1\}. \quad (1)$$

3.1.1 Transaction State Management. SagaLLM maintains a structured state representation in three orthogonal dimensions to support validation, compensation, and recovery, as detailed in Table 2.

- *Application State (S_A):* Application-domain objects, entity states and status, along with checkpoints and snapshots that capture the semantic state of the application at transaction boundaries.
- *Operation State (S_O):* Complete execution metadata including operation inputs and outputs, timestamps and execution status, LLM-generated reasoning chains with justifications, and compensation metadata for recovery operations. This state enables precise replay, debugging, and compensation by maintaining an audit trail of computational steps and decision rationales.
- *Dependency State (S_D):* Graph-structured representation of inter-operation constraints, data and resource flow mappings, and constraint satisfaction criteria with boolean condition checks and satisfaction evidence. This state tracks both explicit dependencies declared by operations and implicit dependencies discovered during execution, enabling intelligent scheduling and conflict resolution.

3.1.2 Dependency Tracking and Compensation Planning. SagaLLM models operation dependencies as a directed graph:

$$D = \{(o_i, o_j, c_{ij}) \mid o_j \text{ depends on } o_i \text{ under condition } c_{ij}\}. \quad (2)$$

To express more complex conditions:

$$c_{\{i_1,...,i_n\},j} = \mathcal{B}(c_{i_1 j}, ..., c_{i_n j}), \quad (3)$$

where $\mathcal{B}$ is a Boolean function under the prerequisite conditions.

Upon failure, SagaLLM traverses this graph to determine the minimal set of affected operations and executes compensatory actions that restore global consistency.

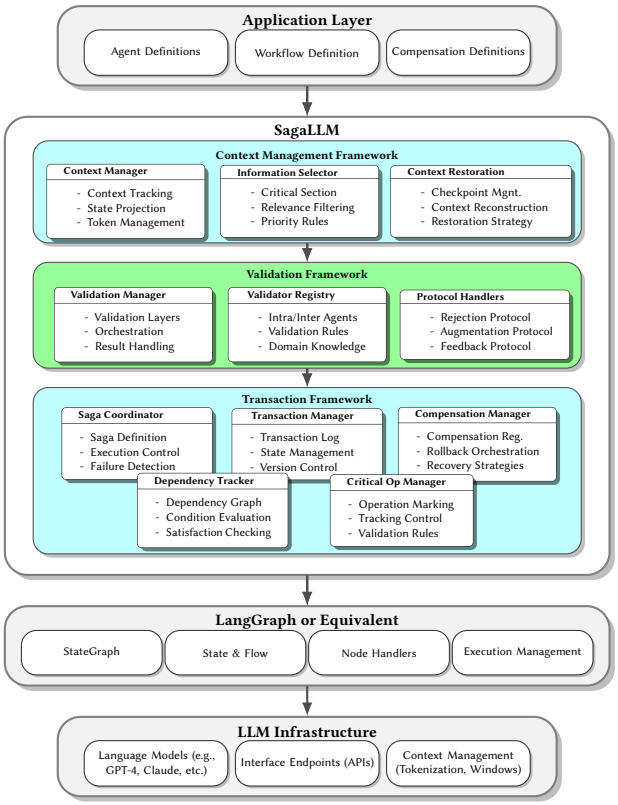

Figure 1: Architecture of SagaLLM. It sits between the application layer and LLMs, consisting of three frameworks: Context Management, Validation, and Transaction.

3.2 Independent Validation Requirements

To address the inherent limitations of LLM self-verification, SAGA introduces a two-tier validation architecture governed by a global validation agent. This agent operates independently of task agents and has visibility into the full transaction history, agent communications, and global state.

Intra-Agent Output Validation. The global validation agent inspects the outputs of the individual task agents before those outputs are committed or transmitted. Outputs are checked for: syntactic correctness (format, schema), semantic coherence and reasoning soundness, factual accuracy against context, constraint adherence and invariants, and context preservation and dependency awareness. Failures trigger compensations.

Inter-Agent Input and Dependency Validation. Inputs and messages between agents are validated before delivery. Checks include: contract conformance, dependency satisfaction, cross-agent consistency, temporal ordering, mutual agreement on shared state, and transaction coherence. Failed validations block message passing and invoke recovery.

Validation Response Protocols. SAGA defines structured validation outcomes:

- *Rejection:* Discard and compensate
- *Augmentation:* Enhance with clarifications
- *Feedback:* Record for future adaptation

3.3 Context Management Requirements

SagaLLM identifies and retains essential context for recovery, validation, and inter-agent dependencies:

- *Selective Retention:* Filters critical info
- *Structured Storage:* Organizes specs, justifications, and reasoning
- *Dependency Tracking:* Maintains prerequisites for rollback
- *Communication Protocol:* Ensures necessary context exchange

Failure Handling and Recovery. Effective recovery depends on preserved context and dependency tracking. SagaLLM supports multi-level failure response:

1. *Operation-Level:* Upon failure, the system invokes compensatory actions using logs and rollback specifications stored in S_O.
2. *Workflow-Level:* SagaLLM traverses the dependency graph to orchestrate reverse execution paths across agents, restoring global consistency based on S_D and recorded constraints.

This layered integration of recovery within strategic context management enables SagaLLM to meet the transactional demands of complex real-world multi-agent LLM workflows.

4 DESIGN AND IMPLEMENTATION

Figure 1 depicts the SagaLLM architecture, which sits between the application layer and LLM multi-agent systems like LangGraph. SagaLLM comprises three frameworks: *context management*, *validation*, and *transaction*. To illustrate the design, we use a simple travel planning example.

Travel Planning Problem

This example demonstrates how SagaLLM automatically manages complex multi-agent LLM workflows for international trip planning with multiple destinations, budget constraints, and transactional booking requirements. The application illustrates the transition from manual planning to automated SagaLLM-managed execution.

4.1 Specifications

- Plan a trip from San Francisco to Berlin and Cologne and then back to San Francisco.
- Travel period: June 2025 (flexible within the month)
- Budget constraint: $5,000 total.
- Required bookings: flights, hotels, and trains between cities.
- Preferences:
  - > Moderately priced accommodations (3-4 star hotels).
  - > Direct flights when possible.
  - > Train pass to save money.
  - > Flexible scheduling with 4 days in Berlin and 2 in Cologne.

4.2 Two-Phase Workflow Architecture

The application workflow consists of two distinct phases with different automation levels:

Phase 1 (Manual Planning): Human-driven itinerary planning and user validation.

Phase 2 (Automated SagaLLM **Execution)**: Fully automated multi-agent transaction management.

Figure 2 illustrates this workflow transition, where gray boxes represent manual planning activities and cyan boxes represent automated SagaLLM-managed transactions.

