Constraint-Checked LLM Planning for Cluster Recovery

Jiesi Yang[1,*], Guanyu Ding[2], Shiyu Yang[3]

[1] University of Utah, Salt Lake City, USA

[2] Amazon Web Services, Inc., Seattle, USA

[3] Amazon Web Services, Inc., Seattle, USA

[*]Corresponding author: Jiesi Yang (u1524481@umail.utah.edu)

Abstract

Node failures invalidate the pre-computed schedules of dependency-aware batch workloads, and the recovery decision must be taken while the cluster keeps running. Language models are attractive repair planners because they can weigh heterogeneous objectives expressed in text, but their raw plans are frequently infeasible and their inference latency competes with the recovery itself. This paper presents CC-LLM, a constraint-checked planning architecture in which a small language model proposes repairs and independent components make them safe and timely. CC-LLM (i) scopes the failure with a slack-aware delay analysis over the task graph, (ii) commits every decision that must execute before the expected inference latency to a verified fast incumbent and exposes only the remaining queued decisions to the model, (iii) encodes these decisions as compact decision sheets whose answers are constrained by a JSON Schema, and (iv) checks each proposal for processing-capacity, memory, dependency, node-availability and deadline violations, returning violation certificates that drive conflict-directed revisions before a legalizer and an anytime arbiter select the dispatched plan. On 18 failure scenarios built from real Pegasus executions of six scientific workflows and under an equal 20 s decision budget, CC-LLM with a 0.5B-parameter model on a single CPU core returned a verified feasible plan in every scenario with zero voluntary migrations, achieved the lowest mean repair cost (777 versus 845 for a warm-started CP-SAT solver on the same scope and 1242 for heuristic local repair), shortened recovery time by 18% relative to local repair, and required a mean decision latency of 6.4 s. Unchecked model proposals were feasible in only 44% of the consultations, and certificate feedback reduced conflicting decisions by 67%.

Keywords

Cluster scheduling, failure recovery, large language models, constraint checking, rescheduling, workflow DAG, decision latency.

1. Introduction

Modern compute clusters execute large numbers of dependency-aware batch jobs, from data-parallel analytics to scientific workflows, under a schedule that maps every task to a node and a start time [1], [2], [3]. Node failures are a routine part of operating such systems: field data from production high-performance computing sites report between 20 and 1000 failures per system and year [4], and production cluster managers are explicitly engineered to minimize fault-recovery time [1]. When a node fails, every task that was running or queued on it must be restarted elsewhere, and the resulting delay propagates along the task graph to downstream tasks that may be planned on healthy nodes. The pre-computed schedule therefore becomes infeasible exactly when deadlines are most at risk.

A recovery decision has four competing requirements. It must be *feasible* with respect to processing capacity, memory, task dependencies and the reduced set of nodes; it must be *timely*, because every second spent deciding is a second in which displaced work does not run; it should be *stable*, i.e., it should move as few unaffected tasks as possible, since each migration discards locality, cached state and operator expectations [5], [6]; and it should protect job *deadlines*. Classical repair heuristics, such as right-shifting combined with earliest-finish-time reassignment [7], [8], are fast and stable but myopic. Time-limited optimization solvers can find better repairs, but they release their plan only when they stop, and the displaced tasks wait for that moment.

Large language models (LLMs) offer a third option. They can read a compact description of the situation, follow operator preferences and propose non-myopic edits, for example deferring a task of a job with ample slack so that an urgent job can use the freed capacity. However, autoregressive LLMs cannot plan reliably on their own [9], [10]; their plans violate constraints that a scheduler must never violate, and recent LLM-based schedulers rely on dedicated fine-tuning [11]. In an operational setting the model also sits on the critical path: inference latency adds directly to recovery time, and on commodity CPU servers this latency is measured in seconds.

This paper studies the question: *can a language model restore a feasible schedule after a node failure within a limited decision time while migrating fewer tasks?* We identify four bottlenecks that a practical answer must remove: (B1) raw proposals are frequently infeasible; (B2) inference latency is paid by the very tasks that are being repaired; (B3) the prompt, and hence the latency, grows with the workload; and (B4) small models handle long numeric contexts poorly. We address them with CC-LLM, a constraint-checked planning architecture in which the model only proposes, while independent deterministic components scope, verify, correct and arbitrate. The main contributions are as follows.

1) **Latency-aware formulation.** We formalize LLM-in-the-loop cluster recovery with a realization operator that forbids every changed decision from starting before the decision exists. Decision latency, including model inference, is thereby charged to recovery quality instead of being reported as a separate cost.

2) **Scoping that hides inference latency.** A slack-aware impact analysis, a verified fast incumbent, a commit horizon and a decision-relevance filter reduce the repair problem from 123 pending tasks to 2.3 model decisions on average and keep every urgent decision off the model's critical path.

3) **Certified proposals.** Decision sheets constrained by a JSON Schema, an independent checker that issues capacity, memory, dependency, node and deadline certificates, conflict-directed feedback, a serial schedule-generation legalizer and anytime arbitration guarantee a feasible plan under any budget and never return a plan that is worse than the verified incumbent.

4) **Evaluation on real workflow executions.** We build failure scenarios from real Pegasus executions of six applications [2], [12] and compare CC-LLM with heuristic local repair, full rescheduling and warm-started CP-SAT under equal budgets, complemented by ablations, budget sweeps, runtime perturbation, two-node failures and a model-scale study.

With a 0.5B-parameter model executed on a single CPU core, CC-LLM returned verified feasible repairs in all scenarios, produced no voluntary migration, obtained the lowest mean repair cost among all methods and the shortest recovery time among the methods that do not move unaffected tasks, with a mean decision latency of 6.4 s. Section II reviews related work, Section III formulates the problem, Section IV presents CC-LLM, Sections V and VI describe the experiments and results, Section VII discusses the findings and Section VIII concludes.

2. Related Work

2.1. Rescheduling and Fault-Tolerant DAG Scheduling

Vieira *et al.* [5] organize rescheduling into strategies, policies and methods and distinguish repair-based from complete rescheduling; right-shift repair and partial rescheduling are the standard means of preserving stability. Herroelen and Leus [6] survey reactive and proactive project scheduling and emphasize schedule stability as an explicit objective, which motivates our migration-aware cost. For workflows, Sakellariou and Zhao [8] reschedule only at selected points to limit the overhead of re-planning, a principle that our impact scoping pushes further by restricting the decision scope itself. List scheduling with upward ranks and earliest-finish-time placement (HEFT) [7] and serial schedule-generation schemes for resource-constrained project scheduling [13] remain the backbone of fast schedulers and of our legalizer. Production cluster managers such as Borg [1] combine task packing with fault-recovery mechanisms, Tetris [14] packs multi-resource demands to reduce fragmentation, and Decima [15] learns workload-specific DAG scheduling policies with reinforcement learning. These systems allocate resources online; we instead study the repair of an existing plan after a failure, where stability, deadlines and the latency of the repair decision interact.

Wang et al. [16] make an important methodological contribution with PRESAGE, which couples adapted time-series foundation models, retrieval-augmented forecasting and calibrated quantiles with an uncertainty-aware reinforcement-learning scheduler, so that forecast uncertainty informs scheduling decisions directly. The experimental design is equally instructive: prediction quality is connected to operational outcomes on six public cloud traces, a vLLM serving harness and an eight-GPU Kubernetes testbed. Across ten stochastic replays, SLO violations decrease from 9.4% with reactive scaling to 1.8% while normalized resource cost decreases from 1.00 to 0.82, and the physical testbed confirms the gains with violation rates of 10.2% and 2.2% at normalized costs of 1.00 and 0.85. This end-to-end validation across simulation and deployment sets a valuable standard for evaluating scheduling policies and shows convincingly that forecasting improvements translate into scheduling benefits. CC-LLM addresses the complementary, reactive problem of repairing an existing DAG schedule after a node failure, with stability and decision latency included explicitly.

