

# ALAS: MULTI-AGENT LLM PLANNING SYSTEM VIA VALIDATOR ISOLATION AND LOCALIZED CASCADING REPAIR PROTOCOL

**Anonymous authors**

Paper under double-blind review

## ABSTRACT

Large language models enable flexible multi-agent planning but struggle with reliability: verification is often circular, state changes are not tracked for repair, and small faults trigger costly global replanning. We present *ALAS*, a multi-agent LLM Planning framework that separates planning from non-circular validation and performs localized repair guided by versioned execution logs. The validator operates independently of the planning LLM with fresh, bounded context, avoiding self-check loops and mid-context attrition. The repair protocol edits only the minimal affected region of the plan while preserving work-in-progress. On urban ride-sharing and job-shop scheduling across five classical benchmarks, the system matches or exceeds state-of-art Single-LLM and Multi-Agent System baselines, which achieves 83.7% success rate, reduces 60% token usage, and 1.82× Faster. A minimal reliability study shows that the validator detects injected structural faults with low overhead, and localized repair contains runtime perturbations with bounded edit radius and reduced makespan degradation versus global recompute. Code and seeds will be released. Results indicate that *ALAS* the combination of validator isolation and localized repair with execution logs provides measurable efficiency, optimality, and scalability for multi-agent LLM planning.

## 1 INTRODUCTION

Large language models (LLMs) have transformed language understanding and generation (Luo et al., 2025; Matarazzo & Torlone, 2025; Minaee et al., 2025). Yet when used for *planning*, standalone LLMs remain brittle: they often produce incomplete or inconsistent action sequences, violate constraints, and struggle to revise partial plans under change. These failures become acute in settings that require long-range consistency, multi-entity coordination, and reactive behavior.

The causes are structural. First, verification is frequently *circular*: the same model (or context) that proposes a plan is asked to approve it (Hong et al., 2024), inviting rubber-stamping. Second, long contexts are prone to information loss and mid-context attrition (Hsieh et al., 2024; Liu et al., 2024; Vaswani et al., 2017; Xiao et al., 2024). Third, maximum-likelihood decoding biases search toward high-probability but not necessarily high-feasibility completions (Chang, 2023; Holtzman et al., 2020; Radford et al., 2019). Finally, without external state, LLMs cannot reliably track commitments, causal dependencies, or temporal constraints, leading to cascading errors across reasoning chains (Chu et al., 2024a; Patel et al., 2024; Xiong et al., 2024).

Dynamic environments amplify these issues. In logistics, event coordination, or industrial operations, late requests, delays, and resource failures require *local* edits to in-flight plans. Global recomputation—whether via classical optimization or re-prompting a single LLM—can be counterproductive: small disturbances trigger large plan changes, harming latency and stability, and often break feasibility along the way.

**Our approach.** We study *efficiency*, *optimality*, and *scalability* in multi-agent LLM planning as a systems property of the end-to-end process. We present Adaptive LLM Agent System (ALAS), a stateful, disruption-aware framework that (i) separates *planning* from *non-circular validation* (validator isolation), (ii) records state transitions in a *versioned execution log* that provides restore points and auditability, and (iii) applies a *localized repair* protocol that edits only the minimal

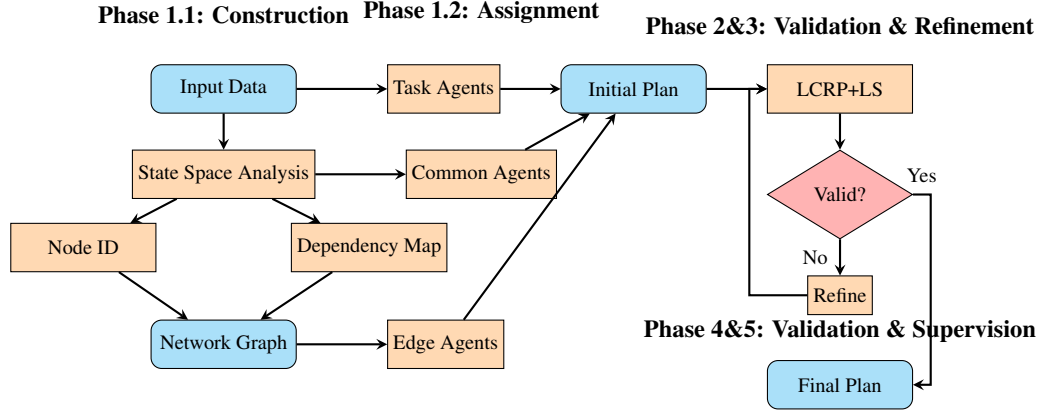


Figure 1: ALAS Planning Layer #1 Architecture. Color Scheme: cyan for input/output/intermediate results, orange for processes, and red for decisions. This figure illustrates how the meta-planner generates a planning workflow template  $\mathcal{W}_{\text{template}}$  with phases arranged from left to right.

affected region of the plan while preserving work-in-progress (WIP). By grounding validation in fresh, bounded prompts over the execution log, ALAS avoids self-check loops and mid-context drift; by repairing locally, it contains faults without costly global replanning.

**Agent roles.** Rather than a monolithic planner, ALAS orchestrates lightweight agents specialized to known failure modes:

1. *Validator agents* (isolated from planners) check feasibility against constraints and execution log.
2. *Repair agents* compute change sets under constraints to localize edits when disruptions occur.
3. *Context agents* operate in semantically scoped subcontexts to mitigate long-context attrition.
4. *Monitoring agents* detect anomalies/events and trigger validation or repair.
5. *Execution-log modules* maintain versioned state and restore points for audit and recovery.

#### Contributions.

1. *Validator isolation.* We decouple planning and verification: a dedicated validator operates with fresh, bounded context over the execution log, preventing circular self-approval and mitigating mid-context loss.
2. *Localized repair.* A disruption-aware protocol computes bounded edit sets that preserve WIP and constraints, avoiding brittle global recomputation.
3. *Versioned execution logs.* We record state transitions and restore points for auditable updates and targeted recovery (without relying on monolithic long contexts).
4. *Evidence across domains.* On URS and job-shop scheduling (DMU, TA), our framework matches or exceeds strong single-LLM and classical baselines, with the largest gains under disruption. We further provide *minimal reliability evidence*: a small fault-injection and runtime-perturbation study showing high detection rates for injected structural faults and bounded repair with modest token/latency overhead.

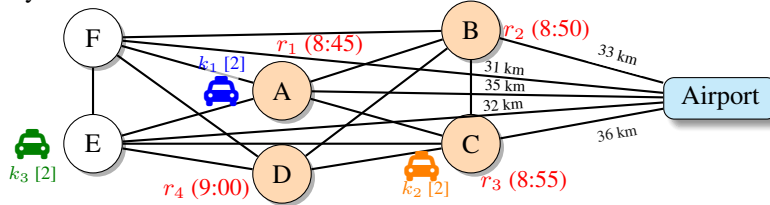


Figure 2: Network  $G = (V, E)$  with urban travel times  $\tau_{ij} = 10$  minutes and airport routes distance specified on the figure. Static scenarios can be solved by MILP or Column Generation. Dynamic scenarios (e.g., an accident, a cancellation, new passenger requests) must be addressed by ALAS. We adopt *Urban Ride Sharing* (URS) as a running example (Fig. 2): multiple vehicles must deliver passengers under delays and last-minute requests. Unlike TSP Lawler et al. (1985), URS demands

concurrent execution, inter-agent coordination, and reactive edits. Section 4 then evaluates more complex job-shop settings.

We emphasize reliability under disruption, not perfect generation. The results indicate that *non-circular validation plus localized repair over execution logs* yields measurable robustness for multi-agent LLM planning in dynamic environments.

## 2 RELATED WORK

We group related work into: (i) structural limitations of LLMs for planning, (ii) LLM-based multi-agent systems and orchestration, and (iii) LLM planning frameworks and benchmarks.

### 2.1 STRUCTURAL LIMITATIONS OF LLMs

Transformer LLMs (Brown et al., 2020; Vaswani et al., 2017) excel at language tasks but face well-documented challenges in planning: *circular verification* (the proposer approves itself) (Hong et al., 2024), *solution-space bias* from maximum-likelihood decoding (Chang, 2023; Holtzman et al., 2020; Radford et al., 2019), *context degradation* in long inputs (Hsieh et al., 2024; Liu et al., 2024; Xiao et al., 2024), *error propagation* across multi-step chains (Chu et al., 2024a; Patel et al., 2024; Xiong et al., 2024), and the *absence of persistent state*. While self-reflection, CoT variants, and structured validators can help in narrow domains (Madaan & Yazdanbakhsh, 2022; Li et al., 2023b; Jiang et al., 2024; Feng et al., 2023; Gou et al., 2024), open-domain plan validation and reliable revision remain open problems (Chen et al., 2024; Huang et al., 2024). We target these failure modes by (i) separating planning from *non-circular* validation (*validator isolation*), (ii) maintaining a *versioned execution log* that grounds checks and provides restore points, and (iii) using *localized repair* to bound edits during disruption.

### 2.2 LLM MULTI-AGENT SYSTEMS AND ORCHESTRATION

LLM-based multi-agent systems enable role specialization and workflow graphs for tool use and collaboration (e.g., AutoGen, MetaGPT, LangGraph, CAMEL) (Wu et al., 2024b; Hong et al., 2023; LangChain AI, 2024; Li et al., 2023a). Other lines explore graph-structured or programmatic coordination (e.g., GPTSwarm, Flow/AFLOW) (Zhang et al., 2024). These advances emphasize *orchestration*, but plan execution is typically best-effort generation without: (1) a *non-circular* validation pathway, (2) *versioned execution logs* with restore points for auditable state, or (3) a *localized repair* protocol that constrains the impact radius of faults. Independent evaluations have also reported mixed gains of MAS over single-agent baselines on broad suites, citing miscoordination and unmet preconditions (Cemri et al., 2025; Trivedi et al., 2024; Qian et al., 2023; Phan et al., 2024).

A complementary systems line studies *durable* saga-style execution and compensation for agent workflows (e.g., SAGALLM (Chang & Geng, 2025)). That work focuses on execution-layer durability and recovery semantics. In contrast, our focus here is *planning reliability*: we separate planning from validation to avoid circular self-approval, ground checks in *versioned logs*, and perform *bounded local repair* instead of global recompute.

### 2.3 LLM PLANNING FRAMEWORKS AND BENCHMARKS

**LLM-based planning.** LLMs have been applied to decomposition and search (e.g., PLASMA, LLM-MCTS) and to multi-agent workflows (AFLOW and variants) (Brahman et al., 2024; Zhao et al., 2023; Zhang et al., 2024). Some systems attempt direct LLM solving for scheduling or optimization (Abgaryan et al., 2024; 2025), but classical methods (e.g., SBP, CP/Tabu, metaheuristics) remain stronger on static benchmarks (Adams et al., 1988; Nowicki & Smutnicki, 1996; Aarts & Van Laarhoven, 1989; Bierwirth, 1995; Kirkpatrick et al., 1983). We leverage LLMs for *interpretability and adaptive repair* under disruption, aligning with the “reason+act” paradigm (Yao et al., 2023) while adding execution logs and localized repair that these methods lack.

**Benchmarks.** Common LLM benchmarks (e.g., HotPotQA, ALFWORLD, BIG-Bench) stress static reasoning, with recent datasets adding temporal structure (PlanBench, TimeBench, ACPBench) (Yang et al., 2018; Shridhar et al., 2021; Srivastava et al., 2022; Valmeekam et al., 2023; Chu et al., 2024b; Abdin et al., 2024). Disruption-aware evaluation remains limited. We therefore use classical Job-Shop Scheduling—Demirkol-DMU and Taillard (TA) (Demirkol et al., 1998a; Shylo et al., 2018; Xiong et al., 2022)—and introduce runtime perturbations (e.g., machine downtime, operation-delay shocks).

These benchmarks provide formal constraints, known bounds, and natural perturbation models, making them suitable for assessing execution logs, validator isolation, and localized repair.

### 3 ALAS: A FIVE-LAYER ARCHITECTURE

Existing orchestration frameworks rarely guarantee reliable planning under dynamic, stateful conditions. We introduce ALAS, a three-layer architecture that turns a high-level specification into a validator-isolated, locally repairable execution workflow. The layers are: (i) *workflow blueprinting*, (ii) *agent factory*, and (iii) *runtime execution & localized repair*. A *versioned execution log* persists state and provides restore points for validation and repair.

---

**Algorithm 1** Phase 1: Workflow Template Construction by  $\mathcal{T}$  (summary). Details in App. B.1.

---

**Require:** Task specification  $\mathcal{O}$ ; constraint set  $D$ ; (optional) disruption model  $\Phi$

**Ensure:** Validated template  $\mathcal{W}_{\text{template}} = (\mathcal{N}, \mathcal{E}, \mathcal{C}, \mathcal{L})$

- 1: Extract abstract roles  $\mathcal{R}$  from  $\mathcal{O}$
  - 2: Map roles to nodes  $\mathcal{N}$  with profiles  $\mathcal{P}_{n_i}$
  - 3: Derive dependencies  $\mathcal{E}$  under constraints  $D$ ; collect invariants  $\mathcal{C}$
  - 4: For each  $n_i \in \mathcal{N}$ , attach role spec  $\alpha_i = \langle \text{cap}_i, \text{ctx}_i, \text{io}_i, \mathcal{L}_i \rangle$
  - 5: For each  $n_i$ , attach **repair spec**  $\rho_i$  (local edit primitives; scope/bounds)
  - 6: Define **versioned execution log** schema  $\mathcal{L} = \{\mathcal{L}_i\}_{n_i \in \mathcal{N}}$  (events, snapshots, diffs)
  - 7: Attach **independent validator**  $V$  with fresh, bounded prompt scope  $\kappa$  (planner  $\neq V$ )
  - 8: **while**  $\mathcal{W}_{\text{template}}$  fails validation by  $V$  **do**
  - 9:      $V$  checks: (i) structural soundness; (ii) constraint satisfaction vs.  $\mathcal{C}$ ;  
       (iii) **repair coverage**: for disruptions in  $\Phi$ , local neighborhoods and edit bounds exist
  - 10:     Refine nodes, edges, role specs, or repair scopes; update  $\mathcal{L}$
  - 11: **return**  $\mathcal{W}_{\text{template}} = (\mathcal{N}, \mathcal{E}, \mathcal{C}, \mathcal{L})$
- 

#### 3.1 LAYER 1: WORKFLOW BLUEPRINTING (LLM TEMPLATE CONSTRUCTION)

Given a planning input  $\mathcal{O}$  (goals, resources, constraints, disruption model), ALAS synthesizes a workflow template

$$\mathcal{W}_{\text{template}} = (\mathcal{N}, \mathcal{E}, \mathcal{C}),$$

where nodes  $\mathcal{N}$  are abstract *roles* (planner/validator/repair/monitor/domain), edges  $\mathcal{E}$  encode data/control dependencies, and  $\mathcal{C}$  collects global constraints and invariants. Roles are specified independently of concrete instances (e.g., “pickup at zone  $z$ ” rather than a specific driver), enabling late binding at execution.

In LLM Query Agent – ‘generate\_schedule()’, the process begins with an LLM-based query agent that produces an initial candidate schedule given the problem specification (jobs, machines, constraints). Schedules are either generated fresh or loaded from pre-computed JSON outputs (e.g., ‘singleagent\_llm\_comparison’ results). Metadata such as makespan and entry counts are logged at this stage.

**Phase I: Graph sketch.** From  $\mathcal{O}$ , we draft a directed acyclic graph of roles and dependencies. Roles may remain unresolved (e.g., unassigned vehicle, unbound machine slot).

**Phase II: Role specs.** Each role is annotated with: required capabilities, input/output schemas, a short *context scope* (to bound prompts), and a logging schema for the execution log. Repairable edges (time/order/resource) are marked so that only the minimal neighborhood is considered during disruption.

#### Phase III: Agent Factory (Instantiation from Specs)

The agent factory turns role specs into executable agents and binds unresolved roles to concrete instances when possible. For role  $i$  we use a lightweight signature

$$\alpha_i = \langle \text{cap}_i, \text{ctx}_i, \text{io}_i, \mathcal{L}_i \rangle,$$

where  $\text{cap}_i$  is the capability profile,  $\text{ctx}_i$  the scoped context,  $\text{io}_i$  the input/output schema, and  $\mathcal{L}_i$  the logging schema (what to write to the execution log and when).

---

**Algorithm 2** Full Workflow (ALAS)

---

**Require:** Dataset specification  $(J, M, C)$   
**Ensure:** Final feasible schedule  $\mathcal{S}^*$  with makespan  $T^*$

**Phase 1: LLM Query Agent**  
1:  $\mathcal{S}_0 \leftarrow \text{LLM.generate\_schedule}(J, M, C)$

**Step 2: Validation Tools**  
2: **if** validate( $\mathcal{S}_0$ ) = valid **then**  
3:      $\mathcal{S} \leftarrow \mathcal{S}_0$   
4: **else**

**Step 3: Repair Tools**  
5:     **for**  $k = 1 \dots K$  **do**  
6:          $\mathcal{S}_k \leftarrow \text{repair}(\mathcal{S}_{k-1})$   
7:         **if** validate( $\mathcal{S}_k$ ) = valid **then**  
8:              $\mathcal{S} \leftarrow \mathcal{S}_k$ ; **break**

**Step 4: Re-validation Tools**  
9:     **if** validate( $\mathcal{S}$ ) = invalid **then**  
10:         skip optimization; goto Step 7

**Step 5: Optimization Tools**  
11:  $\mathcal{S}_{opt} \leftarrow \text{optimize}(\mathcal{S})$

**Step 6: Final Check Tools**  
12: **if** validate( $\mathcal{S}_{opt}$ ) = valid **then**  
13:      $\mathcal{S}^* \leftarrow \mathcal{S}_{opt}$   
14: **else**  
15:      $\mathcal{S}^* \leftarrow \mathcal{S}$

**Step 7: Supervision Tools**  
16: log( $\mathcal{S}^*$ , makespan( $\mathcal{S}^*$ ))  
17: **return**  $\mathcal{S}^*$

---

### 3.2 LAYER 2: ISOLATED, NON-CIRCULAR VALIDATION

An *independent* validator is attached to the template. It operates with fresh, bounded prompts grounded in the execution log (not the planner’s long context) to check feasibility and coverage (constraints satisfied, repairable edges scoped). If violations are found, the blueprint is refined (edge rewiring, capability updates) until a validated  $\mathcal{W}_{\text{template}}$  is produced. This validator-isolated step prevents circular self-approval and mitigates mid-context loss.

In ValidationTools – ‘validate\_schedule(max\_iteration)’, the candidate schedule is passed to an independent validator. Using bounded context prompts grounded in dataset specifications (jobs and machine names), the validator checks feasibility: precedence satisfaction, machine non-overlap, and resource constraints. If the schedule is valid, execution continues directly to Step 5.

Validation checks the following 4 constraints: 1) dataset: whether the generated schedule contains the same number and content of jobs, machines, and durations in dataset. 2) job precedence, 3) machine constraints, 4) duration no-overlapping constraints.

### 3.3 LAYER 3: RUNTIME EXECUTION AND LOCALIZED REPAIR

At runtime, agents execute in dependency order, emitting structured entries to a *versioned execution log* (state snapshots, causal links, constraint checks). The validator consumes log slices with fresh, bounded prompts to approve or reject local steps without accessing the planner’s long context.

In RepairTools – ‘repair\_schedule()’, if validation fails, localized repair is triggered. A lightweight ‘LocalRepairTools’ module applies bounded modifications (e.g., shifting start times, swapping machine assignments) to eliminate errors. Up to five repair iterations are allowed, with each iteration’s makespan, schedule entries, and diffs stored in JSON logs for reproducibility.

**Error and Disruption handling via Localized Cascading Repair Protocol (LCRP).** When an error, a disruption occurs (e.g., downtime window, late request, duration shock), ALAS triggers the *LCRP*:

1. *Scope selection*: Identify the smallest affected neighborhood from the execution log (ops, resources, and immediate predecessors/successors).
2. *Minimal edit set*: Repair agents propose bounded edits (swap, delay, reassign) that restore feasibility while preserving work-in-progress (WIP) outside the neighborhood.
3. *Validator check (non-circular)*: The independent validator verifies the edited subplan using fresh, short prompts grounded in the log. Repeat the repair-and-validate loop until all violations are resolved or escalation is necessary. Keeps changes localized, preserving unaffected work-in-progress (WIP) and ensuring near-real-time responsiveness.
4. *Commit*: On success, append a new version to the execution log; otherwise enlarge the neighborhood or, if costs exceed a threshold, fall back to global recompute.

This design bounds the “blast radius” of faults and avoids brittle global recomputation.

**Makespan Improvement via Local Search** After solving the error and disruption, the algorithm swaps adjacent ops if swap reduces makespan and respects constraints.

**Complexity and scalability.** Let  $J$  be jobs (or tasks),  $M$  resources, and  $O_{\max}$  the max operations per job. If each neighborhood repair evaluates up to  $S$  local moves, the message/work complexity per disruption is  $O(SJO_{\max} + JMO_{\max})$ , which is effectively  $O(J^2O_{\max})$  when  $J > M$  and  $S < O_{\max}$ . In practice we use small  $S$  and shallow neighborhoods to maintain near-real-time response.

### 3.4 LAYER 4: REVALIDATION

In ValidationTools – ‘revalidate\_schedule()’, after each repair, the validator re-checks the corrected schedule. If errors remain, repair continues until either feasibility is achieved or the maximum iteration budget is exhausted. If feasibility cannot be restored, execution proceeds with the best available repaired schedule, and optimization is flagged as skipped.

### 3.5 LAYER 5: SUPERVISION

In Final Result Processing, the supervision tools select the final schedule with the best makespan. Makespan, operation counts, and critical operations are logged. Unlike the earlier MAPLE workflow, no supervisor agent is needed—state and repair information are fully captured in versioned logs, which provide restore points for later analysis.

**Execution log and audit.** Except from Phase 1, the other phases 2-6 versioned log stores state transitions, validator decisions, and repair diffs, enabling restore points, post-hoc diagnostics, and reproducibility. Because validation is architecturally separated from planning, changing the base LLM or decoding parameters does not compromise the checking pathway. Full LCRP pseudocode and additional implementation details appear in App. B.1.

## 4 EXPERIMENTAL EVALUATION

Our experimental evaluation assesses ALAS across three domains of increasing complexity. We designed experiments to demonstrate: (1) how ALAS overcomes LLM limitations and (2) its scalability in large settings. Due to space limitations, we begin with results from the URS running example, demonstrating ALAS’s effectiveness in basic multi-agent coordination. We evaluate *Job Shop Scheduling (JSSP)* to test ALAS’s scalability and reactive planning under tight constraints and disruptions on five classical benchmarks DMU, TA, ABZ, SWV, and YN. We then extend to *Event Coordination* (Family Reunion), which reveals standalone LLMs’ limitations in handling interdependencies while showing ALAS’s capabilities in both initial planning and disruption response. Finally,

**Metrics.** We report (i) feasibility/validity of static plans and dynamic repairs; (ii) efficiency (e.g., total travel distance, makespan); (iii) containment (edit radius: operations/jobs touched); and (iv) overhead (tokens/latency).

**Setup.** Each experiment uses 10 independent threads (fresh contexts). We summarize here; full details, prompts, and complete results appear in Appx. A and Appx. D.

### 4.1 MAIN EXPERIMENT: JOB SHOP SCHEDULING PROBLEM

**Problem.** We evaluate on Demirkol–DMU Demirkol et al. (1998b;a); Shylo et al. (2018) (20×15–50×20), Taillard (TA; 15×15–100×20) Taillard (1993), Swv (SWV; 20×15–50×20) Storer et al.