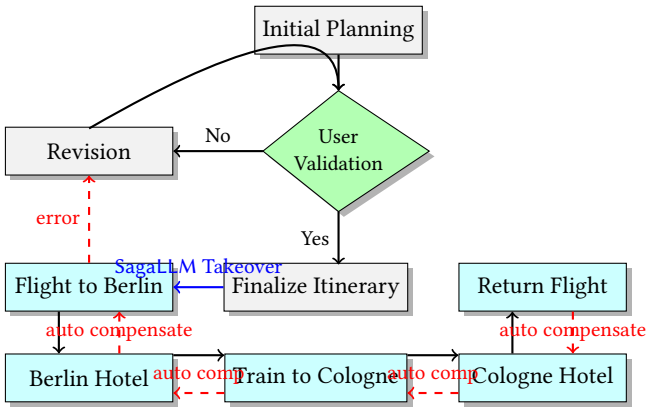

Figure 2: Travel Planning Workflow showing transition from manual planning to automated SagaLLM **execution. Gray boxes represent manual human-driven activities. Cyan boxes represent fully automated** SagaLLM**-managed transactions with automatic compensation, validation, and recovery. The blue arrow indicates the handoff point where** SagaLLM **takes complete control.**

4.2.1 Phase 1: Manual Itinerary Planning. Phase 1 involves traditional human-driven planning activities that establish the requirements and constraints for automated execution:

1. **Initial Plan Generation**: Human planners generate multiple feasible itineraries based on user requirements, specifying flight options, hotel reservations, and transportation with cost estimates validated against budget constraints.
2. **Iterative Refinement**: Users review proposed itineraries and provide feedback, leading to plan adjustments through manual coordination. Essential context including dates, preferences, and constraints is tracked to ensure complete handoff to SagaLLM.
3. **Plan Finalization and** SagaLLM **Handoff**: Users select and approve the final itinerary with all requirements and constraints. The system compiles comprehensive specifications including booking dependencies, budget limits, and user preferences, then initiates automatic handoff to SagaLLM. All subsequent operations—design, coding, agent coordination, and execution—become fully automated through LLM-driven orchestration.

4.2.2 Phase 2: Automated SagaLLM *Execution.* SagaLLM assumes complete control of workflow development and transaction management upon receiving the finalized specifications from Phase 1. Given a planning problem O, constraint set D, and performance metrics $\mathcal{M}$, SagaLLM autonomously generates a complete workflow consisting of nodes and edges, provides specifications for both transaction and compensation agents, and performs validation and refinement. Each workflow component is assigned both a primary agent for transaction execution and a compensation agent for rollback or compensation operations.

Algorithm 1 outlines SagaLLM's complete Phase 2 workflow and code generation process. The following sections detail the core architecture and validation mechanisms, while the extended version [11] provides additional coverage of advanced deployment scenarios and runtime optimization techniques.

Algorithm 1 Workflow $\mathcal{W}_{\text{template}}$ Construction and Agent Code Generation

Require: Problem specification O, constraints D, performance metrics $\mathcal{M}$
Local Variables:
Roles $\mathcal{R}$; Profiles $\mathcal{P}$; Nodes $\mathcal{N}$; Edges $\mathcal{E}$
Log schemas $\mathcal{L}_{n_i}, \mathcal{L}_{e_{ij}}$
Agents, Comp Agents $\alpha_{n_i}, \alpha_{e_{ij}}, \alpha_{n_i}^{comp}, \alpha_{e_{ij}}^{comp}$
Ensure: Validated $\mathcal{W}_{\text{template}} = (\mathcal{N}, \mathcal{E})$

Stage 1: Network Construction (extracting information from O)
$\mathcal{R} \leftarrow \text{ExtractRoles}(O)$
$\{(n_i, \mathcal{P}_i)\} \leftarrow \text{map}_{\text{role}}(O, \mathcal{R})$
$\mathcal{N} \leftarrow \{n_i\}, \mathcal{E} \leftarrow \text{map}_{\text{dep}}(\mathcal{N}, D)$
$\mathcal{W}_{\text{template}} \leftarrow (\mathcal{N}, \mathcal{E})$

Stage 2: Agent Specification (for each node and edge, creating its agent and compensation agent with logging schema)
for all $n_i \in \mathcal{N}$ **do**
    $\mathcal{L}_{n_i} \leftarrow \text{DefineLogSchema}(n_i, \mathcal{P}_{n_i})$
    $\alpha_{n_i} \leftarrow \text{DefineNodeAgent}(n_i, \mathcal{L}_{n_i})$
    $\alpha_{n_i}^{comp} \leftarrow \text{DefineCompAgent}(\alpha_{n_i}, \mathcal{L}_{n_i})$
for all $e_{ij} \in \mathcal{E}$ **do**
    $\mathcal{L}_{e_{ij}} \leftarrow \text{DefineLogSchema}(e_{ij}, \mathcal{P}_{e_{ij}})$
    $\alpha_{e_{ij}} \leftarrow \text{DefineEdgeAgent}(e_{ij}, \mathcal{L}_{e_{ij}})$
    $\alpha_{e_{ij}}^{comp} \leftarrow \text{DefineCompAgent}(\alpha_{e_{ij}}, \mathcal{L}_{e_{ij}})$

Stage 3: Validation and Refinement (code validation and refinement)
$\mathcal{W}_{\text{template}} \leftarrow \text{UpdateWorkflow}(\mathcal{N}, \mathcal{E}, \alpha, \alpha^{comp})$
while not $\text{ValidateWorkflow}(\mathcal{W}_{\text{template}}, \mathcal{M})$ **do**
    $\text{StructuralValidation}(\mathcal{W}_{\text{template}})$
    $\text{ConstraintValidation}(\mathcal{W}_{\text{template}}, D)$
    $\text{CompensationValidation}(\mathcal{W}_{\text{template}}), \{\alpha^{comp}\})$
    $\mathcal{W}_{\text{template}} \leftarrow \text{RefineWorkflow}(\mathcal{W}_{\text{template}}, \mathcal{M})$
return $\mathcal{W}_{\text{template}}$

4.2.3 Detailed Phase 2 Implementation. The automated execution phase consists of four integrated components:

1. **Automatic System Architecture Generation**: SagaLLM analyzes the finalized itinerary and automatically generates the appropriate agent architecture, defines the transaction sequences ($T_1, T_2, ..., T_n$) and the corresponding compensations ($C_1, C_2, ..., C_n$), and establishes validation rules and dependency graphs based on booking requirements.
2. **Automatic Agent Deployment and Coordination**: The system instantiates required domain agents (FlightBookingAgent, HotelBookingAgent, etc.) with appropriate configurations, deploys GlobalValidationAgent and SagaCoordinatorAgent with full system access, and configures agent communication protocols and data schemas.
3. **Automatic Transaction Execution**: The system executes the complete booking sequence: T_1 (International Flight Booking SFO → Berlin), T_2 (Berlin Hotel Booking coordinated with flight confirmation), T_3 (Train Booking Berlin → Cologne scheduled with hotel checkout), T_4 (Cologne Hotel Booking aligned with train arrival), and T_5 (International Return Flight Cologne → SFO coordinated with hotel checkout).
4. **Automatic Exception Handling and Recovery**: SagaLLM automatically detects validation failures and executes appropriate compensations, maintains system consistency without human intervention, automatically replans affected portions using preserved context and constraints, and only falls back to Phase 1 for human re-evaluation when automatic replanning cannot satisfy constraints.

4.3 Agent Architecture and Code Structures

Figure 3 presents the organization of the code implementation, structured into the application interface, SagaLLM core, and LangGraph integration components. The agent architecture consists of task execution agents that handle domain-specific operations and global coordination agents that manage system-wide consistency, validation, and compensation orchestration.

4.3.1 Task Execution Agents. Task execution agents focus solely on their domain-specific operations, with all validation handled externally by the global validation agent. Each agent maintains structured input/output interfaces and internal state for recovery.