Stability in repair is commonly quantified by the number of changed assignments and by start-time deviations relative to the baseline schedule [5], [6]. Our cost (5) prices voluntary node changes explicitly, while start-time deviations are bounded structurally because unaffected tasks may only be right-shifted. This separates the unavoidable disruption caused by the failure itself (forced moves) from the disruption introduced by the repair method, which is the quantity that distinguishes the methods.

2.2. Language Models for Planning and Optimization

PlanBench [10] shows that LLMs fall short on plan generation, and the LLM-Modulo position [9] argues that LLMs should act as candidate generators inside loops with sound external verifiers. LLM-based job-shop scheduling has been demonstrated with a dedicated supervised dataset [11]. FunSearch [17] pairs an LLM with an automated evaluator to discover programs, while Reflexion [18] and Self-Refine [19] improve outputs through verbal feedback.

Qin et al. [20] provide compelling evidence for separating neural generation from symbolic constraint enforcement with VA-NSF. A grammar-constrained actor produces typed transaction structures, and an Answer Set Programming verifier checks business rules and returns structured reports that identify the violated rule and the responsible structure. On 8,600 curated financial instructions, the system achieves 91.8% macro-averaged transaction accuracy with at most 0.3% constraint violations, and median symbolic verification takes only 24 ms. Their controlled repair comparison is especially valuable for feedback design: with

precise diagnostics, 91.8% of initially violating Constrained-tier transactions are resolved within three iterations, versus 73.1% with generic feedback. This experiment offers strong quantitative support for precise, externally grounded diagnostics as the driver of repair, beyond what self-critique alone can provide.

Zhang [21] develops a rigorous verification-guided framework for RTL repair that combines nine families of counterexample tags, cone-of-influence localization and bounded sequential equivalence checking. The controlled experiments are carefully designed: on 261 injected defects, the combined signals rank the faulty line first in 47.9% of the cases, compared with 31.4% without elaboration diagnostics, and in a repair ablation on 90 injected defects under the same 25-query verification budget and edit space, guided repair raises the success rate from 36.7% to 73.3% while reducing the mean number of queries per successful repair from 13.8 to 8.3. The study further strengthens its conclusions with an evaluation on 125 real failing LLM generations, which grounds the analysis in realistic defects. These results clearly demonstrate how diagnostic representation and localization make a limited verification budget substantially more effective.

Together, these studies offer valuable methodological and experimental guidance for evidence-guided decision loops: diagnostics produced by an external verifier should be precise and localized, and their benefit should be established through controlled comparisons under a fixed budget. CC-LLM shares this perspective and carries it to cluster recovery through resource and precedence certificates, a commit horizon that protects urgent execution from inference latency, and verified anytime arbitration under a wall-clock budget. In the same spirit as these controlled comparisons, our evaluation separately measures raw proposal feasibility, conflict reduction and dispatched-plan quality to distinguish the contributions of generation, feedback and deterministic safeguards.

2.3. Structured Decoding and Exact Solvers

Grammar-constrained decoding [22] guarantees that model outputs follow a formal structure without fine-tuning; we use JSON-Schema enumerations so that every answer names an existing option. Code-specialized small models such as Qwen2.5-Coder [23] make on-premise deployment practical. Constraint programming with CP-SAT [24] is the reference exact technology for resource-constrained scheduling and, warm-started from the existing schedule, forms our strongest baseline.

3. Problem Formulation

Cluster and workload. A cluster consists of nodes $n \in N$ with processing capacity C_n (cores) and memory M_n. Job j is a directed acyclic graph $G_j = (V_j, E_j)$ with release time r_j and deadline d_j; task i has duration p_i, core demand c_i and memory demand m_i. A pre-failure plan σ^0 assigns every task a node n_i^0 and start time s_i^0.

Failure event. At time t_f the nodes $F \subset N$ fail. Tasks finished before t_f are kept; tasks running on surviving nodes continue; the *forced set* H contains every task planned on a failed node that has not finished, i.e., running tasks are restarted from scratch and queued tasks must be moved. The repair set is $P = H \cup U$, where U contains the not-yet-started tasks on surviving nodes. A repair σ assigns (n_i, s_i) to every $i \in P$ subject to

$$n_i \in N \setminus F, \quad s_i \geq t_f, \quad \forall i \in P \tag{1}$$

$$s_v \geq s_u + p_u, \quad \forall (u, v) \in E \tag{2}$$

$$\Sigma_{i \in A(n,t)} c_i \le C_n, \quad \Sigma_{i \in A(n,t)} m_i \le M_n, \quad \forall n, t \tag{3}$$

where $A(n, t)$ is the set of tasks active on node n at time t, including tasks that continue to run after t_f.

Latency-aware realization. A repair is only useful from the moment it exists. If a method needs decision latency L, every *changed* task, i.e., every forced task and every task whose node changes or whose start moves earlier, cannot start before $t_f + L$:

$$s_i \ge t_f + L, \quad \forall i \in \Delta(\sigma) \tag{4}$$

We realize every returned plan with (4) by a legalization pass (Section IV-F) and evaluate the realized schedule. This charges solver time and model inference to the repaired schedule itself.

Metrics. Let C_j be the realized completion time of job j and $T_j = \max(0, C_j - d_j)$ its tardiness. Voluntary migrations M_{vol} count tasks of U that change node; total migrations add the $|H|$ forced moves. We use the repair cost

$$J(\sigma) = \Sigma_j T_j + \lambda_L \cdot N_{late} + \lambda_M \cdot M_{vol} \tag{5}$$

with $\lambda_L = 600$ s per late job and $\lambda_M = 60$ s per voluntary migration, so that a missed deadline weighs as ten minutes of tardiness and a migration as one minute. The recovery time is

$$RT = \max_{i \in H \cup Desc(H)} C_i - t_f \tag{6}$$

i.e., the time until the restarted work and all of its descendants have completed. We further report the deadline-miss rate (DMR) over jobs that are active at t_f, the feasibility rate (a plan satisfying (1)–(3) is returned within the budget) and the decision latency L measured as wall-clock time including model inference. All methods receive the same decision budget B and must return a plan within it.

Table 1 summarizes the notation used in the remainder of the paper.

Table 1. Main Notation

Symbol	Meaning
N, F	nodes of the cluster; nodes that fail at t_f
C_n, M_n	core and memory capacity of node n
p_i, c_i, m_i	duration, core demand and memory demand of task i
r_j, d_j, C_j	release time, deadline and completion time of job j
σ^0, σ	pre-failure plan and repair plan
H, U, P	forced tasks, unaffected pending tasks, repair set
$Desc(H)$, I	descendants of H; impacted descendants (8)
LF_i, LS_i	latest finish and latest start that meet the deadline (7)
L, $\hat{L}$, B	decision latency, commit horizon, decision budget
D, D_R	decision set beyond the horizon; decisions shown to the model
J, RT	repair cost (5) and recovery time (6)

4. The CC-LLM Architecture

4.1. Overview

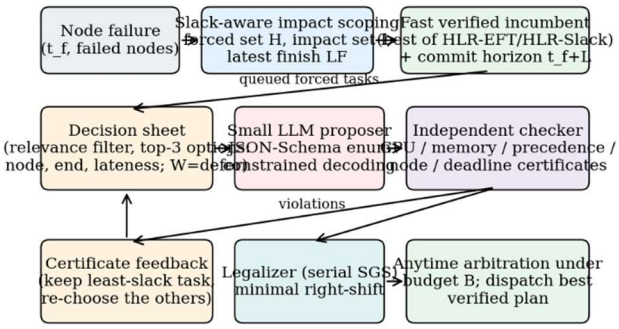


Fig. 1 Architecture of CC-LLM. Deterministic components (scoping, incumbent, checker, legalizer, arbiter) wrap a small language model that only proposes choices for queued decisions beyond the commit horizon.

Fig. 1 summarizes the pipeline. After a failure, the scoping stage derives the forced set, the deadline-driven latest finish times and the set of descendants that must move. A verified incumbent is computed in milliseconds and every decision that must be executed before the expected model latency is committed from it. Only queued forced tasks whose execution lies beyond this *commit horizon* are written into a decision sheet for the model. Each answer is checked by an independent checker; violation certificates are fed back for conflict-directed revisions, every candidate is legalized, and an anytime arbiter dispatches the best verified plan available when the budget expires.