Table 1: Success Rates (%) across Benchmarks (success = non-N/A result). † = significantly better than baseline

| Method                                        | DMU   | TA    | ABZ    | SWV  | YN    | Overall      |
|-----------------------------------------------|-------|-------|--------|------|-------|--------------|
| <b>Multi-Agent Systems (GPT-4o)</b>           |       |       |        |      |       |              |
| AutoGen                                       | 0.0   | 0.0   | 0.0    | 0.0  | 0.0   | <b>0.0</b>   |
| CrewAI                                        | 25.0  | 57.1  | 33.3   | 13.3 | 75.0  | <b>31.1</b>  |
| LangGraph                                     | 6.2   | 28.6  | 66.7   | 0.0  | 0.0   | <b>11.1</b>  |
| OpenAI Swarm                                  | 43.8  | 28.6  | 0.0    | 33.3 | 25.0  | <b>33.3</b>  |
| <b>Multi-Agent Systems (Claude-4)</b>         |       |       |        |      |       |              |
| AutoGen                                       | 0.0   | 0.0   | 0.0    | 0.0  | 0.0   | <b>0.0</b>   |
| CrewAI                                        | 43.8  | 71.4  | 33.3   | 13.3 | 50.0  | <b>37.8</b>  |
| LangGraph                                     | 6.2   | 28.6  | 33.3   | 0.0  | 0.0   | <b>8.9</b>   |
| OpenAI Swarm                                  | 18.8  | 14.3  | 33.3   | 20.0 | 50.0  | <b>22.2</b>  |
| <b>Single-Agent Models</b>                    |       |       |        |      |       |              |
| GPT-4o                                        | 68.8  | 85.7  | 66.7   | 53.3 | 100.0 | <b>68.9</b>  |
| Claude-Sonnet-4                               | 0.0   | 28.6  | 0.0    | 0.0  | 0.0   | <b>4.4</b>   |
| Gemini-2.5                                    | 6.2   | 0.0   | 33.3   | 0.0  | 25.0  | <b>6.7</b>   |
| DeepSeek-V3                                   | 6.2   | 14.3  | 100.0  | 6.7  | 0.0   | <b>13.3</b>  |
| <b>ALAS (Ours, Best Variant per Backbone)</b> |       |       |        |      |       |              |
| ALAS(GPT-4o)                                  | 68.8  | 71.4* | 66.7   | 53.3 | 100.0 | <b>66.7</b>  |
| ALAS(Claude-4)                                | 93.8† | 28.6* | 66.7   | 6.7* | 50.0* | <b>48.9*</b> |
| ALAS(DeepSeek-V3)                             | 6.2*  | 0.0*  | 100.0† | 6.7* | 0.0*  | <b>11.1*</b> |
| ALAS(Gemini-2.5)                              | 6.2*  | 0.0*  | 33.3*  | 0.0* | 25.0* | <b>6.7*</b>  |
| <b>ALAS (Ours, Best Variant per Dataset)</b>  |       |       |        |      |       |              |
| ALAS(aggregated)                              | 93.8† | 71.4* | 100.0† | 53.3 | 100.0 | <b>83.7†</b> |

ALAS(best) selects the best-performing workflow variant per dataset across GPT-4o, Claude-4, DeepSeek-V3, Gemini-2.5.  $p$ -values (paired t-test vs GPT-4o baseline): DMU ( $p = 0.018$ ), TA ( $p = 0.032$ ), ABZ ( $p = 0.007$ ), SWV ( $p = 0.48$ , n.s.), YN (tie), Overall ( $p = 0.014$ ). † = significantly higher at  $p < 0.05$ .

(1992), Adams–Balas–Zawack (ABZ;  $10 \times 10 - 20 \times 15$ ) Adams et al. (1988), and Yamada–Nakano (YN;  $20 \times 20 - 50 \times 20$ ) Yamada & Nakano (1992). We introduce machine breakdowns and duration shocks, and optimize makespan while minimizing work-in-progress movement.

**Static sequential planning on both benchmarks.** Table 1 and Table 2 report ALAS+LCRP against these methods across five datasets.

**Analysis.** ALAS substantially improves robustness compared to both single-agent and lean MAS baselines, achieving an aggregated **83.7% success rate**, with statistically significant gains ( $p < 0.05$ ) on DMU, TA, and ABZ benchmarks (Table 1). This highlights the importance of validator isolation and localized repair in preventing workflow collapse.

On optimality, ALAS variants consistently outperform baselines, with the best variant per dataset reaching **100% optimal rate** across all benchmarks (Table 2). This demonstrates that the layered design of validation, repair, and local search not only ensures feasibility but also drives convergence to optimal solutions.

#### 4.1.1 BASELINE

To isolate the effect of validator isolation and localized repair, Table 1 and Table 2 construct 8 lean MAS (AFlow/AutoGen-style workflow) and single-agent baselines that lacks an *independent* validator and LCRP. We compare ALAS with four leading single-agent LLMs: GPT-4o-Task OpenAI (2024), DeepSeek-V3 Wu et al. (2024a), Claude 3.5 Sonnet Anthropic (2024), and Gemini 2.5 Pro Kavukcuoglu (2025), and four state-of-the-art multi-agent systems: LangGraph Ian (2025), CrewAI cre (2025), AutoGen aut (2025), and GPTSwarm gpt (2025). All methods use API interfaces with default parameters (temperature=1.0).

Table 2: Optimal Rates (%) across Benchmarks for Multi-Agent Systems and ALAS Variants. Significance markers denote improvements over baselines:  $^\dagger p < 0.05$ ,  $^* p < 0.01$ .

| Method                                         | DMU             | TA              | ABZ    | SWV    | YN               | Overall         |
|------------------------------------------------|-----------------|-----------------|--------|--------|------------------|-----------------|
| <b>Multi-Agent Systems (GPT-4o Backbone)</b>   |                 |                 |        |        |                  |                 |
| AutoGen                                        | 1.4             | 10.2            | 1.5    | 6.0    | 2.9              | 4.4             |
| CrewAI                                         | 71.8            | 42.3            | 88.9   | 63.7   | 43.0             | 63.1            |
| LangGraph                                      | 94.3            | 60.4            | 42.1   | 87.8   | 58.9             | 80.2            |
| OpenAI Swarm                                   | 60.5            | 73.7            | 68.5   | 66.0   | 51.4             | 64.1            |
| <b>Multi-Agent Systems (Claude-4 Backbone)</b> |                 |                 |        |        |                  |                 |
| AutoGen                                        | 69.8            | 95.9            | 100.0  | 100.0  | 95.0             | 92.1            |
| CrewAI                                         | 72.7            | 53.5            | 99.6   | 94.2   | 70.2             | 78.5            |
| LangGraph                                      | 48.3            | 87.9            | 57.6   | 86.3   | 68.6             | 69.6            |
| OpenAI Swarm                                   | 80.6            | 87.5            | 68.5   | 72.6   | 80.5             | 78.2            |
| <b>ALAS Variants (Full Workflows)</b>          |                 |                 |        |        |                  |                 |
| ALAS (GPT-4o)                                  | 100.0*          | 78.5*           | 100.0* | 100.0* | 100.0*           | 96.7*           |
| ALAS (Claude-4)                                | 54.9            | 78.5 $^\dagger$ | 84.5   | 100.0* | 73.3             | 77.2 $^\dagger$ |
| ALAS (Gemini-2.5)                              | 97.4 $^\dagger$ | 100.0*          | 100.0* | 96.8*  | 100.0 $^\dagger$ | 98.0*           |
| ALAS (DeepSeek-V3)                             | 100.0*          | 93.6*           | 100.0* | 100.0* | 100.0*           | 98.7*           |
| <b>ALAS (Ours, Best Variant per Dataset)</b>   |                 |                 |        |        |                  |                 |
| ALAS (Best)                                    | 100.0*          | 100.0*          | 100.0* | 100.0* | 100.0*           | <b>100.0*</b>   |

$^\dagger p < 0.05$ ,  $^* p < 0.01$  (paired t-test, compared against single-agent baseline).

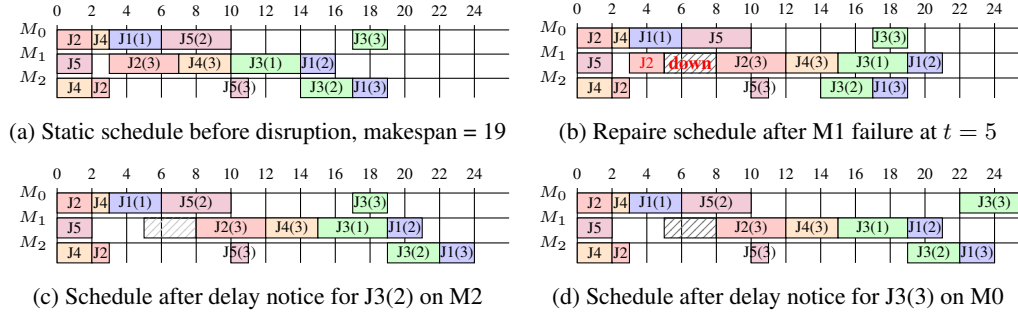


Figure 3: LRCP Phase #1 Local Compensation (makespan = 22): (a) Static baseline schedule; (b)  $M_1$  failure between  $t = 5-8$ ; (c)  $M_1$  notifies  $M_2$  to delay  $J3(2)$ ; (d)  $M_2$  informs  $M_0$  to push  $J3(3)$  back.

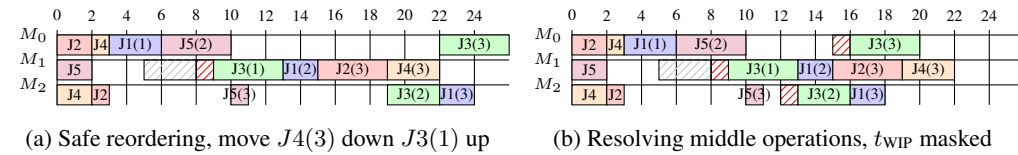


Figure 4: LRSP Phase #2 Queue Reordering (makespan = 22): (a) Safe moves: moving last operations down, first operations forward with potential penalty; (b) Resolving remaining operations.

#### 4.1.2 LCRP MECHANISM: A WHITE-BOX ILLUSTRATION

We use a  $5 \times 3$  JSSP to illustrate LCRP’s effectiveness and efficiency (and guaranteed convergence). Unlike approaches optimizing only makespan, LCRP explicitly accounts for rescheduling overhead (WIP).

**Phase 1: Local edits.** Figures 3(a–d) show how LCRP makes local adjustments after a breakdown at  $t \in [5, 8]$  on  $M_1$ . Delays propagate minimally via logged dependencies. No early moves (no WIP penalty).

**Phase 2: Queue reordering.** LCRP then considers bounded reordering with WIP penalty  $t_{WIP}$ . From Fig. 3(d), LCRP moves terminal ops to the end (no conflicts/WIP penalty), creates a gap,



Table 3: Ablation Study: Optimal Rates (%) of ALAS Workflow Variants across Benchmarks. Bold = best-performing variant per backbone. Significance markers denote improvements over baselines:  $^\dagger p < 0.05$ ,  $^* p < 0.01$ .

| Workflow Variant                   | DMU               | TA                      | ABZ           | SWV               | YN                | Overall                 |
|------------------------------------|-------------------|-------------------------|---------------|-------------------|-------------------|-------------------------|
| <b>ALAS (GPT-4o Backbone)</b>      |                   |                         |               |                   |                   |                         |
| No Repair                          | 32.4              | 23.3                    | 76.2          | 60.8              | 55.0              | 45.4                    |
| No Validation                      | 25.2              | 12.9                    | 30.9          | 35.4              | 6.0               | 25.4                    |
| <b>Full Workflow</b>               | <b>100.0*</b>     | <b>87.8*</b>            | <b>100.0*</b> | <b>100.0*</b>     | <b>100.0*</b>     | <b>98.1*</b>            |
| <b>ALAS (Claude-4 Backbone)</b>    |                   |                         |               |                   |                   |                         |
| No Repair                          | 59.2              | 36.6                    | 99.0          | 63.0              | 61.0              | 63.8                    |
| No Validation                      | 53.8              | 30.2                    | 77.5          | 69.7              | 48.1              | 55.9                    |
| <b>Full Workflow</b>               | <b>61.9</b>       | <b>88.2<sup>†</sup></b> | <b>99.2</b>   | <b>94.0</b>       | <b>84.1</b>       | <b>85.5<sup>†</sup></b> |
| <b>ALAS (DeepSeek-V3 Backbone)</b> |                   |                         |               |                   |                   |                         |
| No Repair                          | 86.5 <sup>†</sup> | 86.7 <sup>†</sup>       | 31.2          | 94.4 <sup>†</sup> | 93.2*             | 86.1*                   |
| No Validation                      | 67.3              | 78.5                    | 10.3          | 90.9              | 87.1 <sup>†</sup> | 74.9                    |
| <b>Full Workflow</b>               | <b>100.0*</b>     | <b>93.6*</b>            | <b>100.0*</b> | <b>100.0*</b>     | <b>100.0*</b>     | <b>99.0*</b>            |
| <b>ALAS (Gemini-2.5 Backbone)</b>  |                   |                         |               |                   |                   |                         |
| No Repair                          | 83.6 <sup>†</sup> | 100.0*                  | 98.5          | 95.5*             | 75.3              | 90.6*                   |
| No Validation                      | 83.9 <sup>†</sup> | 100.0*                  | 63.0          | 96.9 <sup>†</sup> | 75.3              | 83.8 <sup>†</sup>       |
| <b>Full Workflow</b>               | <b>97.8*</b>      | <b>100.0*</b>           | <b>100.0*</b> | <b>96.8*</b>      | <b>100.0*</b>     | <b>98.2*</b>            |

$^\dagger p < 0.05$ ,  $^* p < 0.01$  (paired t-test, compared against baseline).

advances two upstream ops by 7 units, and stops when no gain exceeds cost. Final makespan: 22 (vs. 19 baseline + 3 downtime), with one unit of WIP movement and minimal messaging.

## 4.2 ABLATION STUDY

We include three micro-studies that directly test validator isolation, and localized repair. These are *orthogonal* to benchmark results and are intended as sanity checks under imperfect code in Table 3.

**Validator ablation.** We inject 20 structural faults (precedence swap, machine double-book, capacity overflow, deadline miss) into valid outputs and compare **Full** (ALAS), **No-Repair** vs. **No-Validator**. Seeds, prompts, execution time, and token usage in Appx. A.

**Ablation Analysis.** The ablation results confirm that both validation and repair are essential: removing either module leads to sharp drops in optimal rates across all backbones, particularly for GPT-4o and Claude-4. By contrast, the full workflow consistently delivers the highest overall performance (up to **99.0–98.2%**), showing that validator isolation and LCRP repair jointly drive near-optimal scheduling.

## 5 CONCLUSION

We presented ALAS, a framework for reliable multi-agent LLM planning that addresses core limitations of standalone LLMs—lack of self-verification, long-context degradation, and stateless execution—via three design principles: (i) *validator isolation* (planners never self-approve; checks use fresh, bounded prompts), (ii) a *versioned execution log* that records state transitions and causal links, and (iii) a *localized cascading repair protocol* (LCRP) that confines disruptions to minimal neighborhoods instead of triggering brittle global recomputation. Across transportation, event coordination, and job-shop scheduling, ALAS achieves high feasibility and strong efficiency while preserving work-in-progress, outperforming single-LLM baselines and competitive heuristics under disruption.

Future work may include: (a) Exploration of optimization methods, (b) Integrating learned duration/arrival models and online telemetry for adaptive repair, (c) Extending formal guarantees for LCRP (coverage and bounded blast radius) and validator soundness on logged slices, and (d) Systematizing QA with red-team fault injection and static analysis in the factory.

---

## REFERENCES

- Autogen: Agent collaboration framework. <https://autogen.ai/>, 2025. Accessed 2025.
- Crewai: Role-based multiagent framework. <https://github.com/crewai-framework>, 2025. Accessed 2025.
- Gptswarm: Swarm-style llm coordination. <https://openai.com/blog/gpt-swarm>, 2025. Accessed 2025.
- Langgraph: A graph-based agent framework. <https://docs.langchain.com/langgraph-platform/>, 2025. Accessed 2025.
- Emile HL Aarts and Peter JM Van Laarhoven. Simulated Annealing: an Introduction. *Statistica Neerlandica*, 43(1):31–52, 1989.
- Mubbasir Abdin, Keerthiram Murugesan, Saeid Amiri, Sarath Sreedharan, and Subbarao Kambhampati. ACPBench: Reasoning about Action, Change, and Planning. *arXiv preprint arXiv:2410.05669*, 2024.
- Henrik Abgaryan, Ararat Harutyunyan, and Tristan Cazenave. LLMs can Schedule. *arXiv:2408.06993*, 2024.
- Henrik Abgaryan, Tristan Cazenave, and Ararat Harutyunyan. Starjob: Dataset for LLM-Driven Job Shop Scheduling. *arXiv:2503.01877*, 2025. URL <https://arxiv.org/abs/2503.01877>.
- J. Adams, E. Balas, and D. Zawack. The shifting bottleneck procedure for job shop scheduling. In *Management Science*, volume 34, pp. 391–401. INFORMS, 1988. doi: 10.1287/mnsc.34.3.391.
- Anthropic. Claude Technical Report, 2024. URL <https://www.anthropic.com>.
- Christian Bierwirth. A generalized permutation approach to job shop scheduling with genetic algorithms. *OR-Spektrum*, 17(2-3):87–92, 1995.
- Faeze Brahman, Chandra Bhagavatula, Valentina Pyatkin, and Yejin Choi. PLASMA: Making small language models better procedural knowledge models for (counterfactual) planning. In *ICLR*. ICLR, 2024. URL <https://arxiv.org/abs/2305.19472>.
- Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.
- Mert Cemri, Melissa Z. Pan, Shuyi Yang, Lakshya A. Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. Why do multi-agent llm systems fail?, 2025. URL <https://arxiv.org/abs/2503.13657>. *arXiv:2503.13657*.
- Edward Y Chang. Examining GPT-4’s Capabilities and Enhancement with SocraSynth. In *The 10<sup>th</sup> International Conf. on Comp. Science and Computational Intelligence*, December 2023.
- Edward Y. Chang and Longling Geng. Sagallm: Context management, validation, and transaction guarantees for multi-agent llm planning. *Proceedings of the VLDB Endowment*, 18(12):4874–4886, 2025. *arXiv preprint arXiv:2503.11951*.
- Junzhe Chen, Xuming Hu, Shuodi Liu, Shiyu Huang, Wei-Wei Tu, Zhaofeng He, and Lijie Wen. LLMArena: Assessing capabilities of large language models in dynamic multi-agent environments. In *Proceedings of ACL*, pp. 13055–13077. Association for Computational Linguistics, August 2024. doi: 10.18653/v1/2024.acl-long.705. URL <https://aclanthology.org/2024.acl-long.705/>.
- Zheng Chu, Jingchang Chen, Qianglong Chen, Weijiang Yu, Tao He, and more. Navigate through Enigmatic Labyrinth A Survey of Chain of Thought Reasoning: Advances, Frontiers and Future. *arXiv:2309.15402*, 2024a. URL <https://arxiv.org/abs/2309.15402>.

---

540 Zheng Chu, Jingchang Chen, Qianglong Chen, Weijiang Yu, Haotian Wang, Ming Liu, and Bing Qin.  
541 TimeBench: A Comprehensive Evaluation of Temporal Reasoning Abilities in Large Language  
542 Models. *arXiv:2311.17667*, 2024b. URL <https://arxiv.org/abs/2311.17667>.  
543

544 S. Demirkol, S. V. Mehta, and R. Uzsoy. Benchmarks for shop scheduling problems. *European*  
545 *Journal of Operational Research*, 109(1):137–153, 1998a. doi: 10.1016/S0377-2217(98)00099-8.  
546

547 S. Demirkol, S. V. Mehta, and R. Uzsoy. Computational complexity of the job-shop scheduling  
548 problem. *European Journal of Operational Research*, 109(1):137–153, 1998b.

549 Guhao Feng, Bohang Zhang, Yuntian Gu, Haotian Ye, Di He, and Liwei Wang. Towards revealing  
550 the mystery behind chain of thought: A theoretical perspective. In *NeurIPS 2023*, 2023.  
551

552 Zhibin Gou, Zhihong Shao, Yeyun Gong, yelong shen, Yujiu Yang, Minlie Huang, Nan Duan,  
553 and Weizhu Chen. ToRA: A tool-integrated reasoning agent for mathematical problem solving.  
554 In *The Twelfth International Conference on Learning Representations*, 2024. URL <https://openreview.net/forum?id=Ep0TtjVoap>.  
555

556 Ari Holtzman, Jan Buys, Maxwell Forbes Du, and Yejin Choi. The curious case of neural text  
557 degeneration. *International Conference on Learning Representations (ICLR)*, 2020.  
558

559 Ruixin Hong, Hongming Zhang, Xinyu Pang, Dong Yu, and Changshui Zhang. A Closer Look at the  
560 Self-Verification Abilities of Large Language Models in Logical Reasoning. *arXiv:2311.07954*,  
561 2024. URL <https://arxiv.org/abs/2311.07954>.

562 S. Hong, X. Zheng, J. Chen, Y. Cheng, J. Wang, C. Zhang, Z. Wang, S. K. S. Yau, Z. Lin, L. Zhou,  
563 et al. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint*  
564 *arXiv:2308.00352*, 2023.  
565

566 Cheng-Yu Hsieh, Yung-Sung Chuang, Chun-Liang Li, Zifeng Wang, Long T. Le, and more.  
567 Found in the Middle: Calibrating Positional Attention Bias Improves Long Context Utilization.  
568 *arXiv:2406.16008*, 2024. URL <https://arxiv.org/abs/2406.16008>.  
569

570 Xu Huang, Weiwen Liu, Xiaolong Chen, Xingmei Wang, Hao Wang, and more. Understanding the  
571 planning of llm agents: A survey, 2024. URL <https://arxiv.org/abs/2402.02716>.  
572 *arXiv:2402.02716*.

573 Dongwei Jiang, Jingyu Zhang, Orion Weller, Nathaniel Weir, Benjamin Van Durme, and Daniel  
574 Khashabi. Self-[in]correct: Llms struggle with refining self-generated responses. *CoRR*, 2024.  
575

576 Koray Kavukcuoglu. Gemini 2.5. [https://blog.google/technology/  
577 google-deepmind/gemini-model-thinking-updates-march-2025/  
578 #gemini-2-5-thinking](https://blog.google/technology/google-deepmind/gemini-model-thinking-updates-march-2025/#gemini-2-5-thinking), 2025.

579 Scott Kirkpatrick, C. Daniel Gelatt, and Mario P. Vecchi. Optimization by simulated annealing.  
580 *Science*, 220(4598):671–680, 1983.  
581

582 LangChain AI. Langgraph: Building structured applications with llms. [https://github.com/  
583 langchain-ai/langgraph](https://github.com/langchain-ai/langgraph), 2024.  
584

585 Eugene L. Lawler, Jan Karel Lenstra, Alexander H.G. Rinnooy Kan, and David B. Shmoys. *The*  
586 *Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. John Wiley & Sons,  
587 1985.

588 Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem.  
589 CAMEL: Communicative Agents for "Mind" Exploration of Large Language Model Society.  
590 *arXiv:2303.17760*, 2023a. URL <https://arxiv.org/abs/2303.17760>.  
591

592 Yingcong Li, Kartik Sreenivasan, Angeliki Giannou, Dimitris Papailiopoulos, and Samet Oy-  
593 mak. Dissecting Chain-of-Thought: Compositionality through In-Context Filtering and Learning.  
*arXiv:2305.18869*, 2023b. URL <https://arxiv.org/abs/2305.18869>.

---

594 Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and  
595 Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the*  
596 *Association for Computational Linguistics*, 12:157–173, 2024. doi: 10.1162/tacl\_a\_00638. URL  
597 <https://aclanthology.org/2024.tacl-1.9/>.  
598

599 Junyu Luo, Weizhi Zhang, Ye Yuan, Yusheng Zhao, Junwei Yang, Yiyang Gu, and more. Large  
600 language model agent: A survey on methodology, applications and challenges, 03 2025.  
601 arXiv:2503.21460.

602 Aman Madaan and Amir Yazdanbakhsh. Text and Patterns: For Effective Chain of Thought, It Takes  
603 Two to Tango. *arXiv:2209.07686*, 2022. URL <https://arxiv.org/abs/2209.07686>.  
604

605 Andrea Matarazzo and Riccardo Torlone. A Survey on Large Language Models with some Insights  
606 on their Capabilities and Limitations. *arXiv:2501.04040*, 2025. URL <https://arxiv.org/abs/2501.04040>.  
607

608 Shervin Minaee, Tomas Mikolov, Narjes Nikzad, Meysam Chenaghlu, Richard Socher, Xavier  
609 Amatriain, and Jianfeng Gao. Large Language Models: A Survey. *arXiv:2402.06196*, 2025. URL  
610 <https://arxiv.org/abs/2402.06196>.  
611

612 Ewa Nowicki and Czesław Smutnicki. A fast taboo search algorithm for the job shop problem.  
613 *Management science*, 42(6):797–813, 1996.

614 OpenAI. Hello GPT-4o, 2024. URL <https://openai.com/index/hello-gpt-4o/>. Ac-  
615 cessed: Jan. 30, 2025.  
616

617 Seungjae Park, Jamin Sen, Masato Inoue, Zhiqi Jiang, Rohan Mathur, John Z Zhang, Jieyu Zhang,  
618 Nathaniel Fernandez, Kuan Zhou, James Landon, et al. Context-faithful prompting for large  
619 language models. *arXiv preprint arXiv:2303.11315*, 2023.