FlightBookingAgent.
- *Input Schema*: travel_dates, budget_limit, airline_preferences, passenger_details
- *Output Schema*: flight_details, confirmation_number, total_cost, cancellation_policy
- *Internal State*: reservation_status, booking_reference, payment

HotelBookingAgent.
- *Input Schema*: checkin_date, checkout_date, location_constraints, amenity_preferences, budget_limit
- *Output Schema*: hotel_details, room_type, confirmation_number, total_cost, cancellation_policy
- *Internal State*: reservation_status, booking_reference, payment

TrainBookingAgent.
- *Input Schema*: departure_location, arrival_location, travel_time, connection_requirements
- *Output Schema*: train_details, seat_res., total_cost, schedule_details
- *Internal State*: ticket_status, booking_reference, refund_policy

BudgetTrackingAgent.
- *Input Schema*: expense_item, cost, category, transaction_id
- *Output Schema*: updated_total, remaining_budget, budget_status, expense_breakdown
- *Internal State*: cumulative_expenses, expense_log, constraints

ItineraryPlanningAgent.
- *Input Schema*: user_prefs, travel_constraints, confirmations
- *Output Schema*: optimized_itinerary, timing_schedule, activity recommendations
- *Internal State*: preference_history, optimization_parameters, constraint_violations

4.3.2 Global Coordination Agents.

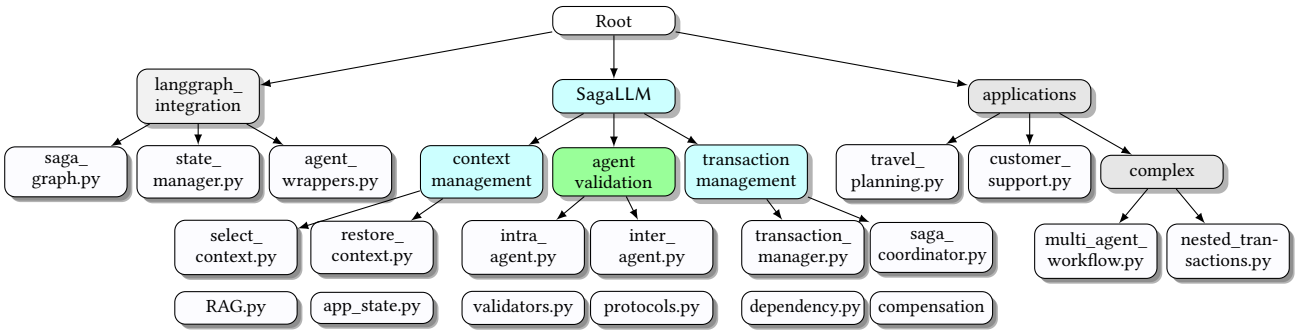

Figure 3: Directory Structure of SagaLLM Integration with LangGraph

GlobalValidationAgent. The central validation authority that maintains access to all system state and performs comprehensive validation before any transaction commitment.

- *System Access*: Complete visibility to all agent outputs, transaction history, dependency graph, and critical context
- *Validation Scope*: Intra-agent output validation and inter-agent communication validation (detailed in Table 3)
- *Response Protocols*: Rejection (triggers compensation), Augmentation (enhances outputs), Feedback (improves future performance)

SagaCoordinatorAgent. Manages transaction sequencing, dependency tracking, and compensation orchestration.

- *Coordination State*: active_transactions, dependency_graph, compensation_queue, transaction_log
- *Responsibilities*: Transaction ordering, failure detection, compensation sequence execution

4.3.3 Critical Context and State Management. SagaLLM maintains comprehensive context across three state dimensions, with specific agents responsible for different aspects:

Application State (S_A). Managed by task execution agents:

- *Travel Configuration*: travel_dates_per_city, destination_sequence, passenger_manifest
- *Booking Details*: confirmation_numbers, cancellation_policies, pricing_breakdown
- *User Constraints*: budget_limits, preference_profiles, accessibility_requirements

Operation State (S_O). Managed by SagaCoordinatorAgent:

- *Transaction Log*: transaction_id, agent_id, input_data, output_data, execution_timestamp
- *Decision Reasoning*: reasoning_chain, alternatives_considered, decision_justification
- *Compensation Actions*: compensation_procedure, rollback requirements, recovery_state

Dependency State (S_D). Managed by GlobalValidationAgent:

- *Inter-Booking Dependencies*: prerequisite_transactions, temporal_constraints, resource_dependencies
- *Validation Status*: validation_results, constraint_satisfaction, dependency_resolution

4.4 Transaction Flow and Validation Protocol

4.4.1 Transaction Execution Sequence. Each transaction follows a standardized execution pattern managed by SagaCoordinatorAgent with validation checkpoints enforced by GlobalValidationAgent:

1. *Pre-execution Validation*: GlobalValidationAgent validates inputs and dependency satisfaction
2. *Transaction Execution*: Task agent performs operation
3. *Output Validation*: GlobalValidationAgent performs comprehensive output validation (Table 3)
4. *State Commitment*: Upon validation success, results are committed to system state
5. *Compensation Registration*: SagaCoordinatorAgent records compensation procedures for potential rollback

4.4.2 Comprehensive Validation Framework. Table 3 details the validation types performed by the GlobalValidationAgent at each checkpoint. All validation occurs externally to task agents, ensuring independent eval of agent outputs and inter-agent communications.

4.5 Compensation and Recovery Mechanisms

4.5.1 Transaction-Specific Compensations. Each transaction maintains explicit compensation procedures executed by the SagaCoordinatorAgent upon validation failure:

Flight Booking Compensation (C_1).

- *Immediate Actions*: Cancel reservation, release seat, refund
- *State Restoration*: Reset booking status, clear confirmation numbers, restore budget allocation
- *Dependency Impact*: Trigger hotel and train booking re-evaluation based on new flight availability

Hotel Booking Compensation (C_2).

- *Immediate Actions*: Cancel reservation per hotel policy, refund
- *State Restoration*: Cancel room, restore budget allocation
- *Dependency Impact*: Notify itinerary planning for location-based activity adjustments

Train Booking Compensation (C_3).

- *Immediate Actions*: Cancel tickets per railway policy, refund
- *State Restoration*: Clear reservations, update travel schedule
- *Dependency Impact*: Recalculate inter-city travel times for dependent bookings

4.5.2 Recovery Protocol Execution. Upon validation failure, the system executes a structured recovery sequence:

Table 3: Validations Performed by GlobalValidationAgent

Validation Type	Implementation Example
Intra-Agent Output Validation	
Syntactic Validation	Verify JSON structure with required fields (departure_time, arrival_time, flight_number)
Semantic Validation	Confirm accommodation covers entire trip duration without gaps
Factual Validation	Maintain consistent travel times (45-minute hotel-to-train travel time)
Constraint Adherence	Enforce budget limits (total cost under $5,000 maximum)
Reasoning Validation	Verify logical decision chains (weather-based activity recommendations)
Inter-Agent Communication Validation	
Dependency Satisfaction	Ensure flight booking completion before hotel finalization
Consistency Checks	Standardize location data formats across all agents
Temporal Validation	Sequence budget finalization after all booking verifications
Mutual Agreement	Coordinate feasible travel times between transportation and itinerary agents
Transaction Boundary Integrity	Trigger compensation cascade when flight booking fails

1. **Failure Detection**: GlobalValidationAgent identifies validation failure and triggers compensation
2. **Dependency Analysis**: SagaCoordinatorAgent analyzes dependency graph to determine affected transactions
3. **Compensation Sequence**: Execute compensations in reverse dependency order ($C_n, C_{n-1}, ..., C_1$)
4. **State Verification**: GlobalValidationAgent confirms system state consistency after compensation
5. **Replanning Initiation**: Re-execute affected portion of workflow with preserved context and constraints

This integrated architecture ensures that SagaLLM's sophisticated validation, state management, and context preservation requirements are systematically implemented through clear agent responsibilities and structured coordination protocols.