4.2. Slack-Aware Impact Scoping

The scoping stage answers two questions: which decisions are forced by the failure, and how much delay each task can absorb. A backward pass over the not-yet-started tasks computes the latest finish time that still meets the job deadline,

$$LF_i = \min\{\, d_{j(i)},\ \min_{(i,v)\in E} (LF_v - p_v) \,\} \tag{7}$$

and a forward pass computes an optimistic finish time φ_i under the assumption that forced tasks restart at t_f on unlimited capacity while every other task keeps its planned start unless a predecessor finishes later. The impact set

$$I = \{\, i \in Desc(H) : \varphi_i > s_i^0 + p_i \,\} \tag{8}$$

contains the descendants that must be delayed even in the best case, whereas descendants outside I absorb the restart within their planned slack. Tasks outside $H \cup Desc(H)$ are independent of the failure through precedence and can only be affected through shared capacity. On our scenarios the analysis reduces 123 pending tasks to 22.7 forced tasks and 11.6 impacted descendants on average, out of 28.9 descendants. LF drives the lateness annotations of the decision sheets, I identifies the jobs at risk, and the closure $H \cup Desc(H)$ defines the scope of the local solver baseline.

4.3. Verified Fast Incumbent and Commit Horizon

Two classical repair rules are evaluated immediately: HLR-EFT reassigns forced tasks to the node with the earliest feasible finish in the order of their planned start, and HLR-Slack uses the latest start $LS_i = LF_i - p_i$ as priority. Both keep unaffected tasks on their nodes. The checker validates both and the one with the lower cost (5) becomes the incumbent σ_{inc}. Neither rule dominates: across our scenarios each one is better than the other on several instances, so the verified choice already removes the risk of committing to the weaker rule.

Let $\hat{L}$ be the latency horizon of the model stage (25 s in our configuration). Every task whose incumbent start lies before $t_f + \hat{L}$ is *committed*: it is dispatched immediately as planned by the incumbent and never waits for the model. The decision set D contains the forced tasks whose incumbent start is at or after $t_f + \hat{L}$, i.e., tasks that are queued behind other work anyway. D is ranked by slack $LS_i - s_i^{inc}$ and capped at $K = 8$ tasks. The commit horizon converts model latency from a delay into a planning lead time: as long as the model answers within $\hat{L}$, its decisions concern tasks that could not have started earlier.

Proposition 3 (latency hiding). If the model stage returns within the horizon, $L \le \hat{L}$, the latency-aware realization (4) leaves the dispatched CC-LLM plan unchanged, i.e., no task is delayed by more than the computation time of the incumbent. *Sketch:* committed tasks are fixed at their incumbent starts, which were decided within milliseconds; every task of D is legalized with a start not earlier than $t_f + \hat{L} \ge t_f + L$; and unaffected tasks keep their nodes and never start earlier than planned, so they are not changed in the sense of (4). A time-limited solver that releases its plan at $t_f + B$ enjoys no such guarantee: all of its changed decisions are shifted by its latency.

4.4. Decision Sheets and Schema-Constrained Proposals

For each $d \in D$ the sheet lists up to three options $o = (n, s, f)$, namely the surviving nodes with the earliest feasible finish on the frozen context (running tasks, committed tasks and unaffected tasks at their incumbent positions). Each option is annotated with the job lateness it implies,

$$\ell_d(o) = \max(0, f_o - LF_d) \tag{9}$$

and an additional option W defers the task to the lowest priority, which realizes the *postpone* action. Three action types are therefore available to the model: reassignment (the node of an option), order adjustment (the start implied by an option) and deferral (W). Times are expressed in units of $\tau = \lceil (\max f^{inc} - t_f)/100 \rceil$ seconds to keep numbers short, which matters because the tokenizer splits digits. A *decision-relevance filter* removes tasks whose options are equivalent, i.e., all options finish within one unit and carry the same lateness; such tasks are placed by sequential earliest-finish assignment without consulting the model. A typical line of a sheet reads a J0: A n3 end40 late0 | B n5 end97 late49 | C n4 end97 late49.

The model answers with a JSON object whose keys are the task identifiers. The response is decoded under a JSON Schema in which every key is required and its value is restricted to the enumeration {A, B, C, W} of the options listed for that task. Grammar-constrained decoding [22] guarantees that every answer names an existing option on an available node with sufficient capacity for the task in isolation, so that NODE and FIT violations cannot occur; the remaining violations stem from interactions among decisions and are left to the checker.

Example 1: Real decision sheet, schema, answer and feedback (SRASearch scenario, seed 0)

Round 1 – decision sheet written by CC-LLM

Node failed. Unit 18s, now 0.

Deadlines: J1:106 J3:45
a J3: A n1 end53 late11 | B n5 end63 late21 | C n6 end84 late43
b J1: A n1 end53 late0 | B n5 end63 late0 | C n6 end84 late0
c J1: A n1 end26 late0 | B n5 end36 late0 | C n7 end43 late0
Response schema (every key required, no other keys)
{"a":{"enum":["A","B","C","W"]}, "b":{...}, "c":{...}}
Model answer (Qwen2.5-Coder-0.5B, 1 core)
{"a": "B", "b": "C", "c": "A"}
Round 2 – certificate feedback after checking and legalization
Problems in your last answer:
J3 finishes 21 late
Choose again:
Deadlines: J3:45
a J3: A n5 end63 late21 | B n3 end89 late47 | C n7 end97 late55

Example 1 reproduces a real exchange. The first answer is conflict-free, but after legalization job J3 remains late, so the deadline certificate re-opens task *a* with options recomputed around the frozen decisions *b* and *c*. Neither revision improves on the incumbent, whose cost is 798 against 855 for HLR-EFT, so the arbiter dispatches the incumbent; the rejected rounds would have cost 967 and 1442. The exchange illustrates the division of labor: the model explores, the checker explains, and the arbiter decides.

4.5. Independent Checker and Violation Certificates

The checker validates the *literal* plan in which every chosen option is placed exactly as proposed. It sweeps the start and end events of each node and reports certificates of the form (CPU, n, t, S, excess) and (MEM, n, t, S, excess) for every overloaded set S, (PREC, u, v, gap) for every dependency violation, NODE and FIT for invalid placements and PAST for starts before t_f; after legalization it adds soft (DL, j, T_j) certificates for late jobs. Overloads arise because options are computed independently, so that two decisions may claim the same free slot.

Conflict-directed feedback. In every overloaded set the decision with the least slack keeps its slot and is frozen; the other members are re-chosen. Menus for the re-chosen tasks are recomputed on the context that contains all frozen decisions, so that the conflict cannot reappear in the same form. The feedback message lists the certificates in compact text (for example n5 cpu over by 2 cores at 42: b,c; keep b) followed by the new options; late jobs are listed as well and their decisions are re-opened. At most three rounds are performed.

The structured feedback reflects the value of actor–verifier separation highlighted by Qin et al. [20] and of diagnostic localization demonstrated by Zhang [21]: the overloaded task set, capacity excess or precedence gap identifies the specific interacting decisions that require revision, making the feedback directly actionable in the next proposal round.

4.6. Legalization and Verified Anytime Arbitration

Every candidate, i.e., the incumbent and the outcome of each round, is legalized by a serial schedule-generation scheme [13]: tasks are processed in priority order (the proposed start; W last) once all predecessors are placed, and each task is put on its assigned node at the earliest time at which precedence and the cumulative CPU and memory profiles admit it. Committed tasks keep their incumbent start, decision tasks start no earlier than $t_f + \hat{L}$, and unaffected tasks start no earlier than their planned start, which bounds disruption to right shifts.