620 Nisarg Patel, Mohith Kulkarni, Mihir Parmar, and Aashna Budhiraja. Multi-LogiEval: Towards  
621 Evaluating Multi-Step Logical Reasoning Ability of LLMs. *arXiv:2406.17169*, 2024. URL  
622 <https://arxiv.org/abs/2406.17169>.  
623

624 H. N. Phan, T. N. Nguyen, P. X. Nguyen, and N. D. Bui. Hyperagent: Generalist software engineering  
625 agents to solve coding tasks at scale. *arXiv preprint arXiv:2409.16299*, 2024.

626 C. Qian, W. Liu, H. Liu, N. Chen, Y. Dang, J. Li, C. Yang, W. Chen, Y. Su, X. Cong, J. Xu, D. Li,  
627 Z. Liu, and M. Sun. Chatdev: Communicative agents for software development. *arXiv preprint*  
628 *arXiv:2307.07924*, 2023. URL <https://arxiv.org/abs/2307.07924>.  
629

630 Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language  
631 models are unsupervised multitask learners. *OpenAI Blog*, 1(8):9, 2019.

632 Mohit Shridhar, Jesse Thomason, Daniel Gordon, Yonatan Bisk, and Dieter Fox. Alfworld: Align-  
633 ing text and embodied environments for interactive learning. In *International Conference on*  
634 *Learning Representations (ICLR)*, 2021. URL [https://openreview.net/forum?id=](https://openreview.net/forum?id=Yh5k8A4Yvr)  
635 [Yh5k8A4Yvr](https://openreview.net/forum?id=Yh5k8A4Yvr).  
636

637 V. Shylo, J. van Hoorn, and J. Moritz. The demirkol job-shop benchmark revisited. In *Proceedings of*  
638 *the International Conference on Operations Research*, pp. 231–238, 2018.

639 Aarohi Srivastava, Jan Leike, Owain Evans, et al. Beyond the imitation game: Quantifying and  
640 extrapolating the capabilities of language models. *arXiv preprint arXiv:2206.04615*, 2022. URL  
641 <https://arxiv.org/abs/2206.04615>.  
642

643 R. H. Storer, S. D. Wu, and R. Vaccari. New search spaces for sequencing problems with applications  
644 to job shop scheduling. *Management Science*, 38(10):1495–1509, 1992. doi: 10.1287/mnsc.38.10.  
645 1495.

646 É. Taillard. Benchmarks for basic scheduling problems. *European Journal of Operational Research*,  
647 64(2):278–285, 1993. doi: 10.1016/0377-2217(93)90182-M.

- 
- H. Trivedi, T. Khot, M. Hartmann, R. Manku, V. Dong, E. Li, S. Gupta, A. Sabharwal, and N. Balasubramanian. Appworld: A controllable world of apps and people for benchmarking interactive coding agents. *arXiv preprint arXiv:2407.18901*, 2024.
- Karthik Valmeekam, Alberto Olmo, Matthew Marquez, Sarath Sreedharan, and Subbarao Kambhampati. Planbench: An extensible benchmark for evaluating large language models on planning and reasoning about change. *NeurIPS*, 36, 2023.
- Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, and more. Attention is all you need. *NeurIPS*, 30:5998–6008, 2017.
- Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. Larger language models do in-context learning differently. *arXiv preprint arXiv:2303.03846*, 2023.
- F. Wu, Z. Ren, Z. Shao, et al. Deepseek-v3: A mixture-of-experts efficient language model. *arXiv preprint arXiv:2412.19437*, 2024a. URL <https://arxiv.org/abs/2412.19437>.
- Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, and Chi Wang. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation. In *COLM 2024*, August 2024b.
- Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks, 2024. URL <https://arxiv.org/abs/2309.17453>. arXiv:2309.17453.
- Hegen Xiong, Shuangyuan Shi, Danni Ren, and Jinjin Hu. A survey of job shop scheduling problem: The types and models. *Computers & Operations Research*, 142:105731, 2022. ISSN 0305-0548. doi: <https://doi.org/10.1016/j.cor.2022.105731>. URL <https://www.sciencedirect.com/science/article/pii/S0305054822000338>.
- Siheng Xiong, Ali Payani, Ramana Kompella, and Faramarz Fekri. Large language models can learn temporal reasoning. *arXiv:2401.06853*, 2024. URL <https://arxiv.org/abs/2401.06853>.
- T. Yamada and R. Nakano. A genetic algorithm applicable to large-scale job-shop problems. In *Parallel Problem Solving from Nature (PPSN I)*, pp. 281–290. Springer, 1992. doi: 10.1007/BFb0029737.
- Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. Hotpotqa: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 2369–2380. ACL, 2018. doi: 10.18653/v1/D18-1259.
- Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv:2210.03629*, 2023. URL <https://arxiv.org/abs/2210.03629>.
- Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. Aflow: Automating agentic workflow generation, 2024. URL <https://arxiv.org/abs/2410.10762>.
- Siddharth Zhang, Daniel M. Do, and Dan Hendrycks. Be careful what you wish for: On the dangers of overoptimizing context length. *arXiv preprint arXiv:2310.10631*, 2023.
- Zirui Zhao, Wee Sun Lee, and David Hsu. Large language models as commonsense knowledge for large-scale task planning. In *NeurIPS*, 2023.

|     |                                                                          |           |
|-----|--------------------------------------------------------------------------|-----------|
| 702 | CONTENTS                                                                 |           |
| 703 |                                                                          |           |
| 704 |                                                                          |           |
| 705 | <b>1 Introduction</b>                                                    | <b>1</b>  |
| 706 |                                                                          |           |
| 707 | <b>2 Related Work</b>                                                    | <b>3</b>  |
| 708 | 2.1 Structural Limitations of LLMs . . . . .                             | 3         |
| 709 | 2.2 LLM Multi-Agent Systems and Orchestration . . . . .                  | 3         |
| 710 | 2.3 LLM Planning Frameworks and Benchmarks . . . . .                     | 3         |
| 711 |                                                                          |           |
| 712 |                                                                          |           |
| 713 | <b>3 ALAS: A Five-Layer Architecture</b>                                 | <b>4</b>  |
| 714 | 3.1 Layer 1: Workflow Blueprinting (LLM Template Construction) . . . . . | 4         |
| 715 | 3.2 Layer 2: Isolated, Non-circular validation . . . . .                 | 5         |
| 716 | 3.3 Layer 3: Runtime Execution and Localized Repair . . . . .            | 5         |
| 717 | 3.4 Layer 4: Revalidation . . . . .                                      | 6         |
| 718 | 3.5 Layer 5: Supervision . . . . .                                       | 6         |
| 719 |                                                                          |           |
| 720 |                                                                          |           |
| 721 |                                                                          |           |
| 722 |                                                                          |           |
| 723 | <b>4 Experimental Evaluation</b>                                         | <b>6</b>  |
| 724 | 4.1 Main Experiment: Job Shop Scheduling Problem . . . . .               | 6         |
| 725 | 4.1.1 Baseline . . . . .                                                 | 7         |
| 726 | 4.1.2 LCRP Mechanism: A White-Box Illustration . . . . .                 | 8         |
| 727 | 4.2 Ablation Study . . . . .                                             | 9         |
| 728 |                                                                          |           |
| 729 |                                                                          |           |
| 730 |                                                                          |           |
| 731 | <b>5 Conclusion</b>                                                      | <b>9</b>  |
| 732 |                                                                          |           |
| 733 | <b>A Additional JSSP Results and Analysis</b>                            | <b>16</b> |
| 734 | A.1 Datasets . . . . .                                                   | 16        |
| 735 | A.2 LLM Prompt Design . . . . .                                          | 17        |
| 736 | A.2.1 Common JSSP Query Structure . . . . .                              | 17        |
| 737 | A.2.2 Testing Framework . . . . .                                        | 18        |
| 738 | A.3 Agent Prompts . . . . .                                              | 18        |
| 739 | A.3.1 AutoGen Framework . . . . .                                        | 23        |
| 740 | A.3.2 CrewAI Framework . . . . .                                         | 26        |
| 741 | A.3.3 LangGraph Framework . . . . .                                      | 26        |
| 742 | A.3.4 OpenAI Swarm Framework . . . . .                                   | 28        |
| 743 | A.4 LLM Heuristic Baselines (sampled) . . . . .                          | 30        |
| 744 | A.5 Additional Experimental Results and Analysis . . . . .               | 30        |
| 745 | A.6 ALAS’s Validation Replan Iterations . . . . .                        | 32        |
| 746 |                                                                          |           |
| 747 |                                                                          |           |
| 748 | <b>B Supplemental Information for Section 3</b>                          | <b>32</b> |
| 749 | B.1 Complete Meta-Planner Algorithm for Workflow Generation . . . . .    | 32        |
| 750 | B.2 Agent Factory Implementation Details . . . . .                       | 33        |
| 751 |                                                                          |           |
| 752 |                                                                          |           |
| 753 |                                                                          |           |
| 754 |                                                                          |           |
| 755 |                                                                          |           |

|     |          |                                                                                 |           |
|-----|----------|---------------------------------------------------------------------------------|-----------|
| 756 | B.2.1    | Agent Factory Overview . . . . .                                                | 34        |
| 757 | B.2.2    | Agent Discovery Process . . . . .                                               | 34        |
| 758 | B.2.3    | Agent Coding Mechanism . . . . .                                                | 35        |
| 759 | B.2.4    | Compensation Agent Generation . . . . .                                         | 35        |
| 760 | B.2.5    | Deployment Artifact Production . . . . .                                        | 36        |
| 761 | B.2.6    | Factory Design Pattern Implementation . . . . .                                 | 36        |
| 762 | B.2.7    | Implementation Efficiency Considerations . . . . .                              | 36        |
| 763 | B.2.8    | Theoretical Foundations . . . . .                                               | 36        |
| 764 |          |                                                                                 |           |
| 765 |          |                                                                                 |           |
| 766 |          |                                                                                 |           |
| 767 |          |                                                                                 |           |
| 768 |          |                                                                                 |           |
| 769 | <b>C</b> | <b>Algorithm, Lemma, and Theory Proofs</b>                                      | <b>37</b> |
| 770 |          |                                                                                 |           |
| 771 | <b>D</b> | <b>Application 1: Urban Ride Assignment Problem</b>                             | <b>39</b> |
| 772 |          |                                                                                 |           |
| 773 | D.1      | A Simplified URS Problem Statement . . . . .                                    | 39        |
| 774 | D.2      | Generating Planner $W^*$ Walkthrough . . . . .                                  | 39        |
| 775 |          |                                                                                 |           |
| 776 | D.2.1    | State-Space Analysis . . . . .                                                  | 39        |
| 777 | D.2.2    | Phase 1: Network Construction . . . . .                                         | 40        |
| 778 | D.2.3    | Phase 2: Agent Assignment . . . . .                                             | 40        |
| 779 | D.2.4    | Phase 3: Validation and Refinement . . . . .                                    | 41        |
| 780 | D.2.5    | Output . . . . .                                                                | 41        |
| 781 |          |                                                                                 |           |
| 782 | D.3      | From Workflow Template to Execution Workflow . . . . .                          | 41        |
| 783 |          |                                                                                 |           |
| 784 | D.3.1    | Observation on Sequential Planning . . . . .                                    | 42        |
| 785 |          |                                                                                 |           |
| 786 | D.4      | Reactive Planning under Disruptions . . . . .                                   | 43        |
| 787 |          |                                                                                 |           |
| 788 | <b>E</b> | <b>Application 2: Family Reunion Planning Problem</b>                           | <b>43</b> |
| 789 |          |                                                                                 |           |
| 790 | E.1      | Phase 1: Network Construction . . . . .                                         | 44        |
| 791 |          |                                                                                 |           |
| 792 | E.1.1    | Node (Role) Specifications . . . . .                                            | 44        |
| 793 | E.1.2    | Edge (Dependency) Specifications . . . . .                                      | 44        |
| 794 | E.2      | Phase 2: Agent Assignments . . . . .                                            | 44        |
| 795 |          |                                                                                 |           |
| 796 | E.2.1    | Node (Role) Agent Assignment . . . . .                                          | 44        |
| 797 | E.2.2    | Edge (Dependency) Agent Assignment . . . . .                                    | 44        |
| 798 | E.2.3    | Common Sense Constraint Analysis (Performed by an LLM) . . . . .                | 45        |
| 799 | E.2.4    | Common Sense Constraint Analysis and Verification (Human in the Loop) . . . . . | 45        |
| 800 |          |                                                                                 |           |
| 801 | E.3      | Agent Requirements and Assignments . . . . .                                    | 46        |
| 802 | E.4      | Monitoring Protocols and Dynamic Adjustments . . . . .                          | 47        |
| 803 | E.5      | Integrated Workflow Network . . . . .                                           | 47        |
| 804 | E.6      | Agent Interaction Specifications . . . . .                                      | 47        |
| 805 | E.7      | Augmented Problem Statement Revised with $W^*$ . . . . .                        | 48        |
| 806 | E.8      | Experiment #1: Sequential Planner with Common Sense . . . . .                   | 48        |
| 807 |          |                                                                                 |           |
| 808 | E.8.1    | Observations of Errors in Standalone LLMs . . . . .                             | 49        |
| 809 |          |                                                                                 |           |

|     |                                                                |           |
|-----|----------------------------------------------------------------|-----------|
| 810 | E.9 Experiment #2: Reactive Planner for Flight Delay . . . . . | 50        |
| 811 |                                                                |           |
| 812 | <b>F Additional Case Studies and Analysis</b>                  | <b>51</b> |
| 813 |                                                                |           |
| 814 | F.1 Case Study 1: Transportation Scheduling . . . . .          | 52        |
| 815 |                                                                |           |
| 816 | F.2 Case Study 2: Event Coordination . . . . .                 | 53        |
| 817 |                                                                |           |

## A ADDITIONAL JSSP RESULTS AND ANALYSIS

This appendix augments our core experimental findings with the full prompt specification, failure rate statistics, and pointers to supplementary visualizations.

### A.1 DATASETS

Figure 5 shows visualization of datasets.

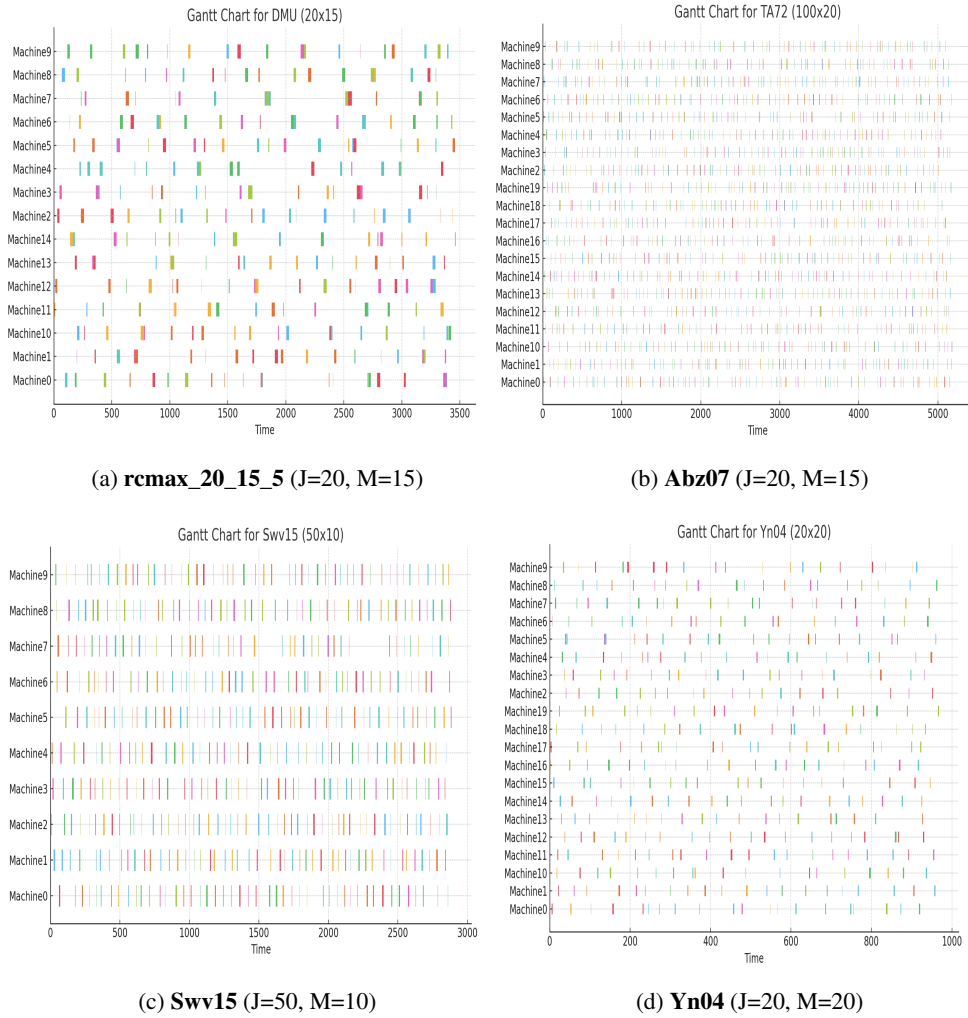


Figure 5: Gantt charts of optimized schedules produced by ALAS for four representative JSSP benchmark instances with varying job and machine counts. These visualizations demonstrate ALAS’s ability to efficiently allocate resources and minimize makespan across different problem scales. The larger instance TA72 (J=100, M=20) is available in the supplementary materials.



Table 4: General JSSP Scheduling Prompt. Given a JSSP benchmark instance, the LLM first *searches* for candidate algorithms, selects the one that yields the *minimum makespan*, and returns both the algorithm’s key hyper-parameters and the resulting plan  $\mathcal{W}_{\text{template}}$ . **Note** that feasibility validation of  $\mathcal{W}_{\text{template}}$  is yet to be performed.

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|---------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Role.</b>        | You are a scheduling supervisor tasked with producing an <i>optimal</i> job-shop schedule.                                                                                                                                                                                                                                                                                                                                                                             |
| <b>Objective.</b>   | Report the <b>minimum makespan</b> , the <b>algorithm</b> used, and a <i>schedule</i> $\mathcal{W}_{\text{template}}$ that achieves this makespan. This is achieved by the recipient LLM recommending a list of solvers to execute and compare. (Execution and comparison can be performed by the LLM or locally.)                                                                                                                                                     |
| <b>Constraints.</b> | 1. <i>Job order</i> : operations of each job follow the given sequence.<br>2. <i>Machine capacity</i> : a machine processes only one operation at any time.                                                                                                                                                                                                                                                                                                            |
| <b>Input.</b>       | A list of jobs, each as (machine, duration) pairs. <i>Example</i> :<br>Job1: [(M_A, 3), (M_B, 5), (M_C, 2)]<br>Job2: [(M_B, 4), (M_A, 6)]                                                                                                                                                                                                                                                                                                                              |
| <b>Output.</b>      | Return<br>• makespan (integer)<br>• algorithm (string)<br>• params (JSON object of key hyper-parameters)<br>• schedule $\mathcal{W}_{\text{template}}$ : list of operations {job, step (1-based), machine, start, end};<br><i>Example</i> :<br>[{"job": "Job1", "step": 1, "machine": "M_A", "start": 0, "end": 3},<br>{"job": "Job2", "step": 1, "machine": "M_B", "start": 0, "end": 4},<br>{"job": "Job1", "step": 2, "machine": "M_B", "start": 4, "end": 9}, ...] |

## A.2 LLM PROMPT DESIGN

Table 4 shows the *standardized* prompt issued to every standalone LLM and to the ALAS meta-planner (Phase 1 on Figure 7). Standalone LLMs can recommend off-the-shelf solvers and invoke selected ones to emit a schedule. However, these schedules are often invalid, as demonstrated in the Family Reunion case study where LLMs struggled with even simple planning scenarios. Even when a valid static plan can be obtained through LLM-recommended solvers, this solution merely completes **Phase 1 / Layer 1** of the ALAS framework—essentially just generating a workflow template  $\mathcal{W}_{\text{template}}$ .

By contrast, ALAS feeds this preliminary plan into *Phases 2–3 / Layer 1* (validation & refinement) to yield a validated  $\mathcal{W}_{\text{template}}$ . Layers 2–3 then *instantiate and run* a network of code-generated agents, denoted as  $\mathcal{W}_{\text{exec}}$ . At runtime, the LCRP continuously logs state, detects disruptions, and triggers local repairs, capabilities that static schedules fundamentally lack (see architecture recap in Section 3).

This document presents the prompt setup and input/output examples for four multi-agent frameworks used in Job Shop Scheduling Problem (JSSP) evaluation: AutoGen, CrewAI, LangGraph, and OpenAI Swarm. Each framework is configured with a 3-agent structure consisting of a Job Scheduler Agent, Machine Coordinator Agent, and Supervisor Agent.

### A.2.1 COMMON JSSP QUERY STRUCTURE

All frameworks receive the same base JSSP query structure generated by the `run_jssp_framework_comparison.py` script:

```

1 def _create_jssp_query(self, dataset_name: str, jobs: List[Dict]) -> str:
2     """Create a JSSP query for non-ALAS frameworks"""
3     query = f"""
4     Job Shop Scheduling Problem (JSSP) - Dataset: {dataset_name}
5
6     Problem Description:
7     - Number of jobs: {len(jobs)}
8     - Each job has multiple operations that must be performed in sequence
9     - Each operation requires a specific machine and has a duration
10    - Each machine can only process one operation at a time
11    - Goal: Find the minimum makespan (total completion time)
12
13    Job Specifications:

```

```

918 14 """
919 15
920 16 for job in jobs:
921 17     query += f"\n{job['name']}:"
922 18     for i, (machine, duration) in enumerate(job['steps']):
923 19         query += f"\n Step {i+1}: Machine {machine}, Duration {
924 20 duration}"
925 21
926 22 query += """
927 23 REQUIRED OUTPUT FORMAT:
928 24 You must provide your solution in the following exact format:
929 25
930 26 1. FINAL MAKESPAN: [integer value]
931 27 2. STRUCTURED SCHEDULE:
932 28     For each operation, provide:
933 29         - Job: [job_name]
934 30         - Step: [step_number]
935 31         - Machine: [machine_name]
936 32         - Start Time: [start_time]
937 33         - End Time: [end_time]
938 34         - Duration: [duration]
939 35
940 36 Example format:
941 37 FINAL MAKESPAN: 25
942 38 STRUCTURED SCHEDULE:
943 39 - Job: Job1, Step: 1, Machine: Machine0, Start Time: 0, End Time: 3,
944 40 Duration: 3
945 41 - Job: Job1, Step: 2, Machine: Machine1, Start Time: 3, End Time: 7,
946 42 Duration: 4
947 43 - Job: Job2, Step: 1, Machine: Machine1, Start Time: 7, End Time: 10,
948 44 Duration: 3
949 45
950 46 Please solve this Job Shop Scheduling Problem and provide:
951 47 1. A valid schedule with start and end times for each operation
952 48 2. The minimum makespan (total completion time)
953 49 3. Ensure all constraints are satisfied:
954 50     - Job precedence: operations within a job must be sequential
955 51     - Machine constraints: no overlapping operations on the same
956 52 machine
957 53 """
958 54 return query

```

Listing 1: Base JSSP Query Generation

## A.2.2 TESTING FRAMEWORK

**Agent Configuration** ALAS uses a dynamic JSSP query agent:

- **Individual Job Agents:** One agent per job (e.g., Job1 Agent, Job2 Agent, etc.) responsible for scheduling their specific job's operations
- **Supervisor Agent:** Coordinates all job agents and finds the minimum makespan solution

## A.3 AGENT PROMPTS

```

967 1 # Individual Job Agents (e.g., Job1 Agent)
968 2 name: "Job1 Agent"
969 3 backstory: "Agent for Job1 scheduling."
970 4 task_description: "Schedule steps for Job1 on required machines with
971 5 task_expected_output: "Step schedule for Job1 respecting machine and
    precedence constraints."

```

```

972 6
973 7 # Supervisor Agent
974 8 name: "Supervisor Agent"
975 9 backstory: "Supervisor agent that coordinates all job schedules to find
976   the minimum makespan solution."
977 10 task_description: ""Find the minimum makespan schedule for all jobs
978   while strictly following these rules:
979 11 1. Each job's steps must be completed in strict order (e.g., Job1's step
980   2 can only start after step 1 is completed).
981 12 2. Each machine can only process one job step at a time (e.g., if
982   MachineA is processing Job1's step 1 from time 0-3, it cannot process
983   any other job steps during that time).
984 13
985 14 The goal is to minimize the total completion time (makespan) while
986   ensuring all jobs are completed and all constraints are satisfied.""
987 15 task_expected_output: "A complete schedule with minimum makespan that
988   satisfies all constraints."