5 EXPERIMENTS

Having established the theoretical foundation for SagaLLM and presented its architecture, we now empirically validate the effectiveness of the framework in addressing the fundamental limitations of current multi-LLM agent systems. As identified in Section 2, existing frameworks suffer from four critical shortcomings that prevent reliable deployment in complex, real-world scenarios: inadequate self-validation capabilities stemming from inherent LLM limitations, context narrowing that leads to information loss during extended workflows, lack of transactional properties that compromise consistency and recovery, and insufficient inter-agent coordination that results in workflow fragmentation.

The SagaLLM framework addresses these limitations through its three core requirements established in Section 3: independent validation to overcome self-validation gaps, automatic context preservation to maintain critical information throughout extended interactions, and comprehensive transactional integrity to ensure consistent state management and reliable recovery. The travel planning specification in the previous section illustrates how these requirements translate into practical system architecture, where manual planning (Phase 1) hands off to fully automated SagaLLM execution (Phase 2) with complete transaction management.

Our experiments aim to validate SagaLLM's effectiveness by measuring improvements in each of the four problematic areas.

1. We aim to compare SagaLLM-managed workflows against baseline multi-agent systems to demonstrate quantifiable improvements in i) validation accuracy, ii) context retention, iii) transactional consistency, and iv) coordination reliability.
2. We seek to demonstrate that SagaLLM can handle unexpected plan disruptions and automatically conduct effective reactive planning, combining the strengths of traditional distributed-MAS and LLM-based MAS while mitigating their limitations.

5.1 Experimental Design

We selected test cases from the REALM benchmark [20], which evaluates multi-agent systems on distinct problems spanning various complexity levels and coordination requirements. For our experiments, we focused on two medium-tier sequential planning challenges (problems #5 and #6) that test systematic workflow execution and dependency management, and two reactive planning challenges (problems #8 and #9) that evaluate dynamic adaptation and compensation capabilities. These problems provide comprehensive coverage of the scenarios where the four identified shortcomings most significantly impact system performance.

We evaluated four LLMs—Claude 3.7 [3], DeepSeek R1 [13], GPT-4o [36], and GPT-o1—alongside our proposed SagaLLM framework. All experiments were conducted between March 12 and 17, 2025. The source code of SagaLLM for conducting these experiments is available at [19].

5.2 Thanksgiving Dinner Problems: P6 and P9

Problem **P6** considers a Thanksgiving dinner scenario in which a family of five must return to their home in a Boston suburb for a 6 p.m. dinner. The problem involves coordinating departure times, managing travel logistics (including possible traffic delays), and ensuring timely arrival. Table 4 formalizes these challenges as a sequential planning problem. This scenario also lays the foundation for a more advanced disruption case, which has proven difficult for standalone LLMs, as discussed in **P9**.

5.2.1 Common Sense Augmentation. Figure 4 presents a feasible schedule planned by Claude 3.7. Similarly, GPT-4o was able to generate a viable plan to ensure dinner was started on time (figure is similar and therefore not shown). However, a subtle, yet important consideration that humans typically account for—but LLMs initially overlooked—is the time required for passengers to retrieve their luggage after landing. In practice, this process typically takes about 30 minutes before they exit the terminal.

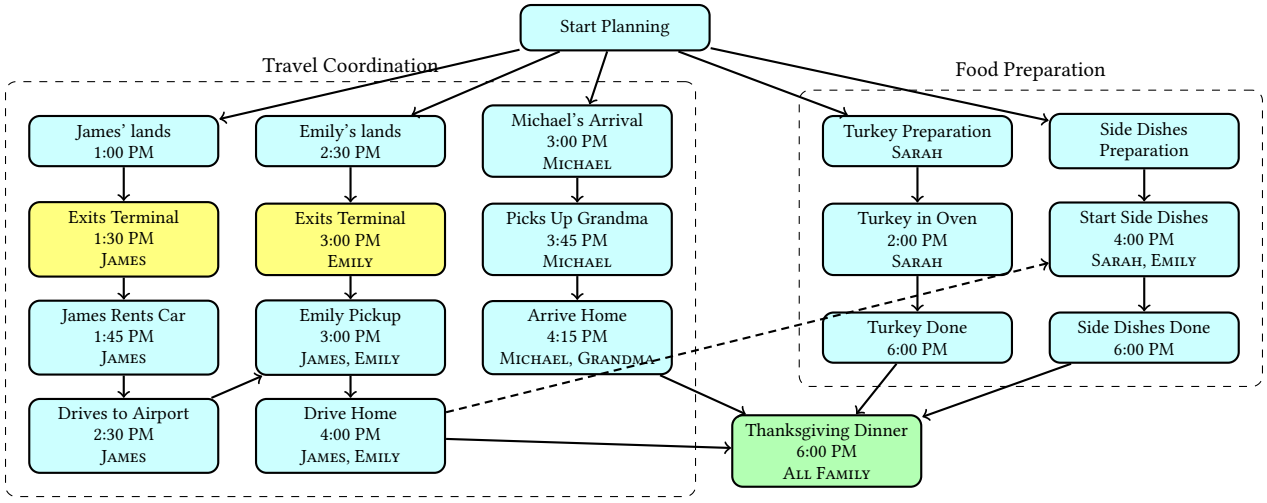

Figure 4: Thanksgiving Dinner Planning Workflow with Common Sense Augmentation, Generated by Claude 3.7

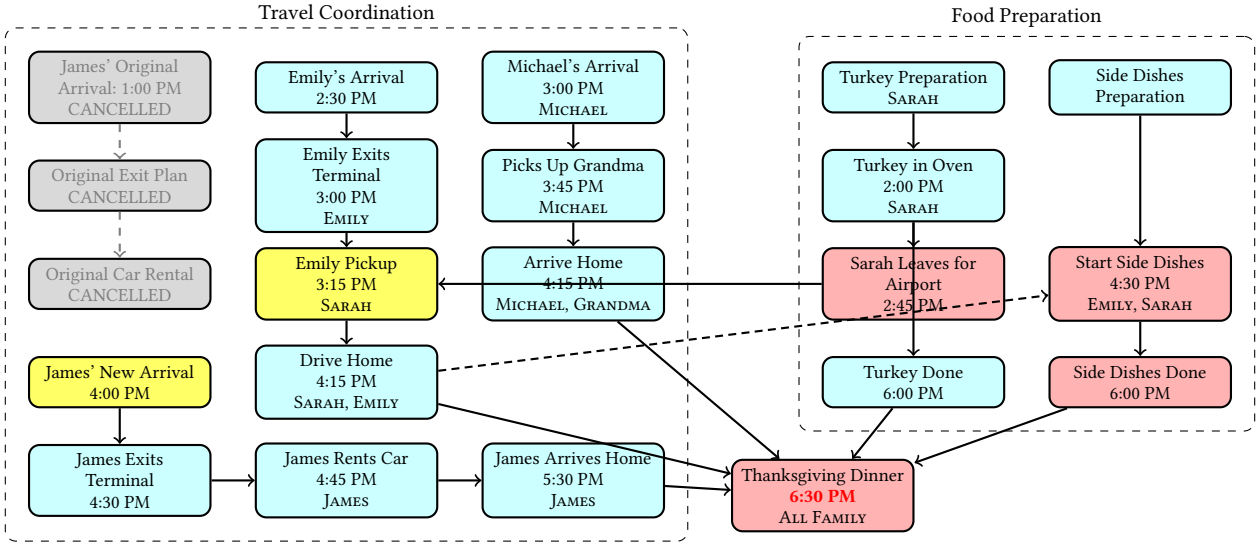

Figure 5: Reactive Planning for Thanksgiving Dinner After James' Flight Delay, by Claude 3.7. Red boxes highlight constraint violations, including travel time, fire safety, side-dish preparation, and dinner deadline.