Proposition 1 (feasibility). If every task fits on its assigned node in isolation, the legalizer returns a plan that satisfies (1)–(3). *Sketch:* tasks are placed in a topological order, each at a time not earlier than all predecessor finishes and t_f, and the profile query admits a start only if

the demand fits over the whole execution interval; placed intervals are never moved afterwards. Because the schema restricts options to nodes on which the task fits, the condition always holds.

Proposition 2 (no regret). The arbiter returns $\sigma_{out} = \text{argmin}_{\sigma \in Q(B)} J(\sigma)$, where $Q(B)$ contains the legalized incumbent and every round completed within the budget B. Hence $J(\sigma_{out})$ never exceeds the cost of the legalized incumbent, and the plan is available at any budget, including budgets shorter than one model call.

Algorithm 1: CC-LLM repair under decision budget B

1: Input: plan σ^0, failure (t_f, F), budget B, horizon $\hat{L}$, cap K, rounds R
2: Compute H, $Desc(H)$, LF by (7) and I by (8)
3: $\sigma_{inc} \leftarrow$ cheaper verified plan of HLR-EFT and HLR-Slack
4: Commit tasks of σ_{inc} that start before $t_f + \hat{L}$; $Q \leftarrow \{\text{legalize}(\sigma_{inc})\}$
5: $D \leftarrow K$ least-slack forced tasks starting after $t_f + \hat{L}$
6: Place decision-irrelevant tasks of D by sequential EFT; $D_R \leftarrow$ remaining tasks
7: for $r = 1, \ldots, R$ while elapsed time $< B$ do
  8: build options and lateness (9) for D_R; write sheet with certificates
  9: $a \leftarrow$ LLM(sheet) under JSON-Schema enum constraints
  10: $V \leftarrow$ Check(literal plan of a); $Q \leftarrow Q \cup \{\text{legalize}(a)\}$
  11: if $V = \emptyset$ and no job of D is late then break
  12: $D_R \leftarrow$ non-kept members of overload sets $\cup$ tasks of late jobs
13: return $\text{argmin}_{\sigma \in Q} J(\sigma)$, realized with (4)

Complexity. With S breakpoints per node profile, building the menus costs $O(|D|\cdot|N|\cdot S)$, a checker pass costs $O(|P| \log|P| + |E|)$ and a legalization costs $O(|P|(\log|P| + S))$. On our scenarios these deterministic steps take a few milliseconds, so the decision latency is dominated by model inference.

5. Experimental Setup

5.1. Workloads from Real Workflow Executions

All task graphs, runtimes and CPU utilizations come from WfInstances [12], the curated collection of real workflow executions of the WfCommons project [2], here recorded by Pegasus [3] on the Chameleon testbed. We use 20 execution instances of six applications (Table 2). Core demands are derived from the measured average CPU utilization (⌈avgCPU/100⌉, capped at four cores) and durations are the measured runtimes rounded to seconds. Measured peak memory is used where the instance records it (27% of the tasks); otherwise the memory demand is twice the total size of the task's recorded input files, clipped to [0.5, 12] GB. Seismology instances were excluded because their tasks run for less than one second.

Table 2. Workflow Instances Used (WfInstances, Pegasus)

Application	Domain	Inst.	Tasks	Median runtime (s)
Montage	Astronomy	4	58–178	0.3–2.9
Epigenomics	Bioinformatics	4	41–241	0.7–10.7
1000Genome	Bioinformatics	3	52–104	52–92
Cycles	Agroecosystems	3	67–219	4.3–26.2
SoyKB	Bioinformatics	2	96–156	94–119
SRASearch	Bioinformatics	4	22–42	75–91

5.2. Failure Scenarios

Each scenario combines $K \in [3, 5]$ executions of one application released at offsets drawn from [0, 0.3] times their critical-path length. The cluster has eight heterogeneous nodes holding 20%, 20% and six times 10% of a total core count sized to a target utilization $\rho \in [0.55, 0.85]$, with 4 GB of memory per core. The pre-failure plan is produced by ASAP-ordered list scheduling with earliest-finish placement, and deadlines are $d_j = r_j + (1 + \beta_j)(C_j^0 - r_j)$ with $\beta_j \in [0.05, 0.30]$, so that all deadlines hold before the failure. The failure time is drawn from busy periods within [0.1, 0.7] of the makespan, and the failed node is drawn among nodes that still hold at least three unfinished tasks. We generate three scenarios per application (18 in total) for the main study. Table 3 summarizes them; the average utilization at failure is 0.89, i.e., failures hit a loaded cluster.

Table 3. Statistics of the 18 Single-Node Failure Scenarios

Quantity	Mean	Range
Tasks per scenario	347	88–721
Pending tasks at failure |P|	123.4	–
Forced tasks |H|	22.7	4–61
Descendants of H	28.9	–
Impacted descendants |I|	11.6	–
Jobs per scenario	3.8	3–5
Total cores	54.8	–
Utilization at failure	0.89	–

5.3. Compared Methods

HLR-EFT and **HLR-Slack** are the heuristic local repairs of Section IV-C. **Full-Resched.** reschedules every pending task with earliest-finish placement and latest-start priority, ignoring the previous assignment. **CP-SAT-G** solves the global repair of all pending tasks and **CP-SAT-L** solves the local repair of $H \cup Desc(H)$ with all other tasks fixed; both use optional interval variables per node, cumulative CPU and memory constraints, precedence constraints and objective (5) with a small completion-time term, are warm-started from the existing schedule (hints for all non-forced tasks plus hint repair) and run with one worker and a time limit equal to the budget. **Menu-Greedy** and **Menu-Random** run the CC-LLM pipeline with the model replaced by the rule "minimum lateness, then earliest finish" and by uniformly random choices. **Unchecked LLM** executes the first model answer literally with a dispatcher that only waits for dependencies. **CC-LLM** is the full system; ablations remove one component at a time.

The CP-SAT models give every scoped task an integer start variable S_i and, for each surviving node n on which it fits, a Boolean $x_{i,n}$ that activates an optional interval of length p_i, with $\Sigma_n x_{i,n} = 1$. Two cumulative constraints per node bound cores and memory and contain fixed intervals for running tasks and for tasks outside the scope. Precedence constraints link scoped tasks and fixed predecessors raise the lower bounds of their successors. The objective $100 \cdot \Sigma_j T_j + 60000 \cdot \Sigma_j \text{late}_j + 6000 \cdot \Sigma_i (1 - x_{i,n0(i)}) + \Sigma_i E_i$ is the integer form of (5) with a completion-time tie-breaker, where $n0(i)$ is the pre-failure node and E_i the end variable. Every incumbent found by the solver is stored with its timestamp, which yields the anytime trajectory used for the budget study.

5.4. Implementation and Platform

All experiments run on a single core of an Intel Xeon processor at 2.1 GHz with 3 GB of memory and no accelerator, which represents an on-premise management node. The proposer is Qwen2.5-Coder-0.5B-Instruct [23] served by the goinfer runtime v0.23.0 [25] with int8 weights

and activations, greedy decoding and JSON-Schema-constrained output; the model-scale study uses Qwen2.5-Coder-1.5B-Instruct in the same runtime. CP-SAT is OR-Tools 9.15 [24]. The main budget is B = 20 s; the budget study uses 5, 10 and 20 s, CP-SAT incumbents are recorded with their timestamps, and the anytime plan of every method at budget B is the best plan available at B. CC-LLM uses $\hat{L}$ = 25 s, K = 8, three options and at most three rounds. Runtime perturbation draws every pending duration from ±20% around its nominal value in six common random draws per scenario and executes each plan with an event-driven dispatcher that keeps its node assignment and order.

Statistical protocol. All methods are evaluated on identical scenarios, so all comparisons are paired. We report means together with medians because the cost distribution is heavy-tailed, count wins, ties and losses with a tie tolerance of 1 s, which is the time resolution of the traces, and test paired differences with the two-sided Wilcoxon signed-rank test. Runtime perturbations use common random numbers, so every method faces the same six perturbed duration vectors in each scenario.