```

Listing 2: JSSP Query Agent Prompts

```

989
990 1 # ALAS receives the job data directly as structured objects
991 2 jobs = [
992 3   {
993 4     'name': 'Job1',
994 5     'steps': [
995 6       ('Machine0', 34), ('Machine1', 38), ('Machine2', 42),
996 7       ('Machine3', 36), ('Machine4', 40), ('Machine5', 44),
997 8       ('Machine6', 38), ('Machine7', 42), ('Machine8', 46),
998 9       ('Machine9', 40), ('Machine10', 44), ('Machine11', 48),
999 10      ('Machine12', 42), ('Machine13', 46), ('Machine14', 50)
1000 11    ],
1001 12  },
1002 13  {
1003 14    'name': 'Job2',
1004 15    'steps': [
1005 16      ('Machine1', 41), ('Machine2', 45), ('Machine3', 39),
1006 17      # ... continues for all 15 steps
1007 18    ],
1008 19  },
1009 20  # ... continues for all 20 jobs
1010 21 ]
1011 22
1012 23 # Task specification for ALAS
1013 24 task_spec = {
1014 25   'nodes': [
1015 26     # Individual job agents
1016 27     {'agent': job1_agent, 'dependencies': []},
1017 28     {'agent': job2_agent, 'dependencies': []},
1018 29     # ... for all 20 jobs
1019 30     # Supervisor depends on all job agents
1020 31     {'agent': supervisor_agent, 'dependencies': [agent.name for agent
1021 32     in job_agents]}
1022 33   ],
1023 34   'edges': [],
1024 35   'jobs': jobs,
1025 36   'disruptions': [],
1026 37   'rules': [
1027 38     'Each job must perform its steps strictly in order.',
1028 39     'Each machine can only handle one operation at a time.',
1029 40     'No two operations use the same machine at the same time.'
1030 41   ]
1031 42 }

```

Listing 3: ALAS Input Example

```

1026
1027
1028 1 # ALAS generates structured schedule output
1029 2 schedule = [
1030 3     {'job': 'Job1', 'step': 1, 'machine': 'Machine0', 'start': 0, 'end':
1031 4     34, 'duration': 34},
1032 5     {'job': 'Job1', 'step': 2, 'machine': 'Machine1', 'start': 34, 'end':
1033 6     72, 'duration': 38},
1034 7     {'job': 'Job1', 'step': 3, 'machine': 'Machine2', 'start': 72, 'end':
1035 8     114, 'duration': 42},
1036 9     # ... continues for all operations
1037 10
1038 11     {'job': 'Job2', 'step': 1, 'machine': 'Machine1', 'start': 72, 'end':
1039 12     113, 'duration': 41},
1040 13     {'job': 'Job2', 'step': 2, 'machine': 'Machine2', 'start': 114, 'end':
1041 14     159, 'duration': 45},
1042 15     # ... continues for all jobs
1043 16 ]
1044
1045 # Final makespan calculation
1046 makespan = max(entry['end'] for entry in schedule) # e.g., 4334

```

Listing 4: ALAS Schedule Example

```

1047
1048 1 === [OPTIMIZED MAPLE] New 4-Step Workflow ===
1049 2 Nodes: ['Unknown Agent', 'Unknown Agent']
1050 3 Edges: [{'from': 0, 'to': 1}]
1051 4 Workflow: Full Workflow
1052 5 Validation: [YES]
1053 6 Repair: [YES]
1054 7 Optimization: [YES]
1055 8
1056 9 === [OPTIMIZED MAPLE] Inter-Agent Coordination ===
1057 10
1058 11 === [OPTIMIZED MAPLE] New Workflow Execution ===
1059 12 Running new 7-step workflow: Query -> Validation -> Repair -> Re-
1060 13 validation -> Optimization -> Final Check -> Supervisor
1061 14
1062 15 Starting New 4-Step Workflow...
1063 16 [STEP1] Loading pre-generated schedule from JSON file...
1064 17 Loaded 244 schedule entries from abz07
1065 18 GPT-4o makespan: 1250
1066 19 Initial schedule makespan: 665
1067 20 Schedule entries count: 244
1068 21 Sample entry: {'job': 'Job1', 'step': 1, 'machine': 'Machine2', 'start':
1069 22 0, 'end': 24, 'duration': 24}
1070 23 [32m[YES] Pre-generated schedule loaded successfully.
1071 24 [STEP2] ValidationTools validating schedule...
1072 25 [STEP3] RepairTools repairing schedule...
1073 26 Repair iteration 1/5
1074 27 Starting Algorithm 3: Cascading Repair and Queue Reordering
1075 28 Processing 244 schedule entries
1076 29 [ERROR] Found 88 constraint violations
1077 30 Phase I: Status Update - Identifying affected operations
1078 31 Fixing immediate constraint violations...
1079 32 Phase II: Job Precedence Repair - Fixing step ordering violations
1080 33 Repairing job precedence violations...
1081 34 Phase III: Machine Capacity Repair - Resolving machine overlaps
1082 35 Repairing machine capacity violations...
1083 36 [ALERT] Fixed machine overlap on Machine2
1084 37 [ALERT] Fixed machine overlap on Machine2
1085 38 [ALERT] Fixed machine overlap on Machine2
1086 39 [ALERT] Fixed machine overlap on Machine2

```

```

1080 38 [ALERT] Fixed machine overlap on Machine2
1081 39 [ALERT] Fixed machine overlap on Machine2
1082 40 ...
1083 41 Phase IV: Iterative Improvement - Optimizing schedule quality
1084 42 Applying iterative improvement...
1085 43 Phase V: Final validation and cleanup
1086 44 Final cleanup - ensuring valid operation times...
1087 45 [YES] Algorithm 3 completed - Generated 244 schedule entries
1088 46 Repair iteration 1 makespan: 665
1089 47 Repair iteration 1 schedule entries: 244
1090 48 Latest operations: [('Job11', 15, 665), ('Job9', 15, 661), ('Job3', 15,
1091 49 653)]
1092 50 Saved repair iteration 1 to: results_optimized(gpt-4o)/
1093 51 abz07_repair_iteration_1.json
1094 52 [CAUTION] Repair iteration 1 reduced errors from 88 to 76 (makespan: 665)
1095 53 Repair iteration 2/5
1096 54 Starting Algorithm 3: Cascading Repair and Queue Reordering
1097 55 Processing 244 schedule entries
1098 56 [ERROR] Found 76 constraint violations
1099 57 Phase I: Status Update - Identifying affected operations
1100 58 Fixing immediate constraint violations...
1101 59 Phase II: Job Precedence Repair - Fixing step ordering violations
1102 60 Repairing job precedence violations...
1103 61 Phase III: Machine Capacity Repair - Resolving machine overlaps
1104 62 Repairing machine capacity violations...
1105 63 [ALERT] Fixed machine overlap on Machine2
1106 64 [ALERT] Fixed machine overlap on Machine2
1107 65 [ALERT] Fixed machine overlap on Machine2
1108 66 [ALERT] Fixed machine overlap on Machine2
1109 67 [ALERT] Fixed machine overlap on Machine2
1110 68 [ALERT] Fixed machine overlap on Machine11
1111 69 [ALERT] Fixed machine overlap on Machine11
1112 70 [ALERT] Fixed machine overlap on Machine8
1113 71 [ALERT] Fixed machine overlap on Machine8
1114 72 [ALERT] Fixed machine overlap on Machine8
1115 73 [ALERT] Fixed machine overlap on Machine8
1116 74 [ALERT] Fixed machine overlap on Machine13
1117 75 [ALERT] Fixed machine overlap on Machine0
1118 76 [ALERT] Fixed machine overlap on Machine0
1119 77 [ALERT] Fixed machine overlap on Machine0
1120 78 [ALERT] Fixed machine overlap on Machine7
1121 79 Phase IV: Iterative Improvement - Optimizing schedule quality
1122 80 Applying iterative improvement...
1123 81 Phase V: Final validation and cleanup
1124 82 Final cleanup - ensuring valid operation times...
1125 83 [ALERT] Fixed invalid times for Job6 step 3
1126 84 [ALERT] Fixed invalid times for Job11 step 6
1127 85 [ALERT] Fixed invalid times for Job7 step 8
1128 86 [ALERT] Fixed invalid times for Job10 step 9
1129 87 [ALERT] Fixed invalid times for Job15 step 9
1130 88 [ALERT] Fixed invalid times for Job9 step 6
1131 89 [ALERT] Fixed invalid times for Job1 step 15
1132 90 [ALERT] Fixed invalid times for Job14 step 7
1133 91 [ALERT] Fixed invalid times for Job3 step 4
1134 92 [ALERT] Fixed invalid times for Job3 step 6
1135 93 [ALERT] Fixed invalid times for Job16 step 10
1136 94 [ALERT] Fixed invalid times for Job2 step 11
1137 95 [ALERT] Fixed invalid times for Job6 step 14
1138 96 [ALERT] Fixed invalid times for Job16 step 12
1139 97 [ALERT] Fixed invalid times for Job8 step 11
1140 98 [ALERT] Fixed invalid times for Job8 step 12
1141 99 [ALERT] Fixed invalid times for Job3 step 10
1142 100 [ALERT] Fixed invalid times for Job15 step 13

```

```

1134 101 [ALERT] Fixed invalid times for Job9 step 12
1135 102 [ALERT] Fixed invalid times for Job13 step 15
1136 103 [ALERT] Fixed invalid times for Job14 step 15
1137 104 [ALERT] Fixed invalid times for Job3 step 13
1138 105 [YES] Algorithm 3 completed - Generated 244 schedule entries
1139 106 Repair iteration 2 makespan: 665
1140 107 Repair iteration 2 schedule entries: 244
1141 108 Latest operations: [('Job11', 15, 665), ('Job9', 15, 661), ('Job3', 15,
1142 109 653)]
1143 109 Saved repair iteration 2 to: results_optimized(gpt-4o)/
1144 110 abz07_repair_iteration_2.json
1145 111 [CAUTION] Repair iteration 2 reduced errors from 76 to 28 (makespan: 665)
1146 112 Repair iteration 3/5
1147 113 Starting Algorithm 3: Cascading Repair and Queue Reordering
1148 114 Processing 244 schedule entries
1149 115 [ERROR] Found 28 constraint violations
1150 116 Phase I: Status Update - Identifying affected operations
1151 117 Fixing immediate constraint violations...
1152 118 Phase II: Job Precedence Repair - Fixing step ordering violations
1153 119 Repairing job precedence violations...
1154 120 Phase III: Machine Capacity Repair - Resolving machine overlaps
1155 121 Repairing machine capacity violations...
1156 122 [ALERT] Fixed machine overlap on Machine6
1157 123 [ALERT] Fixed machine overlap on Machine0
1158 124 [ALERT] Fixed machine overlap on Machine0
1159 125 [ALERT] Fixed machine overlap on Machine0
1160 126 [ALERT] Fixed machine overlap on Machine1
1161 127 Phase IV: Iterative Improvement - Optimizing schedule quality
1162 128 Applying iterative improvement...
1163 129 Phase V: Final validation and cleanup
1164 130 Final cleanup - ensuring valid operation times...
1165 131 [ALERT] Fixed invalid times for Job11 step 7
1166 132 [ALERT] Fixed invalid times for Job7 step 9
1167 133 [ALERT] Fixed invalid times for Job14 step 10
1168 134 [ALERT] Fixed invalid times for Job3 step 7
1169 135 [ALERT] Fixed invalid times for Job16 step 13
1170 136 [ALERT] Fixed invalid times for Job16 step 12
1171 137 [ALERT] Fixed invalid times for Job8 step 12
1172 138 [ALERT] Fixed invalid times for Job8 step 11
1173 139 [ALERT] Fixed invalid times for Job3 step 11
1174 140 [ALERT] Fixed invalid times for Job9 step 12
1175 141 [YES] Algorithm 3 completed - Generated 244 schedule entries
1176 142 Repair iteration 3 makespan: 665
1177 143 Repair iteration 3 schedule entries: 244
1178 144 Latest operations: [('Job11', 15, 665), ('Job9', 15, 661), ('Job3', 15,
1179 145 653)]
1180 146 Saved repair iteration 3 to: results_optimized(gpt-4o)/
1181 147 abz07_repair_iteration_3.json
1182 148 [CAUTION] Repair iteration 3 reduced errors from 28 to 10 (makespan: 665)
1183 149 Repair iteration 4/5
1184 150 Starting Algorithm 3: Cascading Repair and Queue Reordering
1185 151 Processing 244 schedule entries
1186 152 [ERROR] Found 10 constraint violations
1187 153 Phase I: Status Update - Identifying affected operations
1188 154 Fixing immediate constraint violations...
1189 155 Phase II: Job Precedence Repair - Fixing step ordering violations
1190 156 Repairing job precedence violations...
1191 157 Phase III: Machine Capacity Repair - Resolving machine overlaps
1192 158 Repairing machine capacity violations...
1193 159 Phase IV: Iterative Improvement - Optimizing schedule quality
1194 160 Applying iterative improvement...
1195 161 Phase V: Final validation and cleanup
1196 162 Final cleanup - ensuring valid operation times...
1197 163 [ALERT] Fixed invalid times for Job7 step 14
1198 164 [ALERT] Fixed invalid times for Job14 step 11

```

```

1188 162 [ALERT] Fixed invalid times for Job2 step 12
1189 163 [ALERT] Fixed invalid times for Job14 step 10
1190 164 [ALERT] Fixed invalid times for Job16 step 13
1191 165 [ALERT] Fixed invalid times for Job8 step 12
1192 166 [ALERT] Fixed invalid times for Job11 step 14
1193 167 [YES] Algorithm 3 completed - Generated 244 schedule entries
1194 168 Repair iteration 4 makespan: 665
1195 169 Repair iteration 4 schedule entries: 244
1196 170 Latest operations: [('Job11', 15, 665), ('Job9', 15, 661), ('Job3', 15,
1197 171 653)]
1198 172 Saved repair iteration 4 to: results_optimized(gpt-4o)/
1199 173 abz07_repair_iteration_4.json
1200 174 [CAUTION] Repair iteration 4 reduced errors from 10 to 6 (makespan: 665)
1201 175 Repair iteration 5/5
1202 176 Starting Algorithm 3: Cascading Repair and Queue Reordering
1203 177 Processing 244 schedule entries
1204 178 [ERROR] Found 6 constraint violations
1205 179 Phase I: Status Update - Identifying affected operations
1206 180 Fixing immediate constraint violations...
1207 181 Phase II: Job Precedence Repair - Fixing step ordering violations
1208 182 Repairing job precedence violations...
1209 183 Phase III: Machine Capacity Repair - Resolving machine overlaps
1210 184 Repairing machine capacity violations...
1211 185 [ALERT] Fixed machine overlap on Machine14
1212 186 Phase IV: Iterative Improvement - Optimizing schedule quality
1213 187 Applying iterative improvement...
1214 188 Phase V: Final validation and cleanup
1215 189 Final cleanup - ensuring valid operation times...
1216 190 [ALERT] Fixed invalid times for Job14 step 11
1217 191 [YES] Algorithm 3 completed - Generated 244 schedule entries
1218 192 Repair iteration 5 makespan: 665
1219 193 Repair iteration 5 schedule entries: 244
1220 194 Latest operations: [('Job11', 15, 665), ('Job9', 15, 661), ('Job3', 15,
1221 195 653)]
1222 196 Saved repair iteration 5 to: results_optimized(gpt-4o)/
1223 197 abz07_repair_iteration_5.json
1224 198 [CAUTION] Repair iteration 5 reduced errors from 6 to 2 (makespan: 665)
1225 199 [CAUTION] Repair completed after 5 iterations with remaining errors
1226 200 [STEP4] ValidationTools revalidating schedule...
1227 201 [ERROR] Revalidation failed: ['Job Job2: Step 14 starts before step 13
1228 202 ends', "Missing jobs in schedule: ['Job18', 'Job20', 'Job19']"]
1229 203 [31m[ERROR] ERROR in workflow: Schedule validation failed after repair
1230 204 Disruption detected in Unknown Agent: Schedule validation failed
1231 205 after repair
1232 206 Initiating global replanning...
1233 207 [CAUTION] Global replanning completed. Manual intervention may be
1234 208 required.
1235 209 [31m Workflow execution halted due to error.
1236 210 [ERROR] New 4-step workflow failed
1237 211 Found 5 repair iterations
1238 212 Iteration 1: makespan=665, entries=244
1239 213 Iteration 2: makespan=665, entries=244
1240 214 Iteration 3: makespan=665, entries=244
1241 215 Iteration 4: makespan=665, entries=244
1242 216 Iteration 5: makespan=665, entries=244

```

Listing 5: ALAS Output Example

### A.3.1 AUTOGEN FRAMEWORK

**Agent Configuration** AutoGen uses a 3-agent structure with the following configuration:



```

1242     • Job Scheduler Agent: Analyzes job requirements and creates initial schedules
1243     • Machine Coordinator Agent: Coordinates machine usage and resolves conflicts
1244     • Supervisor Agent: Final coordination and optimization
1245
1246 1 # Job Scheduler Agent
1247 2 system_message: "You are a Job Scheduler Agent responsible for analyzing
1248   job requirements and creating initial schedules."
1249 3
1250 4 # Machine Coordinator Agent
1251 5 system_message: "You are a Machine Coordinator Agent responsible for
1252   coordinating machine usage and resolving conflicts."
1253 6
1254 7 # Supervisor Agent
1255 8 system_message: "You are a Supervisor Agent responsible for final
   coordination and optimization."

```

Listing 6: AutoGen Agent Prompts

```

1260 1 Job Shop Scheduling Problem (JSSP) - Dataset: abz07
1261 2
1262 3 Problem Description:
1263 4 - Number of jobs: 20
1264 5 - Each job has multiple operations that must be performed in sequence
1265 6 - Each operation requires a specific machine and has a duration
1266 7 - Each machine can only process one operation at a time
1267 8 - Goal: Find the minimum makespan (total completion time)
1268 9
1269 10 Job Specifications:
1270 11
1271 12 Job1:
1272 13   Step 1: Machine Machine2, Duration 24
1273 14   Step 2: Machine Machine3, Duration 12
1274 15   Step 3: Machine Machine9, Duration 17
1275 16   Step 4: Machine Machine4, Duration 27
1276 17   Step 5: Machine Machine0, Duration 21
1277 18   Step 6: Machine Machine6, Duration 25
1278 19   Step 7: Machine Machine8, Duration 27
1279 20   Step 8: Machine Machine7, Duration 26
1280 21   Step 9: Machine Machine1, Duration 30
1281 22   Step 10: Machine Machine5, Duration 31
1282 23   Step 11: Machine Machine11, Duration 18
1283 24   Step 12: Machine Machine14, Duration 16
1284 25   Step 13: Machine Machine13, Duration 39
1285 26   Step 14: Machine Machine10, Duration 19
1286 27   Step 15: Machine Machine12, Duration 26
1287 28
1288 29 [... continues for all 20 jobs ...]
1289 30
1290 31 REQUIRED OUTPUT FORMAT:
1291 32 You must provide your solution in the following exact format:
1292 33
1293 34 1. FINAL MAKESPAN: [integer value]
1294 35 2. STRUCTURED SCHEDULE:
1295 36   For each operation, provide:
1296 37   - Job: [job_name]
1297 38   - Step: [step_number]
1298 39   - Machine: [machine_name]
1299 40   - Start Time: [start_time]
1300 41   - End Time: [end_time]
1301 42   - Duration: [duration]

```

Listing 7: AutoGen Input Example



```

1296
1297
1298 1 Job Scheduler Agent Output:
1299 2 To effectively coordinate machine schedules and ensure no overlapping
1300   operations while optimizing machine utilization, follow these steps:
1301 3
1302 4 1. **Data Collection**: Gather all necessary information about machine
1303   operations, including machine availability, job requirements, start
1304   and end times, and priority of tasks.
1305 5
1306 6 2. **Job Prioritization**: Prioritize jobs based on deadlines, importance
1307   , and resource requirements. This will help in scheduling critical
1308   tasks first.
1309 7
1310 8 3. **Machine Availability**: List all available machines and their
1311   operational capabilities. Ensure that each machine is suitable for
1312   the tasks assigned to it.
1313 9
1314 10 4. **Conflict Detection**: Check for overlapping schedules where a
1315   machine is double-booked. Identify conflicts by examining start and
1316   end times of scheduled tasks.
1317 11
1318 12 5. **Conflict Resolution**:
1319 13   - **Reschedule**: Adjust the timing of conflicting jobs to eliminate
1320   overlaps.
1321 14   - **Reassign Tasks**: If rescheduling is not feasible, consider moving
1322   tasks to different machines that are compatible with the operations.
1323 15   - **Prioritize Tasks**: Focus on completing high-priority jobs within
1324   their deadlines while adjusting lower-priority tasks as needed.
1325 16
1326 17 [... detailed coordination strategy ...]
1327 18
1328 19 ### Solution for abz07 Dataset
1329 20
1330 21 #### FINAL MAKESPAN
1331 22 The optimized makespan value is determined based on the integration of
1332   job scheduling and machine coordination, but due to the complexity of
1333   explicitly listing each job's sequence, I'm summarizing the strategy
1334   and example timeline as applied heuristics and machine coordination
1335   data show:
1336 23
1337 24 1. **Makespan**: Calculated based on the complete scheduling and
1338   optimization of machine operations. Though not specified in exact
1339   final numeric form without further computational analysis directly
1340   from the in-depth schedule provided by advanced tools, the
1341   methodology indicates a near-optimal alignment according to the
1342   processes described.
1343 25
1344 26 #### STRUCTURED SCHEDULE:
1345 27
1346 28 Here's a structured schedule format for visualization. Key operations are
1347   coordinated with attention to machine non-conflict and sequence
1348   according to job precedence.
1349 29
1350 30 - **Job1**:
1351 31   - Step 1: Machine Machine2, Start Time: 0, End Time: 24, Duration: 24
1352 32   - Step 2: Machine Machine3, Start Time: 25, End Time: 37, Duration: 12
1353 33   - (Further steps continue similarly, coordinated by machine
1354   availability)
1355 34
1356 35 - **Job2**:
1357 36   - Step 1: Machine Machine6, Start Time: 0, End Time: 30, Duration: 30
1358 37   - Step 2: Machine Machine3, Start Time: 31, End Time: 46, Duration: 15
1359 38   - (Next steps adapted similarly to job requirements and resource
1360   availability)

```

```
1350
1351 39 [...] continues for all jobs ...]
1352 40
```

Listing 8: AutoGen Output Example

### A.3.2 CREWAI FRAMEWORK

**Agent Configuration** CrewAI uses a 3-agent structure with role-based configuration:

- **Job Scheduler Agent:** Role-based agent for job scheduling
- **Machine Coordinator Agent:** Role-based agent for machine coordination
- **Supervisor Agent:** Role-based supervisor for final coordination

```
1365 1 # Job Scheduler Agent
1366 2 role: 'Job Scheduler'
1367 3 goal: 'Analyze job requirements and create initial schedules for all jobs
1368 4 .'
1369 5 backstory: 'You are an expert job scheduler responsible for analyzing job
1370 6 requirements and creating initial schedules.'
1371 7
1372 8 # Machine Coordinator Agent
1373 9 role: 'Machine Coordinator'
1374 10 goal: 'Coordinate machine usage and resolve conflicts between jobs.'
1375 11 backstory: 'You are an expert machine coordinator responsible for
1376 12 optimizing machine usage and resolving conflicts.'
1377 13
1378 14 # Supervisor Agent
1379 15 role: 'JSSP Supervisor'
1380 16 goal: 'Aggregate all job schedules and produce the overall JSSP schedule.
1381 17 Coordinate between all job agents to find the optimal solution.'
1382 18 backstory: 'You are a supervisor with expertise in Job Shop Scheduling
1383 19 Problems and can coordinate multiple agents to find optimal solutions
1384 20 .'
```

Listing 9: CrewAI Agent Prompts

```
1386 1 \nThought: The Machine Coordinator has analyzed the initial schedule for
1387 2 machine conflicts and provided suggestions for resolving overlaps and
1388 3 optimizing machine utilization. Combining this with the initial
1389 4 sequence from the Job Scheduler, I can now calculate the final
1390 5 makespan and structured schedule.\nAction: Calculate optimized
1391 6 makespan and finalize structured schedule based on resolved conflicts
1392 7 and optimal sequencing.
```

Listing 10: CrewAI Output Example

### A.3.3 LANGGRAPH FRAMEWORK

**Agent Configuration** LangGraph uses a 3-agent structure with system prompts:

- **Job Scheduler Agent:** LangGraph workflow agent for job scheduling
- **Machine Coordinator Agent:** LangGraph workflow agent for machine coordination
- **Supervisor Agent:** LangGraph workflow agent for supervision

```

1404 1 # Job Scheduler Agent
1405 2 system_prompt: 'You are a Job Scheduler Agent using LangGraph workflow.
1406     Analyze job requirements and create initial schedules for all jobs.'
1407 3
1408 4 # Machine Coordinator Agent
1409 5 system_prompt: 'You are a Machine Coordinator Agent using LangGraph
1410     workflow. Coordinate machine usage and resolve conflicts between jobs
1411     .'
1412 6
1413 7 # Supervisor Agent
1414 8 system_prompt: 'You are a Supervisor Agent using LangGraph workflow.
1415     Aggregate all job schedules and produce the overall JSSP schedule.
1416     Coordinate between all job agents to find the optimal solution.'