To address this, *common-sense augmentation agent* was introduced into the plan. The yellow boxes in Figures 4 and 5 reflect this augmentation by introducing 30 minutes for James and Emily to exit the airport.

5.2.2 Context Narrowing. Next, we use problem **P9** to illustrate the attention-narrowing problem and the importance of independent validation. Problem **P9** is identical to the previous instance, except that at 1 PM, James notifies the group that his plane will land at 4 PM instead of 1 PM due to an emergency detour. Figure 5 shows that Claude 3.7's reactive planning introduces constraint violations:

- *Fire Safety*: Sarah is scheduled to leave home at 2:30 PM, leaving the oven unattended.
- *Travel Time*: The travel time between home and BOS should be one hour, but is scheduled to be only 30 minutes.
- *Side Dish Preparation*: The required preparation time is 2 hours, but only 90 minutes are allocated.
- *Dinner Time*: Dinner is now scheduled for 6:30 PM, violating the 6:00 PM constraint.

Each of these violations is perplexing, given that the constraints are explicitly stated in the context. Furthermore, after multiple iterations of reactive planning within the same thread, several constraints continue to be ignored or misinterpreted (e.g., cooking safety). This highlights a key limitation in the model's ability to maintain global constraint awareness over sequential planning steps due to attention narrowing, presented in Section 2.2.

When tested with GPT-o1, all constraints were correctly observed. However, in the final step, it added 30 additional minutes to James' driving time from Boston airport to home, citing potential

Table 4: Thanksgiving Dinner Coordination Problem

Objective: Coordinate family arrivals and dinner preparation for 6:00 PM dinner in Boston
Family Members and Arrivals:
- Sarah (Mom): Host, at home
- James (Dad): Lands at BOS 1:00 PM from SF
- Emily (Sister): Lands at BOS 2:30 PM from Chicago
- Michael (Brother): Driving, arrives 3:00 PM from NY
- Grandma: Needs pickup from suburban Boston

Cooking Requirements:
- Turkey: 4 hours cooking time
- Side dishes: 2 hours preparation
- Someone must stay home during cooking for fire safety

Transportation Constraints:
- James must rent car after landing
- Emily requires airport pickup
- Travel times:
  - Home to BOS Airport: 60 min
  - BOS Airport to Grandma's: 60 min
  - Home to Grandma's: 30 min

Key Requirements:
- All family members at home for 6:00 PM dinner
- Turkey and sides ready by dinner time
- All pickups completed with available drivers
- Cooking supervision maintained

traffic congestion. This kind of cleverness is, on the one hand, appreciated because the LLM injects common sense. However, it is also concerning, as an LLM may inject its own opinions at unpredictable stages in unpredictable ways. For common sense injection, human supervision would be preferable to ensure that the applied common sense reflects the shared understanding that is truly *common* between people, particularly those living in the Boston area.

This pattern suggests that during reactive planning within the same thread, the model fixates on recent adjustments while progressively disregarding earlier constraints. According to [32, 33], some context may be lost randomly, further contributing to systematic attention narrowing and planning inconsistencies.

5.2.3 SagaLLM ***Remediation: Context Management and Reactive Planning.*** To address the issue of context narrowing and loss, SagaLLM employs global *coordination and context management agents* (as depicted in Sections 4.3.2 and 4.3.3) to checkpoint historical state transitions, unresolved dependencies, and constraints. A key design criterion is to keep the agent's context small to prevent it from suffering from attention narrowing itself. Hence, we employ two validation agents: one for travel coordination and another for food preparation, each maintaining a context of less than $1k$.

The travel coordination agent records in external storage the temporal-spatial states of each individual and relevant temporal constraints. For problem **P9**, it stores the individual's *current state*, *next scheduled state-transition time*, and *all relevant constraints*.

When an unexpected event triggers reactive planning, all individuals roll back to the last saved state. The system then consolidates past and new constraints, resolving conflicts through "compensational" schedule cancellation before proceeding with rescheduling.

Context management ensures that:
- *Past history* is preserved and not inadvertently overridden.

Table 5: Wedding Reunion Logistics Problem

Metrics:
- **On-time performance:** Must arrive at the venue for 3:00 PM photos.

Locations: Four locations: $V = \{B, G, T, W\}$, where B is Boston Airport, G is Gift shop, T is Tailor shop, and W is Wedding venue.
Travel time: (minutes)
B-G : 45, B-T : 30, B-W : 40, G-T : 20, G-W : 25, T-W : 15.
Arrival Times:
- Alex: At B at 11:00 AM from Chicago (need a ride)
- Jamie: At B at 12:30 PM from Atlanta (need a ride)
- Pat: At W at 12:00 PM driving from NYC (has 5-seater car)

Required Tasks:
- Gift collection from G (after 12:00 PM)
- Clothes pickup from T (by 2:00 PM)
- Photos at W (3:00 PM sharp)

Available Resources:
- One car (5-seater) with Pat, available after he is Boston
- Local friend Chris (5-seater) available after 1:30 PM at W

Scheduling Constraints: - All tasks must complete before 3:00 PM photo time - Gift store opens at 12:00 PM - Tailor closes at 2:00 PM - Two cars must accommodate all transport needs

- *New dependencies and constraints* are properly restored (e.g., oven safety watch) and integrated.
- *Consistency across state transitions* is maintained.

By maintaining a structured history of constraint awareness, SagaLLM effectively mitigates LLM-driven attention narrowing and improves consistency in reactive temporal scheduling.

5.3 Wedding Gathering Problems: P5 and P8

Table 5 presents a wedding travel coordination problem (Problem **P5** in [20]). Several friends arrive at different times and locations before a 3:00 PM photo session. The challenge includes using two vehicles for airport pick-ups (for those unable to drive or saving costs) and completing key errands like collecting the wedding gift and retrieving formal attire. All activities must be scheduled to ensure timely arrival at the venue.

5.3.1 Context Narrowing (again). Table 6 presents an infeasible schedule generated by Claude 3.7, where Pat arrives at the tailor shop (T) after closing time: another example of attention narrowing. When queried about the error, Claude 3.7 admitted that it prioritized local route optimization while losing track of global constraints.

To remedy this, SagaLLM can enforce constraint validation checkpoints at 12:50 PM (evaluating whether to send Pat to T) or at 1:30 PM (when Chris becomes available to drive to T). These missed opportunities can be addressed by SagaLLM's validation protocols.