6. Results

6.1. Main Comparison

Table 4. Main Results over 18 Single-Node Failure Scenarios at an Equal Decision Budget B = 20 s

Method	Cost J (mean)	Cost J (median)	Tardiness (s)	DMR (%)	Migrations (total / voluntary)	RT (s)	Feasible (%)	Latency mean / max (s)	J under ±20% runtimes
HLR-EFT	1241.5	615.5	708.2	30.5	22.7 / 0.0	2044	100	<0.01 / <0.01	1344.7
HLR-Slack	1560.9	704.0	927.6	36.5	22.7 / 0.0	2047	100	<0.01 / <0.01	1606.0
Full-Resched.	3316.8	2790.0	106.8	18.9	70.7 / 47.9	1588	100	<0.01 / <0.01	3429.2
CP-SAT-G (warm start)	1278.3	679.5	698.3	30.0	22.9 / 0.2	2085	94.4	16.7 / 20.0	1391.3
CP-SAT-L (warm start)	845.0	637.5	301.7	25.4	24.0 / 1.3	1746	100	10.4 / 20.0	950.0
Menu-Greedy	1202.4	610.0	702.4	29.1	22.7 / 0.0	2042	100	<0.01 / <0.01	–
Menu-Random	1122.9	610.0	622.9	29.1	22.7 / 0.0	2075	100	<0.01 / <0.01	–
CC-LLM w/o arbitration	875.1	623.5	375.1	29.1	22.7 / 0.0	1737	100	6.4 / 19.1	–
CC-LLM w/o feedback	777.1	610.0	310.4	27.2	22.7 / 0.0	1680	100	3.0 / 9.3	–
CC-LLM (ours)	**777.1**	**610.0**	310.4	27.2	**22.7 / 0.0**	**1680**	**100**	6.4 / 19.1	950.9

Cost J follows (5) in seconds; DMR is the deadline-miss rate over active jobs; RT is the recovery time (6); latency includes model inference. Forced migrations (22.7 on average) are identical for all stable methods.

Table 4 and Fig. 2 report the main comparison. CC-LLM attains the lowest mean repair cost, 777.1, which is 37.4% below HLR-EFT, 50.2% below HLR-Slack and 8.0% below the warm-started local CP-SAT, and it ties the best median cost (610.0). Its tardiness (310.4 s) is 56% below heuristic local repair and within 3% of CP-SAT-L, while it migrates no unaffected task, whereas CP-SAT-L moves 1.3 and full rescheduling 47.9 unaffected tasks per scenario on average. The recovery time of 1680 s is 18% shorter than that of HLR-EFT and the shortest among all methods that do not move unaffected tasks; only full rescheduling recovers faster (1588 s), at the price of 3.1× the total migrations and a mean cost of 3316.8. Every CC-LLM plan passed the independent checker, while CP-SAT-G found no solution for one SoyKB scenario within the budget.

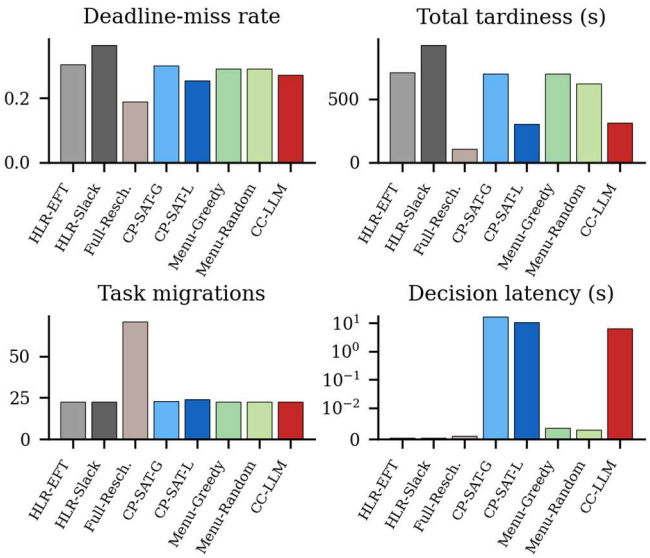


Fig. 2 Main comparison at B = 20 s over 18 scenarios: deadline-miss rate, total tardiness, task migrations (forced plus voluntary) and decision latency (symmetric log scale).

Paired comparisons over the 18 scenarios (differences below 1 s counted as ties) show that CC-LLM is never worse than either fast heuristic: against HLR-EFT it wins 7, ties 11 and loses none (Wilcoxon signed-rank p = 0.041), and against HLR-Slack it wins 4 and ties 14. Against full rescheduling it wins 17 of 18 scenarios (p < 0.001). Against the two CP-SAT variants it wins 9 and 8 scenarios and loses 3 each (p = 0.12 and 0.24); the solver variants are strongest on the Montage scenarios, while CC-LLM is strongest on SoyKB, Epigenomics and SRASearch (Section VI-H). The mean decision latency of CC-LLM is 6.4 s, 39% below CP-SAT-L (10.4 s) and 62% below CP-SAT-G (16.7 s).

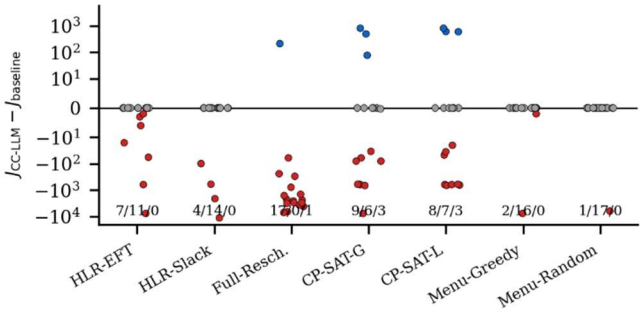


Fig. 3 Per-scenario cost difference between CC-LLM and each baseline (symmetric log scale; negative values favour CC-LLM). Labels give wins/ties/losses of CC-LLM with a 1 s tie tolerance.

Fig. 3 shows the paired differences per scenario. The losses against the solvers are concentrated in the Montage scenarios, whereas the largest gains appear against full rescheduling, whose cost is dominated by voluntary migrations, and on the SoyKB scenarios, where a single verified model proposal changes the outcome by several thousand cost units.

The model contributes beyond the deterministic layers. The legalized incumbent alone has a mean cost of 1203.3, and replacing the model by the greedy or random menu policy yields 1202.4 and 1122.9. The model's proposals reduce the mean cost to 777.1; the largest single gain occurs on a SoyKB scenario, where the accepted proposal lowers tardiness by 68% and removes one late job (Section VI-H), while the greedy menu policy reproduces the incumbent and the random policy reaches 10146 instead of 3922 on the same scenario. Without

arbitration, i.e., dispatching the last model round regardless of its cost, the mean cost is 875.1, still below every heuristic and the global solver, which confirms that verified selection and model proposals are complementary.

6.2. Feasibility and the Role of Checking

The commit horizon and the relevance filter resolved all decisions in 9 of the 18 scenarios without consulting the model; in the remaining 9 the model was called 23 times with 3.8 decisions per call, 590-character prompts, 34.7 output tokens and 5.8 s per call on average, and every response parsed under the schema. Fig. 4 shows why checking is indispensable. Executed literally by a dispatcher that only respects dependencies, the first model answer was feasible in only 44% of the consultations, with 2.4 hard violations per scenario, all of them CPU or memory overloads created by decisions that claimed the same free slot. The greedy menu policy, which sends every task to its individually best option, is feasible in only 11% of the cases. Conflict-directed feedback reduced the number of conflicting decisions from 2.67 in the first round to 0.89 in the last round (−67%), and a conflict-free literal plan was reached within three rounds in 89% of the consultations. After legalization, 100% of the dispatched plans satisfied all hard constraints.