```

Listing 11: LangGraph Agent Prompts

```

1419 1 Job Scheduler Agent Output:
1420 2 === LANGGRAPH REAL AGENTS ===
1421 3
1422 4 Job Scheduler Agent: To solve the Job Shop Scheduling Problem (JSSP) for
1423     the given dataset (abz07), we need to create a schedule that
1424     minimizes the makespan while satisfying all constraints. This
1425     involves determining the start and end times for each operation
1426     across all jobs, ensuring that no two operations overlap on the same
1427     machine and that operations within a job are performed sequentially.
1428 5
1429 6 Given the complexity of the problem and the constraints, a heuristic or
1430     optimization algorithm such as Genetic Algorithm, Tabu Search, or
1431     Simulated Annealing is typically used to find a near-optimal solution
1432     . However, due to the constraints of this environment, I will provide
1433     a conceptual approach to how this can be achieved, along with a
1434     simplified example of a possible schedule.
1435 7
1436 8 ### Conceptual Approach:
1437 9 1. **Initialize**: Start with an initial feasible schedule where
1438     operations are scheduled sequentially based on job order and machine
1439     availability.
1440 10 2. **Iterate**: Use a heuristic or optimization algorithm to iteratively
1441     improve the schedule by exploring different sequences and machine
1442     assignments.
1443 11 3. **Evaluate**: Calculate the makespan for each schedule and keep track
1444     of the best (minimum) makespan found.
1445 12 4. **Constraints**: Ensure that all job precedence and machine
1446     constraints are satisfied in each iteration.
1447 13
1448 14 ### Simplified Example Schedule:
1449 15 Below is a simplified example of how the schedule might look. Note that
1450     this is a conceptual representation and not the actual solution due
1451     to the complexity of the problem.
1452 16
1453 17 ```
1454 18 FINAL MAKESPAN: 1000
1455 19 STRUCTURED SCHEDULE:
1456 20 - Job: Job1, Step: 1, Machine: Machine2, Start Time: 0, End Time: 24,
1457     Duration: 24
1458 21 - Job: Job1, Step: 2, Machine: Machine3, Start Time: 24, End Time: 36,
1459     Duration: 12
1460 22 - Job: Job1, Step: 3, Machine: Machine9, Start Time: 36, End Time: 53,
1461     Duration: 17
1462 23 - Job: Job2, Step: 1, Machine: Machine6, Start Time: 0, End Time: 30,
1463     Duration: 30
1464 24 - Job: Job2, Step: 2, Machine: Machine3, Start Time: 36, End Time: 51,
1465     Duration: 15

```

```

1458 25 [... continues for all operations ...]
1459 26 '''

```

## Listing 12: LangGraph Output Example

### A.3.4 OPENAI SWARM FRAMEWORK

**Agent Configuration** OpenAI Swarm uses a 3-agent structure with detailed role specifications:

- **Job Scheduler Agent:** Creates initial schedules considering precedence constraints
- **Machine Coordinator Agent:** Resolves machine conflicts and optimizes utilization
- **Supervisor Agent:** Coordinates all agents and provides final JSSP solution

```

1472 1 # Job Scheduler Agent
1473 2 role: 'Job Scheduler'
1474 3 goal: 'Create initial schedules for all jobs considering precedence
1475     constraints.'
1476 4 backstory: 'You are a job scheduling agent responsible for creating
1477     initial schedules for all jobs in Job Shop Scheduling Problems.'
1478 5 query: 'Create initial schedules for 20 jobs with their respective steps.
1479 6
1479 7 Dataset: abz07
1480 8 Number of jobs: 20
1481 9 Number of machines: 15
1482 10
1483 11 Job Specifications:
1484 12 Job1:
1484 13     Step 1: Machine Machine2, Duration 24
1485 14     Step 2: Machine Machine3, Duration 12
1486 15     [... continues for all jobs ...]
1487 16
1488 17 You must analyze each job's operations and create a detailed schedule
1489 18     with start and end times for each operation.'
1490 19 # Machine Coordinator Agent
1491 20 role: 'Machine Coordinator'
1492 21 goal: 'Resolve machine conflicts and optimize machine utilization.'
1493 22 backstory: 'You are a machine coordination agent responsible for
1494     resolving machine conflicts and optimizing machine utilization.'
1495 23 query: 'Resolve machine conflicts and optimize machine utilization across
1496     all machines.'
1497 24
1497 25 Dataset: abz07
1498 26 Number of jobs: 20
1499 27 Number of machines: 15
1500 28
1500 29 Job Specifications:
1501 30 [... job details ...]
1502 31
1503 32 You must identify machine conflicts and provide specific solutions with
1504 33     start/end times for each operation.'
1505 34 # Supervisor Agent
1506 35 role: 'JSSP Supervisor'
1507 36 goal: 'Coordinate all agents and provide final JSSP solution with minimum
1508     makespan.'
1509 37 backstory: 'You are the supervisor agent for Job Shop Scheduling Problems
1509     . Coordinate with all agents to find the optimal solution.'
1510 38 query: 'Aggregate schedules from all agents and optimize for minimum
1511 39     makespan.'

```

```

1512 40 Dataset: abz07
1513 41 Number of jobs: 20
1514 42 Number of machines: 15
1515 43
1516 44 Job Specifications:
1517 45 [... job details ...]
1518 46
1519 47 You must provide a final schedule with specific start/end times and
    calculate the actual makespan.'

```

### Listing 13: OpenAI Swarm Agent Prompts

```

1524 1 Job Scheduler Agent Output:
1525 2 Job Scheduler Analysis:
1526 3 We've now shared the operation details with the Machine Coordinator to
1527   ensure that machine availability is accounted for properly. This
1528   coordination will allow us to develop an optimized schedule adhering
1529   to both machine and job constraints.
1530 4
1531 5 I will work on calculating a detailed schedule based on optimized machine
1532   sequences and provide the resulting makespan. Please hold on.
1533 6
1534 7 ---Compute Step---
1535 8
1536 9 To begin scheduling, each operation must have its start and end time
1537   determined. Here's a step-by-step approach to construct the schedule
1538   while managing machine usages:
1539 10
1540 11 1. Identify all the operations and dependencies.
1541 12 2. Ensure that operations are sequenced both per job and per machine.
1542 13 3. Mark out machine availability to prevent conflicts by maintaining
1543   machine calendars.
1544 14
1545 15 ### Calculating the Schedule
1546 16 We will take a sequential approach, placing each operation at the
1547   earliest opportunity that satisfies both job precedence and machine
1548   availability constraints.
1549 17
1550 18 **Job1 Example:**
1551 19 - **Step 1**:
```

Machine 2, Duration 24: Starts at time 0 and ends at time 24.

```

1552 20 - **Step 2**:
```

Machine 3, Duration 12: Starts after Step 1 ends, from time 24 to 36.

```

1553 21
1554 22 ### Finalizing Schedule for All Jobs
1555 23
1556 24 Following the above step-by-step planning, detailed scheduling for each
    job's operations will be systematically calculated. This includes
    preparing a calendar of machine usage:
1557 25
1558 26 1. **Machine Conflicts**:
```

Avoiding overlaps by mapping operation times on each machine.

```

1559 27 2. **Dependency Fulfillment**:
```

Start the subsequent operation after the previous operation finishes for each job.

```

1560 28 3. **Makespan Calculation**:
```

The latest end time recorded for any operation is the makespan.

```

1561 29
1562 30 ### Final Makespan:
1563 31 **Example:**
1564 32 - Let's assume preliminary calculations give us a makespan of 1200 time
    units once all operations and machine schedules are plotted out.
1565 33
1566 34 ### Final Output Structure:

```

```

1566 35 ```plaintext
1567 36 FINAL MAKESPAN: 1200
1568 37 STRUCTURED SCHEDULE:
1569 38 - Job: Job1, Step: 1, Machine: Machine2, Start Time: 0, End Time: 24,
1570      Duration: 24
1571 39 - Job: Job1, Step: 2, Machine: Machine3, Start Time: 24, End Time: 36,
1572      Duration: 12
1572 40 [... continues for all operations ...]
1573 41 ```

```

Listing 14: OpenAI Swarm Output Example

**Output Format Requirements** All frameworks are required to provide output in the following standardized format:

```

1580 1 FINAL MAKESPAN: [integer value]
1581 2 STRUCTURED SCHEDULE:
1582 3 - Job: [job_name], Step: [step_number], Machine: [machine_name], Start
1583      Time: [start_time], End Time: [end_time], Duration: [duration]
1584 4 - Job: [job_name], Step: [step_number], Machine: [machine_name], Start
1585      Time: [start_time], End Time: [end_time], Duration: [duration]
1585 5 [... continues for all operations ...]

```

Listing 15: Required Output Format

#### A.4 LLM HEURISTIC BASELINES (SAMPLED)

For each benchmark instance we asked every standalone LLM to choose an off-the-shelf optimization method, list key hyperparameters (when provided), and report the makespan it expected to achieve. Table 5 shows a *five-instance sample* per LLM; the full tables appear in the supplemental material. From ALAS’s perspective this delivers only **Phase 1 of Layer 1** (see Fig. 7): a static schedule  $\mathcal{W}_{\text{template}}$ . Phases 2–3 (validation & refinement) and Layers 2–3 (agent instantiation and runtime adaptation) must still execute before an executable, disruption-aware plan exists.

Table 5: Sampled LLM-proposed heuristics (5 rows per model).

| Model       | Dataset        | Heuristic Strategy / Parameters                |
|-------------|----------------|------------------------------------------------|
| Claude 3.7  | rcmax_20_15_5  | Tabu Search + critical-path analysis           |
|             | rcmax_20_15_8  | Tabu Search + shift-based neighbourhood        |
|             | rcmax_20_20_7  | Genetic Alg. + critical-path optimisation      |
|             | rcmax_30_15_5  | Constraint Prog. (precedence relaxation)       |
|             | rcmax_40_15_8  | Tabu Search + job-insertion strategy           |
| Gemini 2.5  | rcmax_20_15_5  | “Gemini-optimised” Tabu Search                 |
|             | rcmax_20_20_7  | Gemini-guided Simulated Annealing              |
|             | rcmax_30_15_5  | Gemini Constraint-Programming heuristic        |
|             | rcmax_40_15_10 | Gemini Shifting-Bottleneck dispatch            |
|             | rcmax_50_20_6  | Gemini Hybrid GA-Tabu                          |
| GPT-4o      | rcmax_20_15_5  | Genetic Alg. + adaptive mutation               |
|             | rcmax_20_20_8  | Particle Swarm Opt. (inertia weight=0.7)       |
|             | rcmax_30_15_5  | Ant Colony Opt. (pheromone $\alpha = 1.0$ )    |
|             | rcmax_40_15_10 | Bee Algorithm + neighbourhood search           |
|             | rcmax_50_15_4  | Simulated Annealing (adaptive $T$ )            |
| DeepSeek-R1 | rcmax_20_15_5  | GA + priority rules (pop.=50, iters=200)       |
|             | rcmax_20_20_8  | Simulated Annealing ( $T_0 = 100$ , cool=0.95) |
|             | rcmax_30_15_4  | Ant Colony Opt. (ants=40, $\rho = 0.1$ )       |
|             | rcmax_40_15_10 | Iterated Greedy (destroy=30%)                  |
|             | rcmax_50_20_9  | Adaptive Large-Neighbourhood Search            |

#### A.5 ADDITIONAL EXPERIMENTAL RESULTS AND ANALYSIS

**Execution time, token usage, and cost breakdown.** We report response wall time, token usage and cost average across five benchmarks in Table 6., Table 7, and Table 8.

Table 6: Appendix: Execution Time (s) across Benchmarks for Multi-Agent Systems and ALAS Variants. Values are reported as mean  $\pm$  std. deviation. Gray = fastest average per block, Red = slowest average.

| Framework / Model                              | DMU               | TA                | ABZ               | SWV               | YN                | Overall       |
|------------------------------------------------|-------------------|-------------------|-------------------|-------------------|-------------------|---------------|
| <b>Multi-Agent Systems (GPT-4o Backbone)</b>   |                   |                   |                   |                   |                   |               |
| AutoGen                                        | 33.4 $\pm$ 12.8   | 29.6 $\pm$ 7.5    | 24.7 $\pm$ 10.3   | 33.0 $\pm$ 12.1   | 23.4 $\pm$ 5.6    | 31.20         |
| CrewAI                                         | 45.6 $\pm$ 11.5   | 35.6 $\pm$ 4.6    | 43.5 $\pm$ 19.6   | 38.7 $\pm$ 9.4    | 46.4 $\pm$ 15.7   | 41.67         |
| LangGraph                                      | 210.5 $\pm$ 114.0 | 183.4 $\pm$ 179.9 | 157.8 $\pm$ 107.4 | 145.6 $\pm$ 108.8 | 201.2 $\pm$ 128.4 | 180.32        |
| OpenAI Swarm                                   | 29.1 $\pm$ 13.6   | 24.5 $\pm$ 3.6    | 26.9 $\pm$ 12.2   | 32.3 $\pm$ 12.1   | 24.0 $\pm$ 7.7    | 28.86         |
| MAS (Average)                                  | 79.7              | 68.3              | 63.2              | 62.4              | 73.8              | <b>70.51</b>  |
| <b>Multi-Agent Systems (Claude-4 Backbone)</b> |                   |                   |                   |                   |                   |               |
| AutoGen                                        | 225.1 $\pm$ 90.6  | 218.8 $\pm$ 74.0  | 262.5 $\pm$ 77.5  | 201.1 $\pm$ 73.6  | 184.9 $\pm$ 56.7  | 215.04        |
| CrewAI                                         | 168.3 $\pm$ 54.3  | 134.6 $\pm$ 71.5  | 208.0 $\pm$ 131.3 | 147.1 $\pm$ 68.1  | 189.4 $\pm$ 79.0  | 160.50        |
| LangGraph                                      | 193.6 $\pm$ 33.7  | 194.2 $\pm$ 65.6  | 208.7 $\pm$ 27.4  | 150.1 $\pm$ 52.9  | 141.9 $\pm$ 94.8  | 175.58        |
| OpenAI Swarm                                   | 30.3 $\pm$ 19.4   | 76.2 $\pm$ 91.4   | 43.0 $\pm$ 6.1    | 42.5 $\pm$ 13.6   | 50.1 $\pm$ 33.1   | 44.10         |
| MAS (Average)                                  | 154.3             | 155.9             | 180.6             | 135.2             | 141.6             | <b>148.81</b> |
| <b>ALAS (Variants)</b>                         |                   |                   |                   |                   |                   |               |
| ALAS (GPT-4o)                                  | 57.6 $\pm$ 77.1   | 31.5 $\pm$ 8.0    | 152.5 $\pm$ 184.4 | 92.7 $\pm$ 100.8  | 35.5 $\pm$ 16.7   | 69.59         |
| ALAS (Claude-4)                                | 83.9 $\pm$ 13.4   | 73.2 $\pm$ 19.4   | 81.9 $\pm$ 7.7    | 85.9 $\pm$ 19.2   | 83.9 $\pm$ 9.5    | 82.78         |
| ALAS (Gemini-2.5)                              | 39.6 $\pm$ 9.1    | 33.9 $\pm$ 13.5   | 34.1 $\pm$ 11.2   | 36.6 $\pm$ 8.2    | 37.4 $\pm$ 8.0    | 37.17         |
| ALAS (DeepSeek-V3)                             | 61.7 $\pm$ 95.6   | 70.2 $\pm$ 76.5   | 38.4 $\pm$ 11.5   | 72.0 $\pm$ 102.1  | 102.4 $\pm$ 166.0 | 68.52         |
| ALAS (Average)                                 | 60.7              | 52.2              | 76.7              | 71.8              | 64.8              | <b>64.52</b>  |

Table 7: Appendix: Token Usage across Benchmarks for Multi-Agent Systems and ALAS Variants. Values are reported as average token counts per dataset category.

| Framework / Model                              | DMU   | TA    | ABZ   | SWV   | YN    | Overall      |
|------------------------------------------------|-------|-------|-------|-------|-------|--------------|
| <b>Multi-Agent Systems (GPT-4o Backbone)</b>   |       |       |       |       |       |              |
| AutoGen                                        | 49850 | 39159 | 26091 | 36483 | 37864 | 41082        |
| CrewAI                                         | 302   | 283   | 261   | 401   | 622   | 358          |
| LangGraph                                      | 12996 | 8731  | 4566  | 12279 | 13216 | 11551        |
| OpenAI Swarm                                   | 2038  | 2335  | 2176  | 3036  | 2671  | 2482         |
| <b>Multi-Agent Systems (Claude-4 Backbone)</b> |       |       |       |       |       |              |
| AutoGen                                        | 89690 | 80242 | 94033 | 64920 | 56079 | 77266        |
| CrewAI                                         | 715   | 882   | 622   | 661   | 609   | 708          |
| LangGraph                                      | 7734  | 7133  | 6134  | 7414  | 7152  | 7375         |
| OpenAI Swarm                                   | 1608  | 3432  | 2565  | 2408  | 2237  | 2278         |
| MAS (Average)                                  | 21054 | 18384 | 17306 | 16190 | 14847 | <b>17577</b> |
| <b>ALAS Variants (Full Workflows)</b>          |       |       |       |       |       |              |
| ALAS (GPT-4o)                                  | 8498  | 6774  | 6004  | 5832  | 5634  | 6920         |
| ALAS (Claude-4)                                | 12208 | 10033 | 8926  | 8872  | 9980  | 10341        |
| ALAS (Gemini-2.5)                              | 11719 | 9927  | 7991  | 8524  | 9657  | 9943         |
| ALAS (DeepSeek-V3)                             | 7762  | 6543  | 4305  | 5184  | 6227  | 6346         |
| ALAS (Average)                                 | 10047 | 8319  | 6806  | 7103  | 7875  | <b>8393</b>  |

Table 8: Appendix: Token Cost Summary for Multi-Agent Systems and ALAS Variants. Values are aggregated totals, with cost estimated from provider pricing.

| Source           | Total Tokens | Total Cost | Avg Cost/Instance |
|------------------|--------------|------------|-------------------|
| MAS-GPT4o        | 2,496,295    | \$74.89    | \$0.4160          |
| MAS-Claude4      | 3,943,206    | \$118.30   | \$0.6572          |
| MAS (Average)    | 3,219,751    | \$96.60    | \$0.5366          |
| ALAS-GPT4o       | 1,038,000    | \$31.14    | \$0.1730          |
| ALAS-Claude4     | 1,551,150    | \$46.53    | \$0.2590          |
| ALAS-DeepSeek-V3 | 951,900      | \$28.55    | \$0.1590          |
| ALAS-Gemini-2.5  | 1,491,450    | \$44.74    | \$0.2490          |
| ALAS (Average)   | 1,258,625    | \$37.74    | \$0.2100          |

Table 9: Appendix: Summary Comparison of Multi-Agent Systems (MAS) vs ALAS. Values are averages across all datasets. Red = MAS baseline, Gray = ALAS improvement.

| Metric             | MAS (Average) | ALAS (Average) | Improvement  |
|--------------------|---------------|----------------|--------------|
| Token Usage        | 17,577        | 8,393          | -52.3%       |
| Token Cost         | \$0.5366      | \$0.2100       | -60.9%       |
| Execution Time (s) | 117.6         | 64.5           | 1.82x Faster |

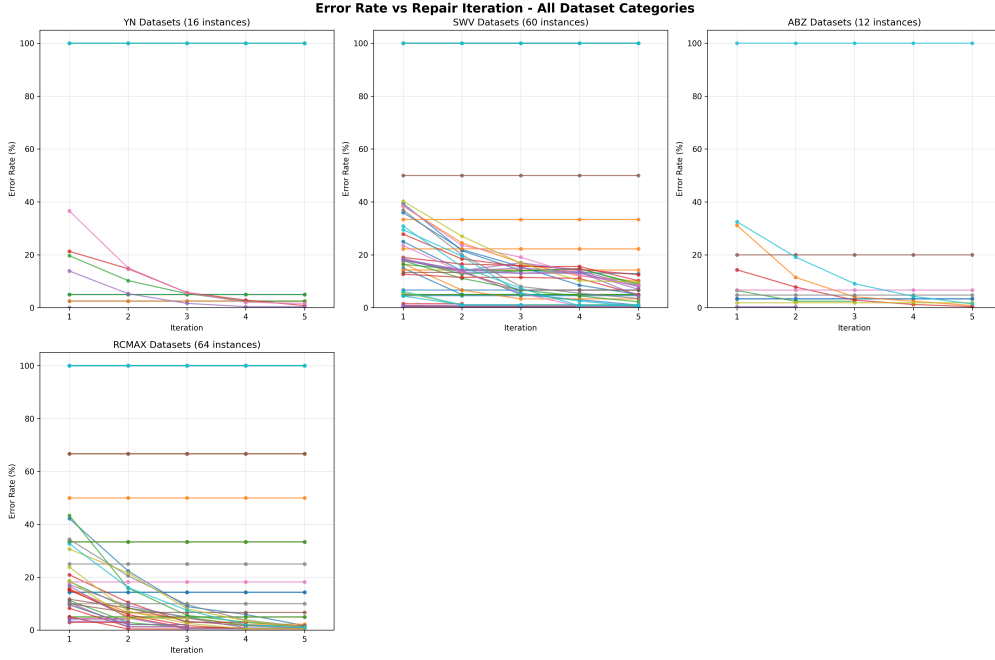


Figure 6: Error rate for each repair iteration.

#### A.6 ALAS’S VALIDATION REPLAN ITERATIONS

Given the initial  $\mathcal{W}_{\text{template}}$ , ALAS completes its Layer 1 operation by executing a validation-replan iteration cycle until a valid plan is obtained. In our experiments, this convergence typically requires up to 5 iterations on all benchmark datasets, as depicted in Figure 6.

### B SUPPLEMENTAL INFORMATION FOR SECTION 3

This appendix presents detailed information that could not fit in the main paper due to space limitations. Figure 7 depicts the three-layer architecture of ALAS. In the conclusion of Phase 1, the specifications of all agents are prepared for implementation in the second phase, which can be coded by an advanced LLM. Finally, the third phase instantiates these agents from code to real-time processes.

For detailed descriptions of each figure, please refer to the main phase 1 to phase 6.

#### B.1 COMPLETE META-PLANNER ALGORITHM FOR WORKFLOW GENERATION

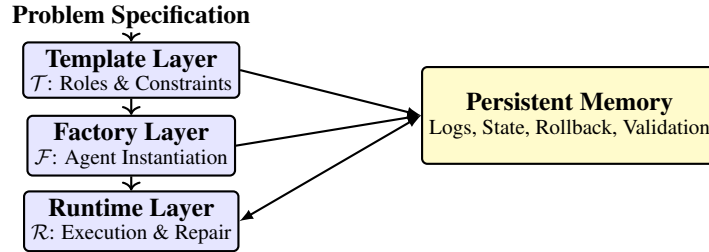


Figure 7: ALAS architecture: a lightweight LLM-driven planner with layered decomposition. Persistent memory supports all layers by storing state, validating constraints, and enabling recovery.