In contrast, GPT-o1 correctly schedules Pat to visit the tailor shop (T) first, ensuring it is open, before proceeding to the gift shop (G) and successfully completing both errands. (Due to space limitations, we do not present the successful results.)

However, both schedules overlook a more efficient alternative: Chris, who is available at 1:30 PM, could have handled both errands, balancing the workload and improving overall efficiency. The comparative travel routes for Pat and Chris are:
- Pat's route: W→B (40 min) + B→W (40 min) = 80 minutes.
- Chris's route: W→T (15 min) + T→G (20 min) + G→W (25 min) = 60 minutes.

Table 6: Wedding Reunion Logistics Schedule, by Claude 3.7. (Planning error rows in red)

Time	Activity	People
11:00 AM	Alex arrives at Boston Airport (B)	Alex
12:00 PM	Pat arrives at Wedding Venue (W)	Pat
12:00 PM	Gift Shop (G) opens	–
12:00–12:40	Pat drives from Wedding Venue (W) to Boston Airport (B)	Pat
12:30	Jamie arrives at Boston Airport (B)	Jamie
12:40–12:45	Pat picks up Alex at Boston Airport (B)	Pat, Alex
12:45–12:50	Pat picks up Jamie at Boston Airport (B)	Pat, A., J.
12:50–1:35	Drive from BOS to Gift Shop (G)	Pat, A., J.
1:30	Chris available at Wedding Venue (W)	Chris
1:35–1:50	Collect gift at Gift Shop (G)	Pat, A., J.
1:50–2:10	Drive from Gift Shop (G) to (T)	Pat, A., J.
2:00	Tailor Shop (T) closes	–
2:10–2:25	Pick up clothes at Tailor Shop (T)	Pat, A., J.
2:25–2:40	Drive from T to Wedding Venue (W)	Pat, A., J.
2:40 PM	Arrive at Wedding Venue (W)	Pat, A., J.
3:00 PM	Photo session at Wedding Venue (W)	All

5.3.2 Transaction Properties. Problem **P8** introduces a traffic alert: **Alert 1:00 PM:** Traffic Alert, an accident near Logan Airport in Boston **triples** all travel times to and from the airport! **Only** SagaLLM **correctly handles this alert.**

Table 7: Claude 3.7 Ignored Traffic Delay (errors in red)

Time	Activity	People
1:00	Traffic Alert: Accident near Airport triples travel times to/from airport	–
1:00	Current status: Pat, Alex, and Jamie en route from Airport (B) to Tailor (T)	Pat, A., J.
1:00–1:10	Emergency decision: Continue to (T)	Pat, A., J.
1:10–1:25	Arrive at (T), collect clothes	Pat, A., J.
1:25–1:45	Travel from Tailor (T) to Gift Shop (G)	Pat, A., J.
1:30	Chris is available at (W)	Chris
1:30–1:45	Chris drives from (W) to (G)	Chris
1:45–2:00	Both cars meet at (G), collect gift	P., A., J., C
2:00–2:25	Pat's car: Drive from (G) to (W)	P., A., J.
2:00–2:25	Chris' car: Drive from (G) to (W)	Chris
2:25	All arrive at Wedding Venue (W)	P., A., J., C.
3:00	Photo session at Wedding Venue (W)	All

This alert requires LLMs to replan in real time. Unfortunately, Claude 3.7, DeepSeek R1, and GPT-4o failed to react accurately to the new traffic constraints, and even GPT-o1 struggled with the precision of the planning. In contrast, SagaLLM can help remedy these shortcomings by maintaining both transaction state and history.

The following is a list of results from four LLMs:

* **Claude**: Table 7 shows that Claude 3.7 recognizes the accident, but does not update Pat's driving time from Boston Airport (departing at 12:50 PM) to the gift shop. In other words, Claude 3.7 fails to fully transition into the new alert state.
* **DeepSeek R1**: Table 8 demonstrates how DeepSeek R1 fails to maintain temporal consistency in reactive planning. When the traffic alert takes effect at 1:00 PM, DeepSeek discards its execution history and attempts to create a new plan starting from that point onward. Critically, it reassigns Pat to begin driving to the airport at 1:00 PM, even though Pat had already arrived at the airport by 12:40 PM under the original schedule. This "rewrite" of already-executed actions illustrates how LLMs can lose track of immutable past events when adapting to new conditions.
* **GPT-4o**: Similar to DeepSeek R1, GPT-4o exhibits temporal-spatial context confusion and violates multiple constraints, demonstrating that it struggles to adapt effectively once alerts are introduced mid-plan. *(Table not shown due to space limitations.)*
* **GPT-o1**: Table 9 shows GPT-o1's *conservative* plan in which Chris handles the tailor shop, avoiding potential delays for Pat. The solution is feasible but coarse-grained, as it doesn't leverage precise spatial-temporal reasoning about Pat's current position relative to the accident location. A more refined approach would first determine whether Pat has already passed the accident site by 1:00 PM, which could eliminate unnecessary detours and resource reallocations. This highlights the difference between merely finding a feasible solution versus optimizing based on detailed state information.

Table 8: DeepSeek's Failed Reactive Schedule After Traffic Alert, and GPT-4o Made Similar Errors (errors in red)

Time	Activity	People
1:00	Traffic alert received - Pat at W	System
1:05	Pat departs W for B	Pat
1:30	Chris becomes available at W	Chris
1:30	Chris departs W for T	Chris
1:45	Chris arrives at T for clothes	Chris
2:00	Chris departs T with clothes	Chris
2:15	Chris arrives at G for gifts	Chris
2:25	Pat arrives at B (delayed by traffic)	Pat
2:35	Pat departs B with Alex & Jamie	Pat
2:40	Chris departs G with gifts	Chris
2:55	Chris arrives at W	Chris
3:55	Pat's group arrives at W (Late)	Pat

*5.3.3 **LLM Limitations and** SagaLLM **Remediation**.* This study reveals critical limitations in how modern LLMs handle disruptions in planning scenarios:

- **State Maintenance Failure:** When an alert occurs, these LLMs might discard the partial context of already-completed actions, attempting to generate entirely new plans rather than adapt existing ones. This reveals their inability to reason about the continuous flow of time in real-world scenarios.
- **Temporal Inconsistency:** They attempt to modify immutable past events.
- **Position Tracking:** Agent locations are lost at critical intervals.
- **Path Dependency:** Models cannot recognize that different segments of a journey may be differently impacted by an alert.

By contrast, SagaLLM implements a comprehensive remediation approach through fine-grained compensation:

- **Persistent Context Repository:** SagaLLM maintains an external state repository that captures the complete world state at each checkpoint, enabling reliable rollback and forward projection regardless of attention constraints in the planning agent.