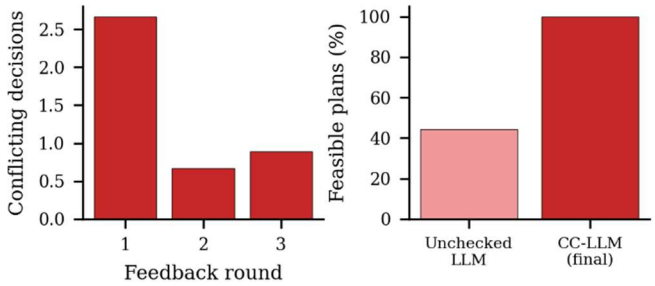


Fig. 4 Effect of independent checking in the scenarios in which the model was consulted: (left) conflicting decisions per feedback round; (right) share of feasible plans for literally executed model output and for CC-LLM.

The model's choices differ markedly from the greedy rule: it selected the earliest-finishing option A in only 5.3% of the first-round decisions and the deferral option W in 11.1%. This diversity is what produces non-myopic proposals such as the SoyKB repair, and it is also what makes unchecked execution unsafe. The checker turns this exploratory behavior into an asset: infeasible combinations are repaired or rejected, and only verified improvements are dispatched (the arbiter accepted a model plan in 22% of the consultations).

6.3. Decision Latency and Budget Sensitivity

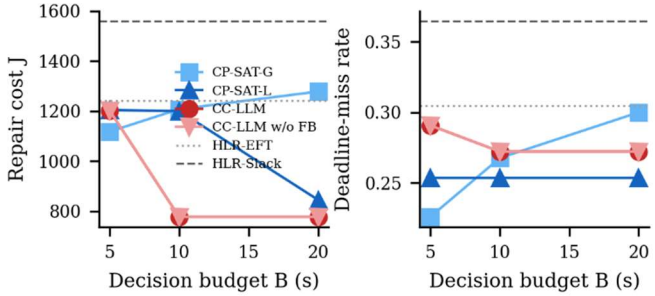


Fig. 5 Repair cost and deadline-miss rate versus the decision budget B. Horizontal lines mark the heuristics, which need less than 10 ms.

Fig. 5 varies the budget. At B = 5 s the model has not answered yet and CC-LLM dispatches its verified incumbent (cost 1202.3, latency 0.8 s); at 10 s the first round is available and the cost drops to 777.1 with a mean latency of 3.6 s; at 20 s the cost is unchanged while additional feedback rounds are used to remove conflicts. The solvers show the opposite trend for the global model: the cost of CP-SAT-G rises from 1117.9 at 5 s to 1278.3 at 20 s, because each improvement of the solver objective is outweighed by the later release of the plan under (4). CP-SAT-L improves from 1204.2 to 845.0 as it proves optimality on half of its local models, but it still needs 10.4 s on average. Both CP-SAT variants returned no solution for one scenario at budgets of 5 s and 10 s. Fig. 6 shows the latency distributions: half of the CC-LLM decisions finish within milliseconds because no model call is needed, and the remainder stay below 19.1 s.

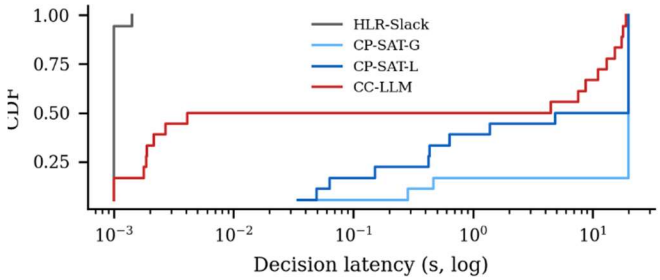


Fig. 6 Empirical CDF of the decision latency per scenario (log scale).

The latency results expose a structural advantage of the commit horizon. A solver that releases its plan at the end of its budget delays every forced task by that budget, whereas CC-LLM dispatches all urgent decisions at once and uses the budget only for decisions whose execution lies after the horizon. The comparison between the variants with and without feedback illustrates the same point from the inside: the additional rounds raised the mean latency from 3.0 s to 6.4 s without increasing the cost, because they act on queued tasks.

6.4. Ablation Study

Table 5. Ablation of CC-LLM Components (B = 20 s)

Variant	J	Raw J	Prompt (chars)	Latency (s)	Conflicts
All 18 scenarios					
CC-LLM (full)	777.1	875.1	590	6.4	2.67→0.89
w/o certificate feedback	777.1	–	590	3.0	2.67
w/o arbitration	875.1	875.1	590	6.4	–
proposer → greedy rule	1202.4	–	–	<0.01	4.11
proposer → random choice	1122.9	–	–	<0.01	0.78
incumbent only	1203.3	–	–	<0.01	–
Six seed-0 scenarios					
CC-LLM (full)	619.2	793.8	577	5.6	3.0
w/o lateness annotations	619.2	726.5	440	4.8	1.0
w/o relevance filter	621.8	729.2	1224	17.6	12.0
w/o commit horizon	631.8	994.5	547	6.0	3.6
w/o commit horizon, greedy	732.0	–	–	<0.01	–

Raw J dispatches the last model round without arbitration. Latency refers to one model call except for the full-pipeline rows of the upper block (end-to-end mean). Conflicts are conflicting decisions per consulted scenario.

Table 5 isolates the components. The upper block uses all 18 scenarios. The lower block runs separate model executions on the six seed-0 scenarios, in which the model was consulted in

four to five scenarios depending on the variant; it reports the verified cost, the raw cost of the last model round, the prompt size, the model latency of the first round and the number of conflicting decisions.

Arbitration is the most important safeguard: dispatching the last model round raises the cost from 777.1 to 875.1 on all scenarios and from 619.2 to 793.8 on the subset. *The commit horizon* is the most important latency mechanism: without it the model decides the most urgent forced tasks, which then wait for the answer, and the raw cost rises to 994.5 and the deadline-miss rate from 25% to 29%; with the greedy rule instead of the model the variant without horizon reaches 732.0, i.e., waiting is costly even when deciding is free. *The relevance filter* keeps the sheet small: without it the model receives 15 instead of 4.5 decisions per call, the prompt grows 2.1×, the first-round latency grows from 5.6 s to 17.6 s and the number of conflicts quadruples, which exhausts most of the budget in a single round. *Certificate feedback* does not change the final cost on these scenarios but removes two thirds of the conflicts at the price of 3.4 s of additional latency that is entirely hidden behind the horizon. On the six-scenario subset the 0.5B proposer did not exploit the *lateness annotations* measurably: the verified cost is identical and the shorter sheet without annotations produced fewer conflicts.

The conflict reduction achieved by certificate feedback agrees with the substantial gains that precise diagnostics deliver in the controlled repair comparisons of Qin et al. [20] and Zhang [21], which underlines the generality of localized verifier feedback across application domains.

6.5. Scoping and Prompt Economy

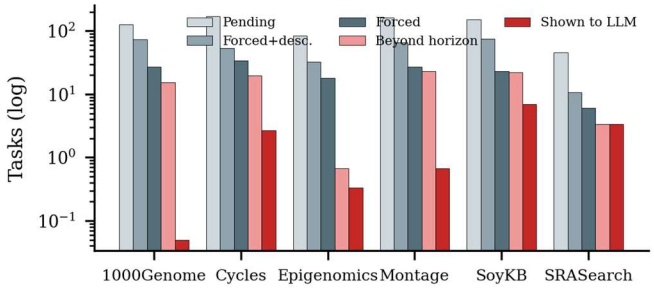


Fig. 7 Scoping funnel per application (mean over scenarios, log scale): pending tasks, forced tasks plus descendants, forced tasks, forced tasks beyond the commit horizon and decisions finally shown to the model.

Fig. 7 traces the decision scope. Of 123.4 pending tasks, 51.6 are forced or descendants of forced tasks and 22.7 are forced; 14.1 forced tasks lie beyond the commit horizon, 4.4 remain after the slack-ranked cap and 2.3 after the relevance filter. The model therefore sees a 98% smaller problem than a global re-planner, which is what makes a CPU-only 0.5B model fast enough. The funnel is application dependent: SRASearch, with few long tasks, has the narrowest scope, whereas wide Montage and Cycles stages produce many forced tasks whose options are equivalent and are placed without the model.