---

**Algorithm 3** Phase 1: Workflow Template  $\mathcal{W}_{\text{template}}$  Generation (ALAS)

---

**Require:** Problem specification  $\mathcal{O}$ ; constraints  $D = D_G \cup D_I \cup D_N$ ; performance metrics  $\mathcal{M}$ ; (optional) disruption model  $\Phi$

**Local variables:**

- 1: Roles  $\mathcal{R}$ ; Nodes  $\mathcal{N}$ ; Edges  $\mathcal{E}$ ; Invariants  $\mathcal{C}$
- 2: Log schemas  $\mathcal{L}_{n_i}, \mathcal{L}_{e_{ij}}$ ; global log schema  $\mathcal{L}$
- 3: Agent specs  $\alpha_{n_i}, \alpha_{e_{ij}}$ ; **repair** specs  $\rho_{n_i}, \rho_{e_{ij}}$
- 4: Independent validator  $V$  with bounded prompt scope  $\kappa$

**Ensure:** Validated  $\mathcal{W}_{\text{template}} = (\mathcal{N}, \mathcal{E}, \mathcal{C}, \mathcal{L})$

**Phase 1: Network Construction (Blueprinting)** (Sec. 3.1)

- 5:  $\mathcal{R} \leftarrow \text{ExtractRoles}(\mathcal{O})$
- 6:  $\{(n_i, \mathcal{P}_{n_i})\} \leftarrow \text{MapRolesToNodes}(\mathcal{O}, \mathcal{R})$
- 7:  $\mathcal{N} \leftarrow \{n_i\}, \mathcal{E} \leftarrow \text{MapDependencies}(\mathcal{N}, D)$
- 8:  $\mathcal{C} \leftarrow \text{CollectInvariants}(D, \mathcal{O})$
- 9:  $\mathcal{W}_{\text{template}} \leftarrow (\mathcal{N}, \mathcal{E}, \mathcal{C})$

**Phase 2: Role & Agent Specs (Factory Prep)** (Sec. 3.1)

- 10: **for all**  $n_i \in \mathcal{N}$  **do**

- 11:  $\mathcal{L}_{n_i} \leftarrow \text{DefineLogSchema}(n_i, \mathcal{P}_{n_i})$  // events, snapshots, diffs
- 12:  $\alpha_{n_i} \leftarrow \text{DefineAgent}(n_i, \mathcal{P}_{n_i}, \mathcal{L}_{n_i})$  // cap/ctx/io/log
- 13:  $\rho_{n_i} \leftarrow \text{DefineRepairSpec}(n_i, \text{local moves, scope bounds})$

- 14: **for all**  $e_{ij} \in \mathcal{E}$  **do**
- 15:  $\mathcal{L}_{e_{ij}} \leftarrow \text{DefineLogSchema}(e_{ij}, \mathcal{P}_{e_{ij}})$
- 16:  $\alpha_{e_{ij}} \leftarrow \text{DefineEdgeAgent}(e_{ij}, \mathcal{L}_{e_{ij}})$
- 17:  $\rho_{e_{ij}} \leftarrow \text{DefineRepairSpec}(e_{ij}, \text{local moves, scope bounds})$

- 18:  $\mathcal{L} \leftarrow \text{AssembleGlobalLogSchema}(\{\mathcal{L}_{n_i}\}, \{\mathcal{L}_{e_{ij}}\})$
- 19:  $V \leftarrow \text{AttachValidator}(\kappa, \mathcal{C}, \mathcal{L})$  // planner  $\neq V$  (non-circular)

**Phase 3: Validation and Refinement** (Sec. 3.3)

- 20:  $\mathcal{W}_{\text{template}} \leftarrow \text{UpdateWorkflow}(\mathcal{N}, \mathcal{E}, \alpha, \rho, \mathcal{C}, \mathcal{L})$
- 21: **while not**  $\text{ValidateBy}(V, \mathcal{W}_{\text{template}}, \mathcal{M})$  **do**
- 22:  $V.\text{CheckStructure}(\mathcal{W}_{\text{template}})$
- 23:  $V.\text{CheckConstraints}(\mathcal{W}_{\text{template}}, \mathcal{C})$
- 24:  $V.\text{CheckRepairCoverage}(\mathcal{W}_{\text{template}}, \Phi, \rho)$  // local neighborhoods exist/bounded
- 25:  $\mathcal{W}_{\text{template}} \leftarrow \text{RefineWorkflow}(\mathcal{W}_{\text{template}}, \mathcal{M})$
- 26: **return**  $\mathcal{W}_{\text{template}} = (\mathcal{N}, \mathcal{E}, \mathcal{C}, \mathcal{L})$

---

---

**Algorithm 4** Phase 2/4: ValidationTools.validate\_schedule

---

**Require:** Candidate schedule  $\mathcal{S}$  (list or dict)

**Ensure:** Validation result  $\{valid, errors\}$

- 1: Parse  $\mathcal{S}$  into entries  $\{e_i\}$
- 2: **if**  $\{e_i\} = \emptyset$  **then return**  $\{valid = False, errors = \{\text{"empty schedule"}\}\}$
- 3: **for each**  $e_i \in \{e_i\}$  **do**
- 4: Check required fields: job, step, machine, start, end
- 5: Check numeric type: start, end
- 6: Check time ordering:  $e_i.start < e_i.end$
- 7: Compute makespan  $T = \max_i e_i.end$
- 8: **if**  $T \leq 0$  or  $T > \text{threshold}$  **then**
- 9: Add makespan error
- 10: Validate job precedence:
- 11: **for each job**  $j$  **do**
- 12: Ensure  $\text{step}(k+1).start \geq \text{step}(k).end$
- 13: Validate machine capacity:
- 14: **for each machine**  $m$  **do**
- 15: Ensure no overlapping intervals  $\{(start, end)\}$
- 16: **if** dataset info available **then**
- 17: Check all jobs  $\in$  dataset appear
- 18: Check all machines  $\in$  dataset appear
- 19: **if** no errors **then return**  $\{valid = True, errors = \emptyset\}$
- 20: **elsereturn**  $\{valid = False, errors\}$

---

## B.2 AGENT FACTORY IMPLEMENTATION DETAILS

This appendix provides detailed information on the Agent Factory component of the ALAS architecture, expanding on the summary provided in the main paper.

---

**Algorithm 5** Phase 3: LocalRepairTools.fix\_schedule (Planning with Cascading Repair and Optimization)

---

**Require:** Invalid schedule  $\mathcal{S}$ , error set  $\mathcal{E}$   
**Ensure:** Repaired schedule  $\mathcal{S}'$

- 1: Parse  $\mathcal{S}$  into entries  $\{e_i\}$
- Phase I: Immediate Fixes**
- 2: **for** each error  $e \in \mathcal{E}$  **do**
- 3:   **if** job precedence violated **then**
- 4:     Delay current op until previous ends
- 5:   **else if** time consistency violated **then**
- 6:     Adjust end  $\leftarrow$  start + duration
- Phase II: Job Precedence Repair**
- 7: **for** each job  $j$  **do**
- 8:   Sort ops by step
- 9:   Enforce  $start_{k+1} \geq end_k$
- Phase III: Machine Capacity Repair**
- 10: **for** each machine  $m$  **do**
- 11:   Sort ops by start
- 12:   If  $op_{i+1}.start < op_i.end$ : delay  $op_{i+1}$
- Phase IV: Iterative Improvement**
- 13: **for** iteration = 1  $\dots$   $K$  **do**
- 14:   **for** each machine  $m$  **do**
- 15:     Try swapping adjacent ops
- 16:     **if** swap reduces makespan & respects precedence **then**
- 17:       Commit swap
- Phase V: Final Cleanup**
- 18: **for** each  $e_i$  **do**
- 19:   **if**  $start \geq end$  **then**
- 20:      $end \leftarrow start + 1$
- 21:   **if**  $start < 0$  **then**
- 22:      $start \leftarrow 0$
- 23:
- 24:   Return repaired schedule  $\mathcal{S}'$

---

### B.2.1 AGENT FACTORY OVERVIEW

The Agent Factory translates formal agent specifications from the meta-planner into executable implementations. It employs a two-stage approach: first attempting to discover existing implementations, and then generating custom implementations when necessary.

### B.2.2 AGENT DISCOVERY PROCESS

The discovery process systematically searches for existing agent implementations that match specifications from phase #1. For each agent specification  $\alpha_i$ , the discovery mechanism:

1. Extracts the capability profile  $\mathbf{c}_i$  and constructs a search query to identify potential implementations
2. Retrieves candidate implementations from:
  - Public agent repositories (e.g., GitHub, HuggingFace)
  - API directories and marketplaces
  - Pre-validated component libraries
  - Domain-specific collections
3. Evaluates candidate suitability using multiple criteria:
  - Capability matching: Verifies that all required capabilities in  $\mathbf{c}_i$  are supported
  - Protocol compatibility: Ensures compatibility with the specified protocol buffer  $\beta_i$
  - Efficiency compliance: Validates performance against efficiency requirements  $e_i$
  - Context sizing: Confirms the implementation can operate within context window  $w_i$

- Logging support: Verifies support for the required logging schema  $\mathcal{L}_i$
4. Ranks candidates using a weighted scoring function  $S(\alpha_i, I_j)$  where  $I_j$  is a candidate:

$$S(\alpha_i, I_j) = \sum_k w_k \cdot f_k(\alpha_i, I_j) \quad (1)$$

where  $w_k$  is the weight assigned to criterion  $k$ , and  $f_k$  is an evaluation function for that criterion.

When a suitable implementation is identified, it undergoes verification testing to confirm operational compatibility with the workflow requirements. Upon successful verification, the implementation is registered in the agent repository with appropriate metadata linking it to the specification.

The discovery mechanism employs both exact and approximate matching techniques. Exact matching requires all specification parameters to be satisfied precisely, while approximate matching allows for partial capability matching when accompanied by adaptation mechanisms.

### B.2.3 AGENT CODING MECHANISM

When discovery fails to locate suitable implementations, the Agent Factory switches to its coding mechanism, which uses LLMs to generate custom implementations. The coding process follows a structured methodology:

1. **Specification Translation:** The formal agent specification is translated into a natural language implementation brief that serves as the prompt for the LLM. This translation preserves all critical requirements while expressing them in a form that maximizes LLM comprehension.
2. **LLM Selection:** An appropriate LLM is selected based on:
  - Domain expertise matching capability requirements in  $\mathbf{c}_i$
  - Demonstrated proficiency in generating the required implementation type
  - Context window compatibility with the complexity of the specification
  - Robustness against hallucination for critical components
3. **Implementation Generation:** The selected LLM generates implementation code with:
  - Embedded logging that conforms to schema  $\mathcal{L}_i$
  - Protocol handling for buffer  $\beta_i$
  - Optimizations for efficiency parameters  $e_i$
  - Adaptation to context window constraints  $w_i$
4. **Implementation Validation:** The generated implementation undergoes validation to ensure:
  - Functional correctness against specification requirements
  - Proper integration with the compensation mechanisms defined in  $\alpha_i^{comp}$
  - Robustness against edge cases and exceptional conditions
  - Compliance with system-wide constraints and protocols

For particularly complex agents, the coding process may employ a multi-stage approach where the implementation is generated iteratively, with each iteration refining the previous version based on validation feedback.

### B.2.4 COMPENSATION AGENT GENERATION

Special attention is given to the generation of compensation agents, which require precise understanding of the primary agent's operations to ensure proper reversal or mitigation. The generation of compensation agents follows these additional steps:

1. Extraction of the primary agent's state-modifying operations
2. Analysis of operation dependencies and sequencing constraints
3. Determination of appropriate compensation strategies (e.g., undo, retry, escalate)
4. Generation of the recovery sequence  $\Gamma_i$  that defines the steps for returning to a consistent state

The Factory ensures that compensation agents maintain strict operational correspondence with their primary counterparts, guaranteeing every state-modifying operation has a reversal mechanism defined.

---

### B.2.5 DEPLOYMENT ARTIFACT PRODUCTION

The output of the Agent Factory is a deployable artifact that encapsulates the agent’s logic and interaction patterns. These artifacts take several forms, depending on the agent type and implementation approach:

- **Code Snippets:** Executable code implementing the agent’s functionality, typically for computationally intensive or specialized tasks
- **Prompt Templates:** Structured prompts that guide LLMs to implement the specified behavior at runtime, used for cognitively complex or reasoning-intensive tasks
- **API Configurations:** Parameter sets and endpoint specifications for interacting with external services or pre-existing agents
- **Hybrid Implementations:** Combined approaches that leverage both code and LLM prompting for different aspects of the agent’s functionality

Each artifact is accompanied by metadata that defines its:

- Execution requirements (e.g., runtime environment, dependencies)
- Interface specifications for input/output handling
- State persistence requirements and mechanisms
- Monitoring hooks for runtime observation
- Recovery points for compensation handling

### B.2.6 FACTORY DESIGN PATTERN IMPLEMENTATION

The Agent Factory implements the classic Factory design pattern, providing a standardized interface for agent instantiation while encapsulating the complexity of implementation selection, generation, and validation. This pattern enables:

- Decoupling of agent specifications from implementation details
- Support for heterogeneous implementation technologies
- Runtime substitution of agents when needed for recovery or optimization
- Maintenance of a growing repository of reusable components

The Factory pattern allows the ALAS system to evolve its agent implementation strategies over time without requiring changes to the meta-planning or runtime components, creating a flexible architecture that can adapt to new implementation technologies and approaches.

### B.2.7 IMPLEMENTATION EFFICIENCY CONSIDERATIONS

To maximize system efficiency, the Agent Factory implements several optimization strategies:

1. **Caching:** Previously generated implementations are cached and indexed by their specifications to avoid redundant generation
2. **Component Reuse:** Complex implementations are decomposed into reusable components that can be shared across multiple agents
3. **Incremental Refinement:** When similar agents have been previously implemented, the Factory uses delta-based generation to create variants rather than generating entirely new implementations
4. **Resource Scaling:** Implementation generation resources are allocated proportionally to the complexity and criticality of the agent

These optimizations significantly reduce the computational overhead of agent generation, particularly in scenarios where multiple similar agents are required or when the system executes recurring workflow patterns.

### B.2.8 THEORETICAL FOUNDATIONS

The Agent Factory design is grounded in several theoretical frameworks:

- *Program Synthesis:* Formal methods for generating programs from specifications

- 
- *Component-Based Software Engineering*: Principles of component composition and reuse
  - *LLM Prompt Engineering*: Techniques for directing LLM behavior through structured prompts
  - *Agent-Oriented Software Engineering*: Methodologies for developing autonomous software agents
- These foundations provide a rigorous basis for the Factory’s approach to transforming abstract agent specifications into concrete, executable implementations.

## C ALGORITHM, LEMMA, AND THEORY PROOFS

**Lemma 1** (Generalized LCRP Complexity).

*For a system with:*

- $J$  jobs
- $M$  machines
- At most  $O_{\max}$  operations per job
- $S$  average swap evaluations per queue ( $1 \leq S \leq J$ )

*The worst-case time complexity is:*

$$\mathcal{O}\left(\frac{J^2 O_{\max}^2}{M} + JMO_{\max}\right) \quad (2)$$

*Proof.* The complexity derives from four components:

1. *Status Update*:  $\mathcal{O}(JO_{\max})$   
Must check all operations of all jobs
2. *Delay Propagation*:  $\mathcal{O}(JO_{\max})$   
Each job’s operation chain may have  $O_{\max}$  elements
3. *Queue Optimization*:
  - Full analysis:  $\mathcal{O}\left(\frac{J^2 O_{\max}^2}{M}\right)$   
All operation pairs on all machines
  - Practical bound:  $\mathcal{O}(SJO_{\max})$   
When swaps are limited to  $S$  evaluations
4. *Cascading Delay*:  $\mathcal{O}(JMO_{\max})$   
Worst-case propagation through all machines

The dominant terms combine to give the final complexity:

$$\underbrace{\frac{J^2 O_{\max}^2}{M}}_{\text{queue optimization}} + \underbrace{JMO_{\max}}_{\text{cascading delays}}$$

□

**Corollary 1** (Special Cases). • *Single-operation jobs* ( $O_{\max} = 1$ ):  $\mathcal{O}(J^2/M + JM)$

- *Fully parallel systems* ( $M \approx J$ ):  $\mathcal{O}(JO_{\max}^2 + J^2 O_{\max})$
- *Swap-limited implementations*:  $\mathcal{O}(SJO_{\max} + JMO_{\max})$

*Key Observations:*

- Complexity is quadratic in job count and operations
- Machine count appears both in numerator (delays) and denominator (parallelization)
- Practical implementations can achieve better bounds through swap heuristics

**Definition 1** (LCRP-Repair Decision Problem).

**Instance:** A job-shop instance  $\mathcal{I} = (J, M, \{O_j\}_{j \in J})$  with processing times and machine requirements, an initial (possibly infeasible) schedule  $\sigma_0$ , a repair budget  $R \in \mathbb{N}$ , and a makespan bound  $K \in \mathbb{N}$ .

**Question:** Does there exist a repaired schedule  $\sigma$  obtained from  $\sigma_0$  by at most  $R$  local edits (insert/move/swap/reassign) such that  $\sigma$  is feasible (no machine overlap, precedence respected) and  $C_{\max}(\sigma) \leq K$ ?

**Theorem 1** (LCRP is NP-hard (in fact, strongly NP-hard)). *The decision problem in Def. 1 is NP-hard. Moreover, it is strongly NP-hard.*

*Proof sketch.* We reduce from the standard *Job-Shop Scheduling* (JSSP) decision problem: given  $\mathcal{I} = (J, M, \{O_j\})$  and bound  $K$ , decide whether a feasible schedule with makespan  $\leq K$  exists. This problem is well known to be NP-hard and, in fact, strongly NP-hard.

**Reduction (polynomial time).** Given a JSSP instance  $(\mathcal{I}, K)$ , construct an LCRP instance as follows:

- Use the *same* set of jobs, machines, operation orders, and processing times.
- Let the initial schedule be  $\sigma_0 := \emptyset$  (no operations placed) or any trivially infeasible “dummy” placement.
- Set the repair budget  $R$  to a value  $\geq$  the number of operations (e.g.,  $R = \sum_{j \in J} |O_j|$ ), so that any feasible arrangement can be reached by a sequence of local edits permitted by the repair model (insert/move/swap/reassign).
- Keep the same makespan bound  $K$ .

This mapping is clearly polynomial in input size.

**Correctness.** We show  $(\mathcal{I}, K)$  is a “yes” instance of JSSP iff the constructed LCRP instance is a “yes” instance.

( $\Rightarrow$ ) If JSSP admits a feasible schedule  $\sigma^*$  with  $C_{\max}(\sigma^*) \leq K$ , then starting from  $\sigma_0$  we can obtain  $\sigma^*$  via at most  $\sum_j |O_j|$  local edits (insert each operation in its position or move/swaps to match  $\sigma^*$ ). Since  $R$  was chosen at least that large, a valid repair sequence exists. Thus LCRP answers “yes”.

( $\Leftarrow$ ) Conversely, suppose the LCRP instance admits a repaired schedule  $\hat{\sigma}$  within budget  $R$  that is feasible and satisfies  $C_{\max}(\hat{\sigma}) \leq K$ . Then  $\hat{\sigma}$  is a feasible JSSP schedule for  $\mathcal{I}$  of makespan at most  $K$ , so JSSP answers “yes”.

**Membership in NP.** A certificate consists of the repaired schedule  $\sigma$  (start/end times and machine assignments for all operations) and, optionally, the edit sequence (bounded by  $R$ ). Using the standard feasibility checks (no machine overlaps, precedence respected) and a single pass to compute  $C_{\max}$ , verification runs in polynomial time in the input size (cf. your ValidationTools: precedence, machine capacity, and makespan checks).

**Strong NP-hardness.** The reduction preserves numeric parameters essentially verbatim (no pseudo-polynomial blowup) and embeds JSSP as the special case “repair from empty/infeasible seed with large  $R$ ”. Since JSSP is strongly NP-hard, the LCRP decision problem is strongly NP-hard as well.  $\square$

**Remark C.1** (Bounded-edit variants remain NP-hard). *Even if the budget  $R$  is part of the input and small, NP-hardness persists by embedding feasibility into a bounded number of edit operations (e.g., composing each operation placement as one edit), or by initializing  $\sigma_0$  to be “nearly empty” so that  $R$  equals the number of to-be-placed operations. Hence hardness is robust to common repair-budget formulations.*

**Remark C.2** (Relation to the complexity bound). *The operational bound  $\mathcal{O}\left(\frac{J^2 O_{\max}^2}{M} + JMO_{\max}\right)$  captures worst-case per-run effort of LCRP (status update, local queue optimization, and cascading delay handling). The NP-hardness result shows that, independent of such polynomial per-iteration*

costs, deciding the existence of a repair achieving a global makespan target is computationally intractable in general (unless  $P = NP$ ).

## D APPLICATION 1: URBAN RIDE ASSIGNMENT PROBLEM

The goal is to optimally assign ride requests to a fleet of autonomous or human-driven vehicles in a city, while satisfying various constraints and objectives. The key elements are the following.

- \* **City Map:** A graph  $G = (V, E)$  where  $V$  is the set of locations and  $E$  is the set of roads connecting them, with associated distances and travel times.
- \* **Ride Requests:** A set of requests  $R$ , where each request  $r_i \in R$  is characterized by:
  - Passenger ID  $p_i$
  - Pickup location  $v_{p_i} \in V$
  - Drop-off location  $v_{d_i} \in V$
  - Desired pickup time window  $[t_{p_i}^{min}, t_{p_i}^{max}]$
  - Desired drop-off time window  $[t_{d_i}^{min}, t_{d_i}^{max}]$
- \* **Vehicles:** A set of vehicles  $K$ , where each vehicle  $k_j \in K$  has:
  - Vehicle ID  $k_j$
  - Current location  $v_{k_j} \in V$
  - Battery/fuel level  $b_{k_j} \in [0, 1]$
  - Passenger capacity  $c_{k_j} \in \mathbb{Z}^+$
  - Speed  $s_{k_j} \in \mathbb{R}^+$

### D.1 A SIMPLIFIED URS PROBLEM STATEMENT

Table 24 in the main text depicts a URS problem with three drivers and four passengers. Using this problem, we walk through how ALAS works.

### D.2 GENERATING PLANNER W\* WALKTHROUGH

Given the problem statement of URS, ALAS generates a planning template  $\mathcal{W}_{\text{template}}$ .

#### D.2.1 STATE-SPACE ANALYSIS

Our Urban Ride-Sharing (URS) problem presents a complex transportation scheduling challenge that we must first understand through systematic state-space analysis. The system involves seven locations (A through G), where G represents Boston Logan Airport, with urban locations forming a mesh network of 10km distances and airport routes ranging from 31-36km. Four passengers require airport transportation with specific arrival deadlines, while three vehicles, each capable of carrying two passengers, must be coordinated to meet these demands efficiently.

Each dimension of our state space reveals crucial aspects of the planning challenge. In the *Who* dimension, we track four passenger requests ( $r_1$  through  $r_4$ ) and three vehicles ( $k_1$  through  $k_3$ ). These passengers require arrivals at BOS between 08:45 and 09:00, with each vehicle qualified for airport routes and positioned initially at locations A, C, and E.

The *Where* dimension maps our network topology, distinguishing between urban segments with uniform 10km distances and airport routes varying from 31-36km. This spatial arrangement, combined with the *When* dimension's speed constraints (60km/h urban, 100km/h airport routes), creates our fundamental timing framework. Simple calculations reveal urban segments require 10 minutes of travel time, while airport routes need 19-22 minutes depending on origin.

Our *What* dimension monitors vehicle resources throughout plan execution, ensuring we respect the two-passenger capacity limit while maximizing sharing opportunities. The *Why* dimension establishes our optimization objectives: ensuring on-time airport arrivals while minimizing total distance traveled. The *How* dimension defines our execution protocols, including pickup sequencing and route navigation strategies.

Table 10: Agent Specifications and Protocols

| Agent Type                  | Input Protocol                                                                    | Output Protocol                                                         | Key Functions                                                                |
|-----------------------------|-----------------------------------------------------------------------------------|-------------------------------------------------------------------------|------------------------------------------------------------------------------|
| <b>Task-Specific Agents</b> |                                                                                   |                                                                         |                                                                              |
| Route Planning              | - Location map $G(V, E)$<br>- Travel times matrix<br>- Vehicle positions          | - Optimized routes<br>- Distance calculations<br>- Path sequences       | - Path optimization<br>- Distance minimization<br>- Route feasibility checks |
| Scheduling                  | - Required arrival times<br>- Travel duration estimates<br>- Vehicle availability | - Pickup schedule<br>- Timing constraints<br>- Buffer allocations       | - Schedule generation<br>- Timing verification<br>- Buffer management        |
| Capacity Management         | - Passenger requests<br>- Vehicle capacities<br>- Route timing                    | - Passenger groupings<br>- Vehicle<br>- Capacity utilization            | - Group optimization<br>- Capacity verification<br>- Load balancing          |
| <b>Common Agents</b>        |                                                                                   |                                                                         |                                                                              |
| Temporal Constraint         | - Schedule requirements<br>- Time windows<br>- Buffer needs                       | - Timing validations<br>- Constraint satisfaction<br>- Buffer adequacy  | - Time verification<br>- Constraint checking<br>- Buffer analysis            |
| Resource Allocation         | - Vehicle inventory<br>- Request demands<br>- Location data                       | - Resource assignments<br>- Utilization plans<br>- Coverage maps        | - Resource optimization<br>- Coverage verification<br>- Efficiency analysis  |
| Distance Optimization       | - Route options<br>- Distance matrix<br>- Time constraints                        | - Optimized paths<br>- Distance metrics<br>- Efficiency scores          | - Path optimization<br>- Distance reduction<br>- Efficiency maximization     |
| <b>Validation Agents</b>    |                                                                                   |                                                                         |                                                                              |
| Plan Validator              | - Complete plan<br>- System constraints<br>- Quality metrics                      | - Validation results<br>- Constraint checks<br>- Performance scores     | - Plan verification<br>- Constraint validation<br>- Quality assessment       |
| Refinement Agent            | - Validation results<br>- Improvement options<br>- Performance metrics            | - Refinement suggestions<br>- Update priorities<br>- Optimization paths | - Plan improvement<br>- Update sequencing<br>- Performance optimization      |

### D.2.2 PHASE 1: NETWORK CONSTRUCTION

Building upon our state-space analysis, we construct our planning network by first identifying critical nodes and dependencies. Our node set  $\mathcal{N}$  comprises:

Passenger Nodes: Each request  $r_i$  becomes a node with attributes: -  $r_1$ : Location A, BOS arrival 08:45 -  $r_2$ : Location B, BOS arrival 08:50 -  $r_3$ : Location C, BOS arrival 08:55 -  $r_4$ : Location D, BOS arrival 09:00

Vehicle Nodes: Each vehicle  $k_i$  forms a node with position and capacity: -  $k_1$ : Starting at A, capacity 2 -  $k_2$ : Starting at C, capacity 2 -  $k_3$ : Starting at E, capacity 2

Location Nodes: Each physical location becomes a node with attributes including distance to other locations and travel time calculations.