Table 9: Wedding Reunion Reactive Schedule, by GPT-o1

Time	Activity	People
11:00 AM	Alex arrives at Airport (B).	Alex
12:00 PM	Pat departs for Airport from (W).	Pat
12:30 PM	Jamie arrives at Airport (B).	Jamie
12:40–12:50 PM	Pat arrives at (B), picks up Alex and Jamie; departs at 12:50 PM.	Pat, A., J.
12:50–1:00 PM	**Drive (B → W) under normal conditions for first 10 minutes.**	Pat, A., J.
1:00–2:30 PM	**Traffic Alert starts: remaining distance (30 min normal) becomes 90 min. Arrival at W by 2:30 PM.**	Pat, Alex, Jamie
1:30 PM	Chris available at (W). Departs for Tailor.	Chris
1:30–1:45	Drive (W → T).	Chris
1:45–1:50	Pick up clothes at (T), closes at 2:00 PM.	Chris
1:50–2:10	Drive (T → G).	Chris
2:10–2:15	Purchase gift at (G).	Chris
2:15–2:40	Drive (G → W).	Chris
2:30 PM	Pat, Alex, Jamie arrive at (W).	Pat, A., J.
2:40 PM	Chris back at (W) with clothes and gift.	Chris
3:00 PM	Wedding photo session at (W).	Everyone

- **Immutable Action Logging:** All executed actions are recorded as immutable transactions in a persistent log, ensuring that historical events remain consistent even when replanning occurs, preventing the "amnesia effect" common in LLM planners.
- **Compensatory Planning:** When disruptions occur, SagaLLM doesn't simply replan from scratch but applies compensatory actions specifically designed to address the deviation while preserving as much of the original plan as possible.
- **Constraint Consistency Validation:** The system continuously validates that new plans remain consistent with both physical limitations and temporal dependencies established in earlier planning phases.

SagaLLM *Compensatory Analysis:* When faced with disruptions (e.g., the 1:00 PM traffic alert in our wedding scenario), SagaLLM executes a structured compensation process:

$$T_{\text{affected}} = \max(0, T_{\text{total}} - T_{\text{elapsed}}) \tag{4}$$

$$T_{\text{new}} = T_{\text{elapsed}} + (M \cdot T_{\text{affected}}) \tag{5}$$

This approach enables three key capabilities: (1) partial journey compensation for route segments, (2) strategic resource reallocation when needed, and (3) principled constraint relaxation with appropriate compensatory actions. Here, the key state to facilitate a precise resolution is to answer the question: "Has Pat's vehicle passed the accident location at 1:00 PM (and hence unaffected)?" The answer determines the remaining time required to reach the originally scheduled destination, the Tailor. In such a case, no rescheduling is required. If Pat's car is unfortunately involved in the accident, a more comprehensive replanning approach would be necessary to accommodate this significant disruption.

5.4 Observations

Our experiments across multiple LLMs (GPT-o1, DeepSeek R1, Claude 3.7, GPT-4o) highlight consistent limitations in complex planning scenarios. While GPT-o1 showed partial historical awareness, all models exhibited attention narrowing, self-validation failure, and inconsistent spatial-temporal reasoning.

Table 10: LLMs vs. SagaLLM on Context Management

Capability	Standard LLMs	SagaLLM
Maintains historical actions	Partial/None	Full
Partial journey compensation	Rarely	Always
Constraint consistency checking	Ad-hoc	Systematic
Handles attention narrowing	Vulnerable	Resistant
Physical-temporal consistency	Inconsistent	Guaranteed

Table 10 summarizes SagaLLM's context management and compensation mechanisms directly address these limitations of LLMs.

6 CONCLUSION

We introduced SagaLLM, a structured transactional multi-agent framework that addresses four fundamental limitations of existing LLM-based planning systems: inadequate self-validation, context narrowing, absence of transaction properties, and insufficient inter-agent coordination.

Our experiments demonstrate that even advanced LLMs like Claude 3.5 and GPT-o1 often fixate on recent context while neglecting critical earlier constraints, particularly in reactive planning scenarios where models attempt to retroactively rewrite past actions rather than adapting from the current state. Moreover, LLMs cannot consistently validate their own adherence to constraints. Critically, as noted by LeCun's critique, current LLMs lack persistent memory capabilities, an essential component for guaranteeing transaction properties and maintaining consistent state across long-lived tasks.

SagaLLM overcomes these limitations with four key innovations:

1. **Independent validation** to address self-validation gaps
2. **Strategic context preservation** to mitigate context narrowing
3. **Transactional state management** with immutable records and compensation mechanisms
4. **Specialized agent coordination** with explicit role distribution and dependency tracking

These contributions enable robust planning across diverse scenarios, from travel logistics to sequential and reactive planning tasks. By enforcing rigorous transactional validation among specialized agents, SagaLLM ensures consistency, reliability, and adaptability for mission-critical applications. Due to space constraints in this experience paper, detailed algorithm specifications (in particular, for reactive planning) and extended empirical studies on job-shop scheduling (JSSP) and supply chain management problems are provided in our companion ALAS paper [11].

Future work will focus on formal verification methods for the LLM-generated compensation code, addressing autoregressive context limitations, and extending SagaLLM to domains requiring scientific reasoning [6, 9, 10] and decision-making under uncertainty.

ACKNOWLEDGMENT

I am deeply grateful to my late advisor, Hector Garcia-Molina, for his invaluable mentorship and for pioneering **Saga**, the foundational inspiration for this work.

REFERENCES

[1] AWS Step Functions. https://aws.amazon.com/step-functions/, 2023. Accessed: 2025-03-04.

[2] Azure Logic Apps. https://azure.microsoft.com/en-us/products/logic-apps/, 2023. Accessed: 2025-03-04.

[3] Anthropic. Claude Technical Report. Technical report, 2024. URL https://www.anthropic.com.

[4] Faeze Brahman, Chandra Bhagavatula, Valentina Pyatkin, and Yejin Choi. PLASMA: Making small language models better procedural knowledge models for (counterfactual) planning. In *International Conference on Learning Representations (ICLR)*, 2024.

[5] Mert Cemri, Melissa Z. Pan, Shuyi Yang, Lakshya A. Agrawal, Bhavya Chopra, et al. Why do multi-agent LLM systems fail? *arXiv preprint arXiv:2503.13657*, 2025. URL https://arxiv.org/abs/2503.13657.

[6] Edward Y. Chang. Prompting large language models with the socratic method. In *IEEE 13th Computing and Communication Workshop and Conference*, 2023.

[7] Edward Y. Chang. Examining GPT-4's capabilities and enhancement with SocraSynth. In *The 10th International Conference on Computational Science and Computational Intelligence*, December 2023.

[8] Edward Y. Chang. EVINCE: Optimizing adversarial LLM dialogues via conditional statistics and information theory. *arXiv preprint arXiv:2408.14575*, August 2024.

[9] Edward Y. Chang. *Multi-LLM Agent Collaborative Intelligence: The Path to Artificial General Intelligence*. ACM Books (accepted), 2025. Amazon (March 2024).

[10] Edward Y. Chang. The unified cognitive consciousness theory (UCCT) for language models: Anchoring semantics, thresholds of activation, and emergent reasoning. *arXiv preprint arXiv:2506.02139*, 2025.

[11] Edward Y. Chang and Longling Geng. ALAS: A stateful multi-LLM agent framework for disruption-aware planning. *arXiv preprint arXiv:2505.12501*, 2025. URL https://arxiv.org/abs/2505.12501.

[12] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, et al. A survey on evaluation of large language models. *ACM Transactions on Intelligent Systems and Technology*, 15(3), March 2024. ISSN 2157-6904.

[13] DeepSeek-AI, Daya Guo, Dejian Yang, et al. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025. URL https://arxiv.org/abs/2501.12948.

[14] Yilun Du, Shuang Li, Antonio Torralba, Joshua B. Tenenbaum, and Igor Mordatch. Improving factuality and reasoning in language models through multiagent debate. *arXiv:2305.14325*, 2023. URL https://arxiv.org/abs/2305.14325.