Whether the model is consulted depends on the structure of the workload rather than on the size of the forced set. All three SoyKB scenarios and two of the three Cycles and SRASearch scenarios produced queued forced tasks with distinguishable options, because long tasks occupy the surviving nodes beyond the horizon. No 1000Genome scenario required the model, even with 61 forced tasks: the restarted tasks either fit into immediate gaps and were committed, or all of their options were equivalent. The pipeline therefore spends inference time only where the residual capacity leaves genuine choices.

6.6. Robustness

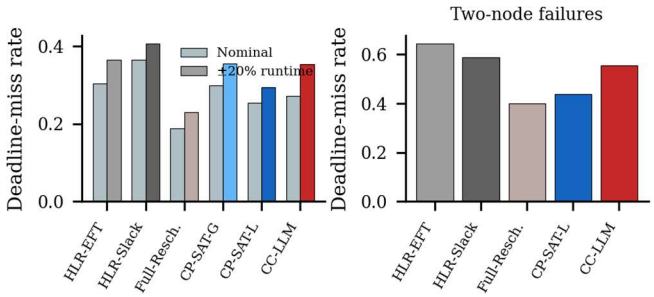


Fig. 8 Robustness: (left) deadline-miss rate under nominal runtimes and under ±20% runtime perturbation (event-driven execution); (right) deadline-miss rate under simultaneous two-node failures.

Runtime perturbation. When the dispatched plans are executed with durations perturbed by up to ±20%, the mean cost of CC-LLM becomes 950.9, practically equal to CP-SAT-L (950.0) and 29% below HLR-EFT (1344.7); full rescheduling remains the most expensive plan (3429.2). The deadline-miss rate of all methods rises by four to eight points under perturbation (Fig. 8, left), and the ranking of the stable methods is preserved.

Two-node failures. Table 6 repeats the comparison on six scenarios in which two nodes fail simultaneously, doubling the forced set to 45.2 tasks on average. CC-LLM again returns feasible plans in all scenarios without voluntary migrations and reduces the cost by 10.3% relative to HLR-EFT and by 25.1% relative to HLR-Slack. CP-SAT-L achieves a lower mean cost when it succeeds, but it returns no solution within the budget in one of the six scenarios and needs 16.7 s on average, versus 8.3 s for CC-LLM. Full rescheduling attains the lowest miss rate at the price of 65.7 voluntary migrations.

Table 6. Two-Node Failures (6 Scenarios, B = 20 s)

Method	J (median)	DMR (%)	Vol. mig.	Feas. (%)	Lat. (s)
HLR-EFT	2287.7 (1898.5)	64.4	0.0	100	<0.01
HLR-Slack	2740.5 (1470.0)	58.9	0.0	100	<0.01
Full-Resched.	5149.8 (4453.5)	40.0	65.7	100	<0.01
CP-SAT-L	1922.5 (1090.0)	43.9	0.2	83.3	16.7
CC-LLM	2051.5 (1470.0)	55.6	0.0	100	8.3

6.7. Sensitivity to Model Scale

Table 7 replaces the 0.5B proposer by the 1.5B model of the same family on the six seed-0 scenarios. Because one 1.5B call takes 35.5 s on our single core, the horizon is raised to $\hat{L}$ = 75 s, one round is allowed and the budget is 90 s; the 0.5B model is run with the same settings for comparison. The larger model behaves much closer to the greedy rule: it picks the earliest-finishing option in 75% of its decisions (0.5B: 0%), its literal plans are feasible twice as often (66.7% versus 33.3%) and its raw cost equals that of the greedy rule (619.2 versus 719.0 for the 0.5B model). After checking and arbitration all variants return the same verified cost on these scenarios. Scale thus improves the reliability of raw proposals, while the smaller model remains attractive because it answers 7.1× faster and its more diverse proposals are made safe by the checker.

Table 7. Model-Scale Study (6 Scenarios, $\hat{L}$ = 75 s, B = 90 s, One Round)

Proposer	Lat./call (s)	Option A (%)	Unchecked feasible (%)	Raw J	Final J
Qwen2.5-Coder-0.5B	5.0	0	33.3	719.0	619.2
Qwen2.5-Coder-1.5B	35.5	75	66.7	619.2	619.2
Greedy rule	<0.01	100	–	619.2	619.2
Random choice	<0.01	–	–	651.2	619.2

The verified incumbent costs 621.8 on these scenarios; both models were consulted in three of the six scenarios.

6.8. Solver and Proposer Behaviour

Table 8 characterizes the warm-started solvers. The local scope shrinks the model from 123.4 to 51.7 tasks on average and lets CP-SAT prove optimality in half of the scenarios, against one sixth for the global model. Both solvers find a first solution quickly (median 0.20 s and 0.49 s), but the global model keeps improving its objective until the time limit in most scenarios and therefore releases its plan late, which is the mechanism behind the rising cost of CP-SAT-G in Fig. 5. The global model also failed to find any solution for one of the SoyKB scenarios.

Table 8. Behaviour of the Warm-Started CP-SAT Baselines (18 Scenarios)

Solver	Tasks in model	Incumbents	First sol. (s, median)	Optimal (%)	Unsolved at 20 s
CP-SAT-G	123.4 (15–305)	56.7	0.49	16.7	1
CP-SAT-L	51.7 (7–149)	86.9	0.20	50.0	0

Table 9 reports the proposer per feedback round. Later rounds re-open fewer decisions (4.7, 3.5 and 3.0) and stay within about 6 s per call. Conflicts fall from 2.67 in the first round to 0.75 in the second; the third round is reached only by the six consultations with persistent deadline certificates, and its conflict count (1.33) refers to this harder subset. Output length stays near 35 tokens because the schema admits only one letter per decision.

Table 9. Proposer Statistics per Feedback Round (Main Study)

Round	Consult.	Decisions	Prompt (chars)	Output tokens	Latency (s)	Conflicts
1	9	4.67	590	34.7	6.00	2.67
2	8	3.50	667	35.1	6.45	0.75
3	6	3.00	538	23.0	4.63	1.33

6.9. Per-Application Analysis and Case Study

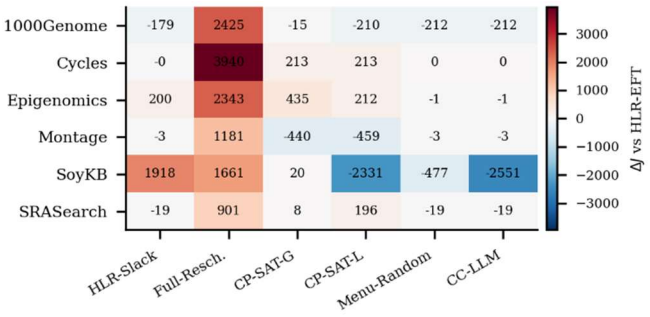


Fig. 9 Mean cost difference to HLR-EFT per application (negative values are better).

Fig. 9 breaks the cost down by application. On Montage, whose wide stages of short tasks create many interchangeable decisions, the local solver is clearly superior (mean cost 835 versus 1291)

because it can re-sequence entire stages. On SoyKB, with long alignment and variant-calling tasks, CC-LLM achieves the lowest cost (1307 versus 1527 for CP-SAT-L and 3859 for HLR-EFT). On 1000Genome, Epigenomics, Cycles and SRASearch, CC-LLM matches or improves the best heuristic, and on Epigenomics, Cycles and SRASearch it is better than both CP-SAT variants, whose plans are released only at the end of their budget.

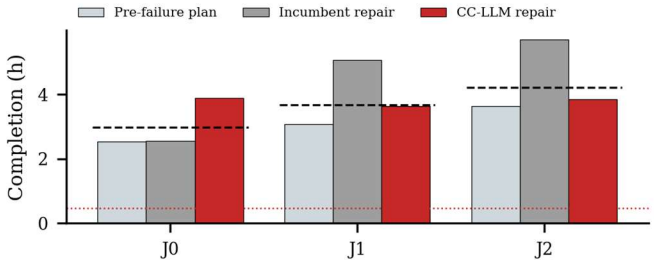


Fig. 10 Case study (SoyKB, seed 1): job completion times of the pre-failure plan, the verified incumbent repair and the dispatched CC-LLM repair. Dashed lines mark the deadlines and the dotted line the failure time.