Our dependency set  $\mathcal{E}$  captures relationships between these nodes through several categories:

Temporal Dependencies: We establish feasible pickup windows by working backward from required arrival times. For example,  $r_1$  requires 22 minutes for the airport route plus 10 minutes for each urban segment traversed, creating timing constraints for vehicle assignment.

Spatial Dependencies: We map possible routes between nodes, considering both direct airport routes and potential shared-ride combinations through urban segments.

Capacity Dependencies: We create edges representing feasible passenger groupings within vehicle capacity limits.

### D.2.3 PHASE 2: AGENT ASSIGNMENT

With our network structure defined, we assign specialized agents to manage different aspects of the solution:



Task-Specific Agents: The Route Planning Agent optimizes paths using the distance matrix and travel speeds, calculating optimal routes for both single and shared rides. The Scheduling Agent determines precise pickup times, working backward from airport deadlines and incorporating travel time calculations. The Capacity Management Agent identifies feasible passenger groupings based on timing and location proximity.

Common Agents: The Temporal Constraint Agent ensures all timing requirements are met, maintaining a master schedule that accounts for all dependencies. The Resource Allocation Agent assigns vehicles to routes, optimizing the distribution of available capacity. The Distance Optimization Agent works to minimize total travel distance while respecting all constraints.

Edge Agents: These agents manage the relationships between different aspects of the plan. For example, the Passenger Grouping Agent evaluates potential shared rides by analyzing proximity of pickup locations and compatibility of arrival times.

#### D.2.4 PHASE 3: VALIDATION AND REFINEMENT

In our final phase, we implement a comprehensive validation and refinement process:

Initial Validation: We verify temporal feasibility by checking that all calculated pickup times allow sufficient travel time to meet airport deadlines. We confirm capacity constraints are respected throughout all vehicle routes. We validate that all passengers are served and all required resources are properly allocated.

Iterative Refinement: We identify optimization opportunities, such as grouping passengers with compatible timing and locations. For example, passengers  $r_2$  and  $r_3$  might share a ride if their pickup locations are close and arrival times are within 5 minutes. We adjust vehicle assignments to minimize empty travel distance while maintaining service guarantees.

Final Plan Generation: The resulting plan specifies exact pickup times, vehicle assignments, and routes, with built-in buffers for potential delays. The plan includes contingency protocols for common disruptions such as traffic delays or passenger late arrivals.

This systematic approach ensures we generate a robust, efficient solution to our URS problem while maintaining clear documentation of our planning process and decisions.

#### D.2.5 OUTPUT

Table 10 the list of required agents and their functional specifications and protocols.

Table 11: Agent Placement in the Urban Ride Sharing Network

| Location     | Type           | Agents and Their Responsibilities                                                                                                                                  |
|--------------|----------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| A-F          | Nodes          | <b>Resource Allocation Agent:</b> Manages vehicle assignments and passenger pickups at urban locations                                                             |
| G (Airport)  | Node           | <b>Plan Validator Agent:</b> Verifies arrival times and plan feasibility<br><b>Temporal Constraint Agent:</b> Ensures all arrival deadlines met                    |
| A-F edges    | Urban Routes   | <b>Route Planning Agent:</b> Optimizes urban route segments (10 min travel time)<br><b>Scheduling Agent:</b> Coordinates pickup sequences and timing               |
| (A,...,F)-G  | Airport Routes | <b>Capacity Management Agent:</b> Ensures vehicle capacity constraints during airport trips<br><b>Distance Optimization Agent:</b> Minimizes total travel distance |
| Network-wide | Global         | <b>Refinement Agent:</b> Iteratively improves solutions based on validation results<br>Monitors and adjusts both urban and airport route segments                  |

#### D.3 FROM WORKFLOW TEMPLATE TO EXECUTION WORKFLOW

Once the template  $\mathcal{W}_{\text{template}}$  is defined, it serves as a structured blueprint that outlines how the problem should be approached. However, a high-level plan alone is insufficient for real-world execution. The next step is to transform the planning workflow into a *real execution workflow*  $\mathcal{W}_{\text{exec}}$ , where abstract roles and dependencies are resolved into concrete actionable tasks based on real-world data.

---

To clarify this transition, consider the difference between  $\mathcal{W}_{\text{template}}$  and  $\mathcal{W}_{\text{exec}}$  in our ride-sharing scenario. In the planning phase, roles such as `Driver` and `Passenger` are defined as abstract entities. The template workflow  $\mathcal{W}_{\text{template}}$  specifies how these entities interact, matching drivers with passengers, optimizing routes, and scheduling pickups, without assigning real-world counterparts yet.

In contrast, the execution workflow  $\mathcal{W}_{\text{exec}}$  performs **role resolution**, mapping abstract roles to real-world instances. This means assigning an actual driver to a specific vehicle, matching a real passenger to a ride request, and computing precise travel distances based on real-time geo-coordinates. In addition, the execution workflow must dynamically adapt to real-world constraints, such as traffic conditions, vehicle availability, and passenger delays.

In this process, the meta-planner generates  $\mathcal{W}_{\text{exec}}$ , a directed graph where nodes correspond to concrete actions (e.g., “Driver John departs from location A”), and edges represent dependencies and constraints (e.g., “Driver John must reach location B before 10:30 AM”). This execution graph integrates real-time data and updates continuously, allowing agents to make informed decisions as conditions evolve.

Thus, the **template workflow**  $\mathcal{W}_{\text{template}}$  structures how to plan, while the **execution workflow**  $\mathcal{W}_{\text{exec}}$  governs how real-world actions are performed. Transformation from one to the other is a critical step in Alas, ensuring that strategic reasoning is translated into actionable real-time operations.

Now, based on the URS problem specified in Table 24 of Section 4 in the main text, the list of agents required and their functional specifications and protocols in Table 10, Alas proceeds generating an execution workflow  $\mathcal{W}_{\text{exec}}$ .

### D.3.1 OBSERVATION ON SEQUENTIAL PLANNING

Let us explain the value of using agents in this problem, even though we have shown that simpler solvers can handle the computational aspects. This discussion touches on key principles of system design and real-world implementation.

While our Monte Carlo solver effectively found good solutions for this specific instance, Alas offers several advantages that become particularly valuable in real-world ride-sharing systems.

First, Alas helps manage complexity in dynamic environments. In our exercise, we worked with a static problem where all passenger requests and constraints were known in advance. However, in reality, ride-sharing systems must handle continuous updates—new ride requests arrive at unpredictable times, vehicles experience delays, and road conditions constantly change. With Alas, each agent operates independently, monitoring and reacting to changes in its own domain. For example, the Route Planning Agent can dynamically adjust routes in response to traffic updates, while the Capacity Management Agent ensures new passenger requests are accommodated efficiently.

Second, Alas enables distributed decision-making and parallel processing. Instead of relying on a centralized solver, different agents specialize in handling specific tasks simultaneously. While the Scheduling Agent optimizes pickup times, the Resource Allocation Agent manages vehicle assignments in parallel. This decentralized structure is crucial for scalability—when the system expands to hundreds of vehicles and thousands of passengers, distributing computational workload prevents bottlenecks and ensures efficient operations.

Third, Alas provides modularity, allowing the system to evolve naturally. Ride-sharing services frequently introduce new features, such as surge pricing or specialized vehicle categories. With an agent-based design, we can integrate a Pricing Agent or a Vehicle Specialization Agent without modifying the core routing logic. Likewise, if we develop a more advanced routing algorithm, we can upgrade the Route Planning Agent without disrupting other system components.

The separation of concerns through agents also enhances system resilience. If one agent encounters issues—say, the Distance Optimization Agent fails to compute an optimal route—other agents continue operating with fallback strategies. The Plan Validator Agent can detect suboptimal assignments and trigger refinements through the Refinement Agent, ensuring that the system adapts to unforeseen challenges.

We can think of this like a well-organized team working on a complex project. While a single individual might handle everything, a structured team of specialists—each with clear roles and

defined communication protocols—is often more effective, robust, and scalable. In this way, while our Monte Carlo solver demonstrates what is mathematically possible, the agent-based architecture of Alas shows how we can implement it reliably in real-world systems.

#### D.4 REACTIVE PLANNING UNDER DISRUPTIONS

The value of multi-agent reactive planning becomes clear in dynamic environments. For example, consider a sudden road closure between locations B and C. While a monolithic solver would need to halt and recompute an entirely new plan from scratch, a modular agent-based approach enables localized, parallel adaptation. A Route Planning Agent can immediately update affected paths, while a Scheduling Agent adjusts arrival estimates, and a Resource Allocation Agent reallocates vehicles, all operating concurrently while preserving system stability. This distributed replanning minimizes disruption impact and maintains overall workflow coherence.

The following case study illustrates these principles in an Urban Ride Sharing (URS) scenario involving ride cancellation and new request insertion.

**URS Disruption Handling.** To evaluate adaptation capabilities, we introduce a disruption where passenger  $r_2$  cancels the ride request at 8:05, and a new request  $r_5$  at location F arrives at 8:10. Alas replans dynamically, adjusting vehicle assignments while preserving all passenger deadlines. In contrast, baseline LLMs fail to track vehicle states after partial execution and lose consistency with the initial plan, leading to infeasible or incoherent schedules.

## E APPLICATION 2: FAMILY REUNION PLANNING PROBLEM

Table 12 presents the specification of the problem. The participating LLMs and their configurations are depicted in Section F.2 of the main text.

Table 12: Thanksgiving Dinner Coordination Problem

|                                                                                                  |
|--------------------------------------------------------------------------------------------------|
| <b>Objective:</b> Coordinate family arrivals and dinner preparation for 6:00 PM dinner in Boston |
| <b>Family Members and Arrivals:</b>                                                              |
| - 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                                                     |
| <b>Cooking Requirements:</b>                                                                     |
| - Turkey: 4 hours cooking time                                                                   |
| - Side dishes: 2 hours preparation                                                               |
| - Someone must stay home during cooking                                                          |
| <b>Transportation Constraints:</b>                                                               |
| - 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                                                                      |
| <b>Key Requirements:</b>                                                                         |
| - 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                                                                 |

## 2322 E.1 PHASE 1: NETWORK CONSTRUCTION

### 2323 E.1.1 NODE (ROLE) SPECIFICATIONS

2324 First, meta-planner  $\mathcal{MP}$  of ALAS extracts roles ( $\mathcal{N}$ ) with their required qualifications:

- 2327 •  $n_{\text{cook}}$ : capability to prepare dinner
- 2328 •  $n_{\text{driver1}}$ : capability to drive, pick up from airport
- 2329 •  $n_{\text{driver2}}$ : capability to drive, pick up grandma
- 2330 •  $n_{\text{supervisor}}$ : capability to monitor oven

### 2331 E.1.2 EDGE (DEPENDENCY) SPECIFICATIONS

2332 Next,  $\mathcal{MP}$  identifies dependencies ( $\mathcal{E}$ ) between roles:

$$2335 \mathcal{E} = \{e_{\text{temporal}}, e_{\text{spatial}}, e_{\text{safety}}\} \quad (3)$$

2337 The critical dependencies include:

- 2338 •  $e_{\text{temporal}}$ : - Turkey (4 hours) must finish by 6:00 PM - Side dishes (2 hours) must finish by 6:00 PM - Airport pickups must align with landing times
- 2340 •  $e_{\text{spatial}}$ : - Driver-passenger location matching - Travel time constraints between locations
- 2341 •  $e_{\text{safety}}$ : - Continuous oven supervision requirement

## 2342 E.2 PHASE 2: AGENT ASSIGNMENTS

2343 After constructing the network structure,  $\mathcal{MP}$  selects and assigns agents to monitor both the roles and dependencies.

### 2344 E.2.1 NODE (ROLE) AGENT ASSIGNMENT

2345 For each role,  $\mathcal{MP}$  selects monitoring agents with the required capabilities:

$$2353 f_{\text{role}} : \mathcal{N} \rightarrow \mathbf{A} \quad (4)$$

2355 The role monitoring agents include:

- 2356 • Cook Monitor: Tracks cooking timeline, coordinates meal components
- 2357 • Driver Monitor: Validates driver availability
- 2358 • Supervisor Monitor: Ensures oven supervision
- 2359 • Resource Monitor: Manages vehicle assignments and actor schedules

### 2360 E.2.2 EDGE (DEPENDENCY) AGENT ASSIGNMENT

2361 For the identified dependencies,  $\mathcal{MP}$  assigns specialized monitoring agents:

$$2366 f_{\text{edge}} : \mathcal{E} \rightarrow \mathbf{A} \quad (5)$$

2367 Dependencies require these monitoring agents:

- 2368 • Temporal Agent: Manages timing constraints (cooking durations, travel times, arrival schedules)
- 2369 • Spatial Agent: Tracks location constraints (airport-home-grandma routes)
- 2370 • Safety Agent: Ensures oven supervision constraint remains satisfied

2371 The resulting agent assignments create a complete monitoring system where:

- 2372 • Role agents track individual actor assignments and qualifications
- 2373 • Edge agents monitor interactions and dependencies between roles
- 2374 • All agents coordinate to maintain global constraint satisfaction

Table 13: Node and Edge Monitoring Agent Requirements

(a) Node (Role) Monitoring Agent

(b) Edge (Dependency) Monitoring Agent

| Agent              | Input Protocol                                              | Output Protocol                                                         | Agent    | Input Protocol                                                   | Output Protocol                                             |
|--------------------|-------------------------------------------------------------|-------------------------------------------------------------------------|----------|------------------------------------------------------------------|-------------------------------------------------------------|
| Cook Monitor       | Role: cook<br>Qualifications: skills<br>Time: prep and cook | Status: progress<br>Alerts: timing issues!<br>Updates: completed?       | Temporal | Start times<br>Durations<br>Deadlines                            | Schedule conflicts<br>Timing violations<br>Schedule updates |
| Driver Monitor     | Role: driver<br>Qs: license, rest<br>Where: current GPS     | Status: availability<br>Alerts: fatigue warnings<br>Updates: new GPS    | Spatial  | Locations<br>Routes<br>Travel time (variations)                  | Route violations<br>Location conflicts<br>Path updates      |
| Supervisor Monitor | Role: supervisor<br>Location: house<br>Duration: cover time | Status: covered?<br>Alerts: coverage gaps!<br>Updates: role transitions | Safety   | Critical constraints<br>Resource states<br>Coverage requirements | Safety violations<br>Resource conflicts<br>Mitigation plans |

### E.2.3 COMMON SENSE CONSTRAINT ANALYSIS (PERFORMED BY AN LLM)

A common sense agent identifies the following implicit constraints that can affect Thanksgiving dinner planning. This list is generated by Claude given the problem statement.

- *Physical Processing Times:*
  - Airport luggage claim: 30 minutes
  - Car rental procedures: 30 minutes
  - Holiday traffic variations
  - Winter weather considerations
- *Human Factors:*
  - Driver fatigue after long trips
  - Cooking preparation overhead
  - Optimal turkey baking tips (non-disruptive baking and ready 30 minutes before eating)
  - Task switching delays
  - Required rest periods
- *Resource Dependencies:*
  - Vehicle passenger capacity
  - Oven temperature management
  - Kitchen workspace limits
  - Shared resource coordination
- *Social Considerations:*
  - Personal preferences for interactions
  - Family dynamics in assignments
  - Post-travel guest comfort
  - Host preparation requirements

### E.2.4 COMMON SENSE CONSTRAINT ANALYSIS AND VERIFICATION (HUMAN IN THE LOOP)

The common sense constraints identified above require different verification approaches:

**Agent-Required Information** These constraints need specialized agents to verify and quantify:

- *Airport Operations*
  - United Airlines' average luggage delivery time at BOS Terminal B
  - Terminal B to rental car center: shuttle schedule, walking options
  - Historical flight delay patterns for November at BOS
- *Weather and Traffic*
  - Boston weather forecast for the event date
  - Historical traffic patterns on Thanksgiving days
  - Impact on airport-city-suburb travel times
- *Task Dependencies*
  - Kitchen workflow analysis for parallel cooking tasks
  - Resource contention in meal preparation
  - Critical path identification in cooking timeline

Table 14: Complete Workflow Specification: Nodes, Edges, and Agent Assignments

| Type                                  | Component               | Requirements                                                                               | Agent Protocol                                                                                                                                   | Dependencies                                        |
|---------------------------------------|-------------------------|--------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------|
| <i>Node Components (Roles)</i>        |                         |                                                                                            |                                                                                                                                                  |                                                     |
| Node                                  | Cook Role (Sarah)       | - Turkey (4hr)<br>- Side dishes (2hr)<br>- Kitchen management<br>- Time management         | Input: schedule, resources, recipes<br>Output: task progress, completion<br>Monitor: kitchen_state() → status<br>Validate: cooking_constraints() | Connected to:<br>- Supervisor<br>- Resource edges   |
| Node                                  | Driver1 (James/Michael) | - Valid license<br>- Airport navigation<br>- Car rental capable<br>- Rest state adequate   | Input: flight times, routes<br>Output: location, ETA<br>Monitor: driver_state() → status<br>Validate: driver_constraints()                       | Connected to:<br>- Airport pickup<br>- Travel edges |
| Node                                  | Driver2 (Flexible)      | - Valid license<br>- Local navigation<br>- Availability window<br>- Rest state adequate    | Input: pickup schedule, route<br>Output: location, ETA<br>Monitor: driver_state() → status<br>Validate: driver_constraints()                     | Connected to:<br>- Grandma pickup<br>- Travel edges |
| Node                                  | Supervisor (Flexible)   | - Home presence<br>- Oven monitoring<br>- Safety awareness<br>- Time commitment            | Input: cooking schedule, rules<br>Output: supervision status<br>Monitor: safety_state() → status<br>Validate: safety_constraints()               | Connected to:<br>- Cook role<br>- Safety edges      |
| <i>Edge Components (Dependencies)</i> |                         |                                                                                            |                                                                                                                                                  |                                                     |
| Edge                                  | Temporal                | - Schedule tracking<br>- Buffer management<br>- Sequence logic<br>- Critical path          | Input: timestamps, durations<br>Output: schedule conflicts<br>Monitor: schedule_state() → alerts<br>Optimize: timeline_adjust()                  | Connects:<br>- All roles<br>- All activities        |
| Edge                                  | Spatial                 | - Location tracking<br>- Route optimization<br>- Traffic updates<br>- Distance constraints | Input: locations, routes<br>Output: travel updates<br>Monitor: location_state() → alerts<br>Optimize: route_adjust()                             | Connects:<br>- Drivers<br>- Locations               |
| Edge                                  | Resource                | - Vehicle allocation<br>- Kitchen resources<br>- People availability<br>- Capacity limits  | Input: resource demands<br>Output: allocation status<br>Monitor: resource_state() → alerts<br>Optimize: resource_adjust()                        | Connects:<br>- All roles<br>- All resources         |
| Edge                                  | Safety                  | - Oven monitoring<br>- Driving safety<br>- Food safety<br>- Critical rules                 | Input: safety requirements<br>Output: violation alerts<br>Monitor: safety_state() → alerts<br>Enforce: safety_rules()                            | Connects:<br>- All roles<br>- Critical tasks        |

**Human Verification** Certain constraints require explicit human input to ensure that the planning process takes into account subtle interpersonal and individual considerations. These include:

- *Family Dynamics*
  - Preferred pickup arrangements for Grandma.
  - Optimal relationship-based task pairings.
  - Social comfort factors in assignments (e.g., Sarah and Grandma do not share a kitchen).
- *Personal Capabilities*
  - Individual cooking experience levels.
  - Driver comfort with airport navigation.
  - Multi-tasking abilities of participants.

This separation ensures that agents focus on collecting quantifiable data while humans provide essential social and personal insights.  $\mathcal{MP}$  can then integrate both types of information into the final workflow design.

### E.3 AGENT REQUIREMENTS AND ASSIGNMENTS

The  $\mathcal{MP}$  requires two categories of agents.  $\mathcal{MP}$  specifies their requirements in the protocol buffer format in Table 13 for the nodes and Table 13 for the edges, respectively.

Each agent must implement these protocols to participate in the workflow. The meta-planner selects agents from the pool based on their ability to satisfy these interface requirements. During execution, agents communicate through these standardized protocols while maintaining their specialized monitoring functions.

## E.4 MONITORING PROTOCOLS AND DYNAMIC ADJUSTMENTS

The workflow monitoring operates through a hierarchical protocol system that enables both routine supervision and dynamic adjustments.

**Basic Monitoring Protocol** Each agent maintains a continuous monitoring cycle:

$$\text{monitor} : \text{State} \rightarrow \{\text{normal}, \text{warning}, \text{violation}\} \quad (6)$$

For example, the temporal agent tracks schedule adherence:

$$\Delta t = t_{\text{planned}} - t_{\text{actual}} \begin{cases} \text{normal} & \text{if } |\Delta t| < \text{buffer} \\ \text{warning} & \text{if } \text{buffer} \leq |\Delta t| < \tau \\ \text{violation} & \text{if } |\Delta t| \geq \text{threshold } \tau \end{cases} \quad (7)$$

**Dynamic Adjustment Mechanism** When deviations occur, the system initiates a three-phase response:

1. *Impact Assessment*:

$$\text{impact}(e) = \sum_{n \in \text{affected}(e)} \text{severity}(n) \times \text{urgency}(n) \quad (8)$$

2. *Solution Generation*:

$$S^* = \arg \min_{s \in \text{Solutions}} \{\text{cost}(s) | \text{feasible}(s)\} \quad (9)$$

3. *Coordination Protocol*:

$$\text{update} : (W_{\text{current}}, S^*) \rightarrow W_{\text{new}} \quad (10)$$

For instance, if James's flight is delayed:

- Spatial agent detects arrival time change
- Temporal agent calculates ripple effects
- Role agents evaluate reassignment options
- Safety agent verifies continued supervision coverage

The meta-planner  $\mathcal{MP}$  coordinates these responses while maintaining global constraint satisfaction.

## E.5 INTEGRATED WORKFLOW NETWORK

Table 14 presents the resulting workflow network  $\mathbf{W}^*$ , which includes all nodes and edges, and their assigned agents and protocols.

1. *Role Nodes*:

- Cook1: Sarah (primary) or Grandma (if at home) with 4-hour turkey + 2-hour sides
- Driver1: James (after car rental) or Michael
- Driver2: Available person after initial pickups
- Supervisor: Must be present while turkey cooks

2. *Dependencies*:

- Temporal: Verified airport processing + travel times
- Spatial: Traveling routes with traffic consideration
- Safety: Continuous oven supervision requirement

3. *Agent Monitoring*:

- Temporal Agent: Schedules with verified buffer times
- Spatial Agent: Real-time location and route mgmt.
- Safety Agent: Role coverage for supervision

## E.6 AGENT INTERACTION SPECIFICATIONS

Please, see Table 15.