[15] Barbara Dunin-Keplicz and Rineke Verbrugge. *Teamwork in Multi-Agent Systems: A Formal Approach*. Wiley Series in Agent Technology. Wiley, 2010. URL https://api.semanticscholar.org/CorpusID:26838202.

[16] Edmund H. Durfee. Distributed problem solving and planning. In *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*, pages 121–164. MIT Press, Cambridge, MA, USA, 1999. ISBN 0262232030.

[17] Dawei Gao, Zitao Li, Xuchen Pan, Weirui Kuang, Zhijian Ma, et al. AgentScope: A flexible yet robust multi-agent platform. *arXiv preprint arXiv:2402.14034*, 2024. URL https://arxiv.org/abs/2402.14034.

[18] Hector Garcia-Molina and Kenneth Salem. Sagas. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, SIGMOD '87, pages 249–259, New York, NY, USA, 1987. Association for Computing Machinery. ISBN 0897912365. doi: 10.1145/38713.38742. URL https://doi.org/10.1145/38713.38742.

[19] Longling Geng. Source code for SagaLLM paper experiments. https://github.com/genglongling/SagaLLM, 2025.

[20] Longling Geng and Edward Y. Chang. Realm-Bench: A real-world planning benchmark for LLMs and multi-agent systems. *arXiv:2502.18836*, 2025.

[21] Kurt Gödel. On formally undecidable propositions of Principia Mathematica and related systems I. In Jean van Heijenoort, editor, *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931*, pages 596–616. Harvard University Press, 1967. Translated by Jean van Heijenoort.

[22] Jim Gray. The transaction concept: Virtues and limitations. In *Proceedings of the Seventh International Conference on Very Large Data Bases*, volume 7 of *VLDB '81*, pages 144–154. VLDB Endowment, 1981.

[23] Cheng-Yu Hsieh, Yung-Sung Chuang, Chun-Liang Li, Zifeng Wang, Long T. Le, et al. Found in the middle: Calibrating positional attention bias improves long context utilization. *arXiv:2406.16008*, 2024. URL https://arxiv.org/abs/2406.16008.

[24] Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, et al. Large language models cannot self-correct reasoning yet. In *International Conference on Learning Representations (ICLR)*, 2024.

[25] Xu Huang, Weiwen Liu, Xiaolong Chen, Xingmei Wang, Hao Wang, Defu Lian, and more. Understanding the planning of llm agents: A survey. *arXiv:2402.02716*, 2024. URL https://arxiv.org/abs/2402.02716.

[26] Nicholas R. Jennings. Commitments and conventions: The foundation of coordination in multi-agent systems. *Knowledge Engr. Review*, 8(3):223–250, 1993.

[27] Dongwei Jiang, Jingyu Zhang, Orion Weller, Nathaniel Weir, Benjamin Van Durme, et al. Self-[in]correct: LLMs struggle with refining self-generated responses. *CoRR*, 2024.

[28] LangChain AI. LangGraph: Building structured applications with LLMs. https://github.com/langchain-ai/langgraph, 2024.

[29] Victor Lesser, Charles L. Ortiz Jr., and Milind Tambe. *Distributed Sensor Networks: A Multiagent Perspective*, volume 9. Springer Science & Business Media, 2004.

[30] Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. CAMEL: Communicative agents for "mind" exploration of large language model society. *arXiv preprint arXiv:2303.17760*, 2023. URL https://arxiv.org/abs/2303.17760.

[31] Yingcong Li, Kartik Sreenivasan, Angeliki Giannou, Dimitris Papailiopoulos, and Samet Oymak. Dissecting chain-of-thought: Compositionality through in-context filtering and learning. *arXiv preprint arXiv:2305.18869*, 2023. URL https://arxiv.org/abs/2305.18869.

[32] Dancheng Liu, Amir Nassereldine, Ziming Yang, Chenhui Xu, Yuting Hu, et al. Large language models have intrinsic self-correction ability. *arXiv preprint arXiv:2406.15673*, 2024. URL https://arxiv.org/abs/2406.15673.

[33] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, et al. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2024. doi: 10.1162/tacl_a_00638. URL https://aclanthology.org/2024.tacl-1.9/.

[34] Aman Madaan and Amir Yazdanbakhsh. Text and patterns: For effective chain of thought, it takes two to tango. *arXiv preprint arXiv:2209.07686*, 2022. URL https://arxiv.org/abs/2209.07686.

[35] Ali Modarressi, Hanieh Deilamsalehy, Franck Dernoncourt, Trung Bui, Ryan A. Rossi, et al. NoLiMa: Long-context evaluation beyond literal matching. *arXiv preprint arXiv:2502.05167*, 2025. URL https://arxiv.org/abs/2502.05167.

[36] OpenAI. Hello GPT-4o. https://openai.com/index/hello-gpt-4o/, 2024. Accessed: Jan. 30, 2025.

[37] Dan Pritchett. BASE: An ACID alternative. *Queue*, 6(3):48–55, 2008.

[38] Chris Richardson. *Microservices Patterns: With Examples in Java*. Manning Publications, Shelter Island, NY, USA, 2018.

[39] Kaya Stechly, Karthik Valmeekam, and Subbarao Kambhampati. Chain of thoughtlessness? an analysis of CoT in planning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024. URL https://arxiv.org/abs/2405.04776.

[40] Wil M. P. van der Aalst and Arthur H. M. ter Hofstede. YAWL: Yet another workflow language. *Information Systems*, 30:245–275, 2005. URL https://api.semanticscholar.org/CorpusID:205487187.

[41] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, et al. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 30, pages 5998–6008, 2017.

[42] Hui Wei, Zihao Zhang, Shenghua He, Tian Xia, Shijia Pan, et al. PlanGenLLMs: A modern survey of LLM planning capabilities. *arXiv preprint arXiv:2502.11221*, 2025. URL https://arxiv.org/abs/2502.11221.

[43] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, et al. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, NIPS '22, 2022.

[44] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, San Francisco, CA, USA, 2001.

[45] Michael Wooldridge. *An Intro. to Multiagent Systems*. John Wiley & Sons, 2009.

[46] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, and Chi Wang. AutoGen: Enabling next-gen LLM applications via multi-agent conversation. In *Conference on Language Modeling (COLM)*, August 2024.

[47] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. *arXiv preprint arXiv:2309.17453*, 2024. URL https://arxiv.org/abs/2309.17453.

[48] Khurram Yamin, Shantanu Gupta, Gaurav R. Ghosal, Zachary C. Lipton, and Bryan Wilder. Failure modes of LLMs for causal reasoning on narratives. *arXiv preprint arXiv:2410.23884*, 2024. URL https://arxiv.org/abs/2410.23884.

[49] Shinnosuke Yao, Dong Yu, Jianfeng Zhao, Izhak Shafran, Thomas Griffiths, et al. Tree of thoughts: Deliberate problem solving with large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 36, 2024.

[50] Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, et al. AFlow: Automating agentic workflow generation. *arXiv preprint arXiv:2410.10762*, 2024. URL https://arxiv.org/abs/2410.10762.

[51] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2025. URL https://arxiv.org/abs/2303.18223.

[52] Zirui Zhao, Wee Sun Lee, and David Hsu. Large language models as commonsense knowledge for large-scale task planning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.