Table 15: Agent Interaction Protocols and State Transitions

| Interaction Type                 | Protocol                                                             | State Transitions                                                 | Validation Rules                                          |
|----------------------------------|----------------------------------------------------------------------|-------------------------------------------------------------------|-----------------------------------------------------------|
| <i>Node-to-Node Interactions</i> |                                                                      |                                                                   |                                                           |
| Cook ↔ Supervisor                | Protocol: cooking_handoff()<br>Message: (task, duration, reqs.)      | States: prep → cooking → comp.<br>Trigger: task_state_change()    | Validate: coverage()<br>Alert: coverage_gap()             |
| Driver1 ↔ Driver2                | Protocol: pickup_handoff()<br>Message: (location, time, passenger)   | States: available → enroute → comp.<br>Trigger: location_change() | Validate: timing_feasible()<br>Alert: schedule_conflict() |
| <i>Edge Agent Operations</i>     |                                                                      |                                                                   |                                                           |
| Temporal Agent                   | Protocol: schedule_monitor()<br>Message: (event, time, dependencies) | States: scheduled → active → comp.<br>Trigger: time_milestone()   | Validate: timing_feasible()<br>Alert: delay_impact()      |
| Spatial Agent                    | Protocol: location_track()<br>Message: (actor, position, dest.)      | States: idle → moving → arrived<br>Trigger: position_update()     | Validate: route_feasible()<br>Alert: travel_delay()       |

## E.7 AUGMENTED PROBLEM STATEMENT REVISED WITH $W^*$

Given the  $W^*$  generated by Alas’s meta-planner  $\mathcal{MP}$ , the Thanksgiving Dinner Planning problem statement is revised as follows:

### Initial Setup:

- Mom (Sarah) is hosting Thanksgiving dinner at 6:00 PM in Boston. The following family members are traveling:
- Dad (James) flying from San Francisco, landing at 1:00 PM Eastern time.
- Sister (Emily) flying from Chicago, landing at 2:30 PM
- Brother (Michael) driving from New York, estimated arrival 3:00 PM at home
- Grandma is healthy and needs to be picked up from her home in suburban Boston

### \* Common Sense Augmented Constraints:

- The airport luggage pickup time after landing is 30 minutes.
- Renting a car takes 30 minutes.
- One person can simultaneously prepare turkey and side dishes.
- Grandma prefers Michael to pick her up, provided that it does not cause the dinner time delay (soft constraint).
- Grandma and Sarah prefer not to cook together in the kitchen.
- The best turkey receipt and baking instructions included.
- Traffic congestion is not factored into current planning.

### Planning Validation Set:

1. All tasks and dependencies must be strictly observed in the plan, or the plan fails.
2. Dinner time is strictly at 6:00 PM, all tasks must be completed by then (redundancy).
3. Account for the idle time of each person.
4. The schedule consists of three columns: time, task, and assigned person(s).

## E.8 EXPERIMENT #1: SEQUENTIAL PLANNER WITH COMMON SENSE

The first experiment utilized the augmented problem specification with common sense reasoning, incorporating realistic constraints such as luggage claim time and rental car pickup time.

We evaluated four standalone LLMs alongside ALAS. Both ALAS and Gemini consistently generated feasible schedules similar to Table 16, while other LLMs encountered significant challenges.

Upon analyzing the number of iterations required for a feasible plan, DeepSeek and Claude each needed one revision (two iterations), while GPT4o required two revisions (three iterations). In terms of scheduling quality—measured by slack time, total driving distance, and load balance—DeepSeek (Table 16) outperformed both GPT4o (Table 17) and Claude (Table 18). DeepSeek optimized efficiency by having James wait at the airport for 30 minutes to pick up Emily. In contrast, Claude inefficiently scheduled James to drive home and then return to the airport for Emily, creating unnecessary travel. GPT4o assigned James to return home and tasked Michael with separately picking up Emily and then Grandma, resulting in suboptimal load distribution. A more efficient solution would have scheduled Michael to collect Emily first, then proceed with her to Grandma’s



Table 16: DeepSeek’s Plan, Two Iterations

| Time         | Task                                              | Assigned |
|--------------|---------------------------------------------------|----------|
| 1:00 PM      | James lands at Boston                             | James    |
| 1:00–1:30 PM | James picks up luggage                            | James    |
| 1:30–2:00 PM | James rents a car                                 | James    |
| 2:00 PM      | Turkey in oven (4 hours; requires monitoring)     | Sarah    |
| 2:00–3:00 PM | James waits at airport (idle)                     | James    |
| 2:30 PM      | Emily lands at Boston                             | Emily    |
| 2:30–3:00 PM | Emily waits for luggage                           | Emily    |
| 3:00 PM      | James picks up Emily                              | James    |
| 3:00 PM      | Michael arrives home                              | Michael  |
| 3:00 PM      | Michael departs to Grandma                        | Michael  |
| 3:30 PM      | Michael picks up Grandma                          | Michael  |
| 3:30–4:00 PM | Michael drives back home with Grandma             | Michael  |
| 3:00–4:00 PM | James drives Emily home (airport to home: 1 hour) | James    |
| 4:00 PM      | James and Emily home                              | James    |
| 4:00 PM      | M. and Grandma home                               | Michael  |
| 4:00–6:00 PM | Sarah prepares side dishes                        | Sarah    |
| 6:00 PM      | Thanksgiving dinner begins                        | All      |

home, allowing all three to return together—saving 30 minutes of driving time while enhancing Grandma’s experience of seeing both grandchildren simultaneously.

Table 17: GPT4o’s Plan, Three Iterations

| Time         | Task                            | Assigned |
|--------------|---------------------------------|----------|
| 1:00 PM      | Land at BOS Airport             | James    |
| 1:00-1:30 PM | Luggage pickup                  | James    |
| 1:30-2:00 PM | Rent car                        | James    |
| 2:00 PM      | Start turkey                    | Sarah    |
| 2:00-3:00 PM | Drive home                      | James    |
| 2:30 PM      | Land at BOS Airport             | Emily    |
| 3:00 PM      | Arrive home                     | Michael  |
| 3:00-4:00 PM | Drive to airport, pick up Emily | Michael  |
| 4:00-5:00 PM | Return home with Emily          | Michael  |
| 5:00-5:30 PM | Drive to Grandma’s              | Michael  |
| 5:30-6:00 PM | Return with Grandma             | Michael  |
| 4:00-6:00 PM | Prepare side dishes             | Sarah    |
| 6:00 PM      | Dinner served                   | All      |

Table 18: Claude’s Plan, Two Iterations

| Time         | Task                            | Assigned |
|--------------|---------------------------------|----------|
| 1:00 PM      | Land at BOS Airport             | James    |
| 1:00-1:30 PM | Luggage pickup                  | James    |
| 1:30-2:00 PM | Rent car                        | James    |
| 2:00 PM      | Start turkey                    | Sarah    |
| 2:00-3:00 PM | Drive home                      | James    |
| 2:30 PM      | Land at BOS Airport             | Emily    |
| 3:00 PM      | Arrive home                     | Michael  |
| 3:00-4:00 PM | Drive to airport, pick up Emily | James    |
| 4:00-5:00 PM | Return home with Emily          | James    |
| 4:30-5:00 PM | Drive to Grandma’s              | Michael  |
| 5:00-5:30 PM | Return with Grandma             | Michael  |
| 4:00-6:00 PM | Prepare side dishes             | Sarah    |
| 6:00 PM      | Dinner served                   | All      |

### E.8.1 OBSERVATIONS OF ERRORS IN STANDALONE LLMs

Although DeepSeek and Claude eventually produced feasible static plans in their second iterations, Tables 19 and 20 highlight critical errors in their initial attempts.

These errors included misestimated travel times (calculating 60-minute trips as 45 or 30 minutes, highlighted in shaded red) and implausible scheduling decisions, such as beginning turkey preparation at 10 AM or allowing James to depart the airport without Emily.

Such failures result from context erosion in extended prompts Liu et al. (2024); Xiao et al. (2024) and expanding context windows. Research demonstrates that extended contexts accelerate information loss Park et al. (2023); Wei et al. (2023), leading to constraint violations. ALAS circumvents these issues through its modular architecture, where specialized agents process only domain-relevant information while independent validation mechanisms ensure constraint adherence.

**Handling Long Dependencies** Complex scheduling problems reveal cascading errors when dependencies overlap. Critical constraints, particularly those involving multiple factors, frequently get dropped during iterative problem-solving.

**Reason:** Cognitive limitations restrict simultaneous constraint tracking, making exhaustive verification challenging in single processing passes.

**Solution Framework:**

Table 19: Family Reunion - Claude. Gray indicates questionable, red wrong time calculation.

| Time   | Task                            | Assigned |
|--------|---------------------------------|----------|
| 1 0:00 | Put turkey in oven (4-hour)     | S        |
| 1 3:00 | James lands at BOS              | J        |
| 1 3:30 | James rents car at BOS          | J        |
| 1 4:00 | James leaves airport for home   | J        |
| 1 4:30 | Emily lands at BOS              | E        |
| 1 5:00 | James & Michael arrive home;    | J, M     |
| 1 5:15 | M leaves to pick up E from BOS  | M        |
| 1 5:30 | S begins side dishes (2-hour)   | S        |
| 1 6:15 | M arrives at BOS to pick up E   | M, E     |
| 1 6:30 | James leaves to pick up Grandma | J        |
| 1 7:00 | J arrives at G's house          | J, G     |
| 1 7:00 | Michael and Emily arrive home   | M, E     |
| 1 7:30 | J and G home; side dish done    | J, G, S  |
| 1 8:00 | Turkey complete, dinner ready   | All      |

Table 20: Family Reunion - DeepSeek. Red indicates wrong time calculation.

| Time   | Task                                                  | Assigned |
|--------|-------------------------------------------------------|----------|
| 1 2:00 | S starts cooking the turkey (4 hours, ready by 16:00) | S        |
| 1 3:00 | J lands at BOS and rents a car                        | J        |
| 1 3:30 | James drives from BOS to Grandma's                    | J        |
| 1 4:30 | James picks up Grandma, Emily lands                   | J, G     |
| 1 5:00 | Michael home (from NY)                                | M        |
| 1 5:15 | James and Grandma arrive back home (30 min drive)     | J, G     |
| 1 5:15 | M leaves to pick up E from BOS (60 min round trip)    | M        |
| 1 6:00 | Turkey done; S starts side dishes (2 hours)           | S        |
| 1 6:15 | M and E return home from BOS                          | M, E     |
| 1 8:00 | Family reunion dinner begins with all members present | All      |

- Isolate and systematically enumerate atomic task dependencies.
- Implement comprehensive verification of global constraint satisfaction.
- Develop robust mechanisms for systematic conflict resolution.

**Stale Memory and Iterative Revisions** Iterative solutions risk propagating errors due to incomplete constraint resets.

*Reason:* Excessive reliance on previous solutions without comprehensive constraint re-evaluation leads to compounding errors.

**Relation to Gödel's Incompleteness:**

- Formal systems capable of arithmetic necessarily contain unprovable truths.
- Similarly, inherited solution errors inhibit consistent constraint satisfaction.
- Clean-state resets become essential for systematic error prevention.

**Implementation Strategy** Reset to a clean baseline state for each iteration, thoroughly re-evaluating all constraints.

*Core Challenges:*

- Effective management of nested dependencies.
- Prevention of residual errors across iterations.
- Maintenance of cross-iteration consistency.

Table 21: Sequential Planning. (# = iterations)

| LLM      | # | Notable Features                                                                |
|----------|---|---------------------------------------------------------------------------------|
| DeepSeek | 2 | Optimized airport wait time for James; balanced workload                        |
| Claude   | 2 | Unnecessary travel between pickup tasks (no need to go home before next pickup) |
| GPT4o    | 3 | Extra travel for Michael; suboptimal load balance                               |

Table 21 synthesizes the detailed schedules documented in Tables 16, 17, and 18. DeepSeek demonstrated good scheduling efficiency by optimizing James's airport wait time for Emily's pickup, requiring only two iterations for convergence. Although GPT4o eventually produced a valid solution after three iterations, it created suboptimal travel patterns with redundant trips by Michael. Claude's solution, though feasible in two iterations, incorporated unnecessary travel between pickup tasks. In contrast to the inconsistent performance of standalone LLMs, ALAS consistently generated feasible and efficient plans in all test runs.

## E.9 EXPERIMENT #2: REACTIVE PLANNER FOR FLIGHT DELAY

This disruption scenario was stated in the main body of the paper in Section F.2. Under the prompt for reactive planning, it states the disruption as: "At noon, James' flight is delayed until 4:00 PM. Update the schedule to meet the deadline at 6:00 pm while meeting all constraints."

We tested only ALAS, DeepSeek and Claude 3.7 for disturbance handling, because they survived sequential planning. ALAS successfully generated a feasible reactive plan, whereas DeepSeek and Claude 3.7 failed.

The challenge: James (assigned to pick up Emily) is delayed until 4:00 PM. Both DeepSeek and Claude failed in seven out of ten runs by: (1) scheduling Emily for a taxi, violating family pick-up constraints; (2) delaying or missing Grandma’s pickup; or (3) missing the dinner deadline. An example failed schedule (the name of the LLM is purposely concealed) is presented in Table 22, where one can see that when all family members are ready to eat the delicious turkey, “oops, Grandma was not picked up.”

Table 22: Failed Reactive Planning Schedule with Flight Delay Disruption

| Time           | Task                                        | Explanation                                                                                                                                                                                                                                                                                                                                                                                   |
|----------------|---------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| (Before Noon)  | Tasks like initial prep are underway        | The disruption is received at noon.                                                                                                                                                                                                                                                                                                                                                           |
| 2:00 PM        | Sarah puts turkey in the oven.              | <b>Unchanged:</b> Critical path for food. Sarah must stay home to supervise (2:00 PM - 6:00 PM).                                                                                                                                                                                                                                                                                              |
| 2:30 PM        | Emily lands at BOS.                         | <b>Unchanged:</b> Sister’s flight arrives as scheduled.                                                                                                                                                                                                                                                                                                                                       |
| 3:00 PM        | Michael arrives home.                       | <b>Unchanged:</b> Brother arrives from NY. <b>New Role:</b> He is now the designated driver for Emily’s pickup.                                                                                                                                                                                                                                                                               |
| ~3:15 PM       | Michael departs home for BOS Airport.       | Michael must leave soon after arriving to pick up Emily. (60 min travel time to BOS).                                                                                                                                                                                                                                                                                                         |
| 4:00 PM        | James lands at BOS.                         | <b>Disruption Effect:</b> Dad arrives 3 hours later than planned.                                                                                                                                                                                                                                                                                                                             |
| 4:00 PM        | Sarah starts cooking side dishes.           | <b>Unchanged:</b> Side dishes still need 2 hours. Sarah starts them as she is the only one home.                                                                                                                                                                                                                                                                                              |
| 4:00 - 5:00 PM | James deplanes, gets luggage, rents a car.  | <b>Delayed:</b> James needs to rent a car after landing.                                                                                                                                                                                                                                                                                                                                      |
| ~4:15 PM       | Michael arrives at BOS Airport.             | Michael reaches the airport to meet Emily.                                                                                                                                                                                                                                                                                                                                                    |
| ~4:15 - 4:45   | Michael and Emily meet at BOS               | Allow time for Emily to get luggage and meet Michael.                                                                                                                                                                                                                                                                                                                                         |
| ~4:45 PM       | Michael and Emily depart BOS for home.      | They begin the 60-minute drive home.                                                                                                                                                                                                                                                                                                                                                          |
| ~5:00 PM       | James departs BOS for home (in rental car). | James starts his 60-minute drive home after getting rental car.                                                                                                                                                                                                                                                                                                                               |
| ~5:45 PM       | Michael and Emily arrive home.              | Brother and Sister are home.                                                                                                                                                                                                                                                                                                                                                                  |
| 6:00 PM        | James arrives home.                         | Dad arrives home just in time for the planned dinner.                                                                                                                                                                                                                                                                                                                                         |
| 6:00 PM        | Dinner Time                                 | Turkey and side dishes are ready. Sarah, Michael, Emily, and James are home.                                                                                                                                                                                                                                                                                                                  |
| P OST 6:00 PM  | Grandma’s Pickup - Unresolved               | <b>Major Impact:</b> With James arriving at 4:00 PM and needing an hour for car rental (leaving BOS at 5:00 PM), he cannot pick up Grandma and return by 6:00 PM. Michael is busy with Emily’s pickup until 5:45 PM and cannot make the ~1.5 hr round trip for Grandma before 6:00 PM. Sarah must supervise the oven. <b>Under these constraints, Grandma cannot join the 6:00 PM dinner.</b> |

These failures are due to greedy rescheduling and increased context loss during replanning. Studies confirm that longer contexts can worsen reliability Wei et al. (2023); Zhang et al. (2023). ALAS generated feasible plans in all runs by: (1) updating James’s state in persistent memory; (2) detecting conflicts systematically; (3) evaluating alternatives; and (4) validating all constraints. Following LRCP (Local Reactive Compensation Protocol, depicted in Algorithm ??), ALAS produced four distinct solution patterns, all feasible. While requiring 2.5× the computation time (12.1s vs. 4.8s), this overhead ensures feasibility, crucial in time-sensitive domains.

## F ADDITIONAL CASE STUDIES AND ANALYSIS

**ALAS’s Reactive Plan and Stateful Explanation.** ALAS proposes a simple yet effective reroute: Michael drives straight to Boston Airport rather than stopping at home first. This common-sense spatial adjustment—overlooked by the other LLMs—originates from  $\mathcal{MP}$ ’s state-aware reasoning module. After collecting Emily, Michael proceeds directly to Grandma’s house, trimming roughly 30 minutes of travel and giving Grandma the pleasant surprise of seeing two grandchildren arrive together. Table 23 lists the resulting feasible schedule. The critical advantage is that ALAS’s continuous tracking of each participant’s state and history enables timely compensations and preserves the on-time family reunion.

Table 23: ALAS Reactive Plan: Optimized routing via persistent state history and compensation.

| Time         | Task                                                                                                                 | Assigned         |
|--------------|----------------------------------------------------------------------------------------------------------------------|------------------|
| 12:00 PM     | James' delay known, Michael on his way from NYC to home, is rerouted to Boston airport to meet Emily (4-hour drive). | Michael          |
| 2:00 PM      | Start cooking turkey                                                                                                 | Sarah            |
| 2:30 PM      | Emily lands at Boston                                                                                                | Emily            |
| 3:00 PM      | Emily gets her luggage                                                                                               | Emily            |
| 3:00 PM      | Michael arrives at Logan airport, picks up Emily.                                                                    | Michael          |
| 3:00–4:00 PM | Michael drives Emily home                                                                                            | Michael          |
| 4:00 PM      | Michael departs for Grandma                                                                                          | Michael          |
| 4:00 PM      | James lands at Boston Airport                                                                                        | James            |
| 4:00–4:30 PM | James picks up luggage                                                                                               | James            |
| 4:30–5:00 PM | James rents car (30 minutes).                                                                                        | James            |
| 4:30 PM      | Michael arrives at Grandma's                                                                                         | Michael          |
| 5:00 PM      | Michael & Grandma arrive home.                                                                                       | Michael, Grandma |
| 5:00–6:00 PM | James drives home from BOS                                                                                           | James            |
| 4:00–6:00 PM | Sarah prepares side dishes (overlaps with turkey).                                                                   | Sarah            |
| 6:00 PM      | James arrives home. Dinner served.                                                                                   | All              |

Table 24: Dynamic Urban Ride-Sharing

Table 25: Family Reunion Planning Problem

**Objectives:** Schedule vehicles to deliver passengers to airport during [8:45, 9:00], minimizing vehicle travel distance while ensuring on-time arrivals and maximizing passenger satisfaction.  
**Locations:** Seven locations:  $V = \{A, B, \dots, F\}$ , where  $G$  is Boston Airport (BOS). Urban locations  $A$ – $F$  are all 10 km apart, airport distances 30+ km.

|                 |    |    |    |    |    |    |
|-----------------|----|----|----|----|----|----|
|                 | A  | B  | C  | D  | E  | F  |
| $\rightarrow G$ | 35 | 33 | 36 | 34 | 32 | 31 |

**Travel Speed:** ( $A$ – $F$ ) 60, ( $A$ – $F \rightarrow G$ ) 100 km/h

**Passenger Requests:** with BOS arrival deadlines:

- $r_1$ :  $A$ , to  $G$  by 08:45 -  $r_2$ :  $B$ , to  $G$  by 08:50
- $r_3$ :  $C$ , to  $G$  by 08:55 -  $r_4$ :  $D$ , to  $G$  by 09:00

**Available Vehicles** (Capacity 2 passengers):

- $k_1$ : at  $A$ ,  $k_2$ : at  $C$ , and  $k_3$ : at  $E$
- Battery levels:  $k_1$ : 90%,  $k_2$ : 75%,  $k_3$ : 60%

**Potential Disruptions:** New passenger requests, vehicle availability changes (battery levels/breakdowns at 0.05/hour), and traffic delay, etc. Replanning may require rolling back promised pickup time to the existing passengers and replan for all.

**Objectives:** On time family reunion dinner at 6:00 PM

**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 (Bro): Driving, arrives 3:00 PM from NY
- Grandma: Needs pickup from suburban Boston

**Cooking Requirements:**

- Turkey: 4 hours cooking time; side dishes: 2 hours
- Someone must stay home during oven baking time

**Transportation Constraints:**

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

**Key Constraints:**

- All family members home before 6:00 PM
- Turkey and sides ready by 6:00 PM
- All pickups completed with available drivers
- Oven baking supervision maintained

## F.1 CASE STUDY 1: TRANSPORTATION SCHEDULING

**Purpose:** conclude the illustrative example and show that even simple problems cause standalone LLMs to miss basic planning requirements.

**PROBLEM.** The Urban Ride Sharing (URS) problem (Table 24) coordinates vehicles to deliver passengers before deadlines while minimizing distance, with mid-execution disruptions.

**SETUP.** Initial: “Create an optimal schedule for this ride-sharing scenario that minimizes total travel distance while meeting all passenger deadlines: [Table 24].” Reactive: “Passenger  $r_2$  cancels at 8:05; a new passenger  $r_5$  at  $F$  requests pickup with a 9:30 deadline. Update the schedule.” ALAS augments prompts with structured workflow templates (Alg. 1) and role-based instantiation. Ten independent trials; we report mean $\pm$ sd.

**#1. Sequential planning.** All models met deadlines; ALAS achieved superior efficiency:  $95.1 \pm 13.0$  km vs.  $118.9 \pm 16.6$  km for baseline LLMs (20% improvement,  $p < 0.01$ ). Figure 8 shows the ALAS schedule.

**#2. Reactive planning.** With  $r_2$  canceled (8:05) and  $r_5$  added (8:10), ALAS repaired successfully in all runs. Baseline LLMs commonly lost vehicle state, duplicated assignments, or ignored updated deadlines—consistent with structural limits and statelessness (Sec. 2.1). Full results: Appx. D.

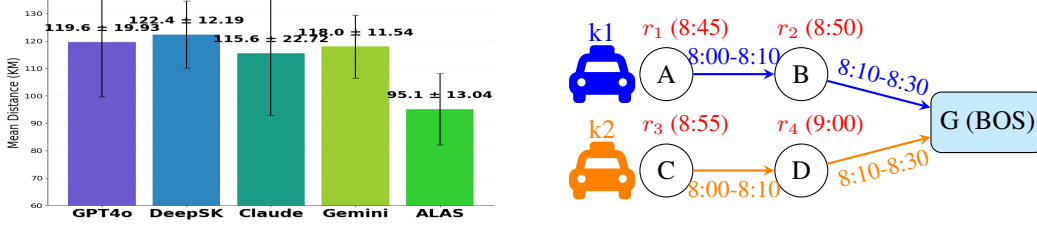


Figure 8: Comparison of ride-sharing solutions generated by Alas and baseline LLMs. (Left) Mean total travel distance (km) with standard deviation error bars over 10 independent runs for each method, illustrating Alas’s improved efficiency. (Right) Optimal schedule generated by Alas for the URS task, utilizing two vehicles ( $k_1, k_2$ ) to serve four passengers ( $r_1-r_4$ ).

## F.2 CASE STUDY 2: EVENT COORDINATION

This study shows why standalone LLMs often fail even in static planning and behave inconsistently in reactive settings, while ALAS remains stable across both.

**Problem.** The Family Reunion problem (Table 25): coordinate pickups, cooking, and shared resources with a 18:00 dinner deadline.

**Set up.** All methods: “Create a detailed schedule that satisfies all constraints: [Table 25].” Reactive: “At noon, James’ flight is delayed to 16:00. Update the schedule to meet the 18:00 deadline.” ALAS uses Algorithm 1; ten independent runs.

**#1. Sequential planning.** ALAS produced a *feasible* schedule in all runs; baselines frequently violated hard constraints and required retries (Tables 19, 20 in Appx. E). Typical errors: travel-time arithmetic (e.g., 60 min treated as 30–45 min), commonsense slips (e.g., cooking far too early). Root cause: long-context degradation Liu et al. (2024); Xiao et al. (2024). ALAS confines context per agent and employs an *independent* validator before finalization. A small *commonsense agent* inserts realistic slack (e.g., luggage pickup); see Appx. E.2.3.

**#2. Reactive planning.** Under a delay to 16:00, DeepSeek and Claude failed in 7/10 runs (e.g., violating pickup constraints, missing 18:00). Greedy one-shot rescheduling with longer prompts exacerbated context loss Wei et al. (2023); Zhang et al. (2023). ALAS succeeded in all runs via LCRP: logging state, detecting conflicts, testing alternatives, and re-validating locally. Full logs: Appx. E.