Fig. 10 shows the SoyKB scenario in which the model made the decisive contribution. A node fails at 1709 s while three jobs are running; the incumbent repair restarts the forced tasks greedily and delays jobs J1 and J2 by 5028 s and 5348 s beyond their deadlines. The model's sheet contained five decisions with a time unit of 189 s. Instead of the locally best option for task *a* of job J0 (node n3, lateness 0), the model chose a later slot on node n5 and spread the remaining tasks over other nodes. The checker found no conflict, and after legalization J1 and J2 finish before their deadlines while J0 is late by 3322 s. Total tardiness drops from 10376 s to 3322 s, the number of late jobs from two to one and the recovery time from 18823 s to 12318 s. The greedy menu policy, which follows the per-task annotations, reproduces the incumbent, and the warm-started local solver reaches a cost of 4582 versus 3922 for CC-LLM. The case illustrates the value of deferral decisions whose benefit is invisible in per-task annotations but becomes evident after verification.

7. Discussion

Where the gains come from. The results separate three sources of improvement. First, verification turns two cheap heuristics into a stronger incumbent, because neither HLR-EFT nor HLR-Slack dominates the other. Second, scoping and the commit horizon decouple the urgent part of the recovery, which must be decided at once, from the queued part, which can wait for a better decision; this is why CC-LLM does not pay for its latency the way a time-limited solver does. Third, the model explores combinations of reassignment, ordering and deferral that per-task rules do not consider, and the checker and arbiter guarantee that such exploration can only help. The answer to our research question is therefore affirmative under a precise condition: the model must be confined to decisions that lie beyond its latency horizon, and every proposal must pass an independent check before dispatch.

Quantitatively, verified selection between the two repair rules lowers the mean cost from 1241.5 (HLR-EFT) and 1560.9 (HLR-Slack) to 1203.3 for the legalized incumbent. Replacing the model by the greedy or random menu policy moves it only to 1202.4 and 1122.9, whereas the model's proposals bring it to 777.1. Of the total improvement over HLR-EFT, 92% is therefore realized by the model stage, concentrated in the scenarios in which non-myopic deferrals pay off, while the deterministic layers guarantee that no scenario becomes worse.

Fewer migrations by construction. All forced tasks must move in any repair, so the meaningful quantity is voluntary migration. CC-LLM keeps it at zero because its action space contains only forced decisions and deferrals, whereas global optimizers trade migrations for tardiness unless they are explicitly penalized, and even then move unaffected tasks occasionally. Operators can widen the action space, e.g., by exposing at-risk descendants as additional decisions, and the same checker and arbiter apply unchanged.

Deployment. All measurements were taken on one CPU core, the most demanding setting for inference. A GPU or a larger CPU allocation shortens model latency by more than an order of magnitude, which would allow a shorter horizon, more rounds or a larger model within the same budget; the architecture itself needs no change because the horizon is a parameter. The deterministic components require only the cluster state that every scheduler already maintains, and the certificates double as an audit log that explains why a repair was accepted or rejected. Natural-language operator policies, such as protecting a tenant or avoiding a draining node, can be added to the sheet without changing the checker.

Reproducibility. Every number in this paper is computed from released artifacts: the scenario generator with fixed seeds, the raw per-scenario JSONL records, the model response cache with the measured latency of every call, the CP-SAT incumbent traces and the analysis script that produces all tables and figures. Greedy decoding and a single solver worker make the pipeline deterministic up to wall-clock timing, and replaying the cached responses reproduces every plan without re-running the model.

Extensions. The study uses permanent node failures, restart-from-scratch semantics and no data-transfer delays. Checkpointed restarts, correlated rack-level failures, transfer costs and repeated failures are natural extensions; they change the checker and the legalizer but not the division of labor between proposer and verifier. Combining the model's proposals with the local solver, for example as warm starts, is a promising direction given that the two approaches excel on different applications.

Uncertainty-aware forecasting as developed in PRESAGE [16] is a particularly promising complement: predicted load and runtime quantiles could set the commit horizon and the lateness annotations adaptively, combining proactive and reactive scheduling in one loop.

8. Conclusion

We presented CC-LLM, a constraint-checked planning architecture for recovering cluster schedules after node failures with a small language model. Slack-aware scoping, a verified incumbent and a commit horizon keep urgent decisions off the model's critical path; schema-constrained decision sheets, an independent checker with violation certificates, conflict-directed feedback, legalization and anytime arbitration make every dispatched plan feasible and never worse than the incumbent. On failure scenarios built from real executions of six scientific workflows and under equal decision budgets, a 0.5B-parameter model on a single CPU core produced feasible repairs in all cases, migrated no unaffected task, achieved the lowest mean repair cost of all compared methods and an 18% shorter recovery time than heuristic local repair, with a mean decision latency of 6.4 s. These results indicate that language models can be used safely and productively in the recovery loop of cluster schedulers when they act as proposers inside a verified, latency-aware control structure.

References

[1] Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., & Wilkes, J. (2015). Large-scale cluster management at Google with Borg. In Proceedings of the 10th European Conference on Computer Systems (EuroSys).

[2] Coleman, T., Casanova, H., Pottier, L., Kaushik, M., Deelman, E., & Ferreira da Silva, R. (2022). WfCommons: A framework for enabling scientific workflow research and development. Future Generation Computer Systems, 128, 16–27.

[3] Deelman, E., Vahi, K., Juve, G., Rynge, M., Callaghan, S., Maechling, P. J., Mayani, R., Chen, W., Ferreira da Silva, R., Livny, M., & Wenger, K. (2015). Pegasus, a workflow management system for science automation. Future Generation Computer Systems, 46, 17–35.

[4] Schroeder, B., & Gibson, G. A. (2010). A large-scale study of failures in high-performance computing systems. IEEE Transactions on Dependable and Secure Computing, 7(4), 337–350.

[5] Vieira, G. E., Herrmann, J. W., & Lin, E. (2003). Rescheduling manufacturing systems: A framework of strategies, policies, and methods. Journal of Scheduling, 6(1), 39–62.

[6] Herroelen, W., & Leus, R. (2005). Project scheduling under uncertainty: Survey and research potentials. European Journal of Operational Research, 165(2), 289–306.

[7] Topcuoglu, H., Hariri, S., & Wu, M.-Y. (2002). Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE Transactions on Parallel and Distributed Systems, 13(3), 260–274.

[8] Sakellariou, R., & Zhao, H. (2004). A low-cost rescheduling policy for efficient mapping of workflows on grid systems. Scientific Programming, 12(4), 253–262.

[9] Kambhampati, S., Valmeekam, K., Guan, L., Verma, M., Stechly, K., Bhambri, S., Saldyt, L., & Murthy, A. (2024). Position: LLMs can't plan, but can help planning in LLM-Modulo frameworks. In Proceedings of the 41st International Conference on Machine Learning (ICML) (Vol. 235, pp. 22895–22907). PMLR.

[10] Valmeekam, K., Marquez, M., Olmo, A., Sreedharan, S., & Kambhampati, S. (2023). PlanBench: An extensible benchmark for evaluating large language models on planning and reasoning about change. In Advances in Neural Information Processing Systems 36 (NeurIPS), Datasets and Benchmarks Track.

[11] Abgaryan, H., Harutyunyan, A., & Cazenave, T. (2024). LLMs can schedule [Preprint]. arXiv. https://arxiv.org/abs/2408.06993

[12] WfCommons Project. (n.d.). WfInstances: Collection of open access production workflow executions [GitHub repository]. https://github.com/wfcommons/WfInstances

[13] Kolisch, R. (1996). Serial and parallel resource-constrained project scheduling methods revisited: Theory and computation. European Journal of Operational Research, 90(2), 320–333.

[14] Grandl, R., Ananthanarayanan, G., Kandula, S., Rao, S., & Akella, A. (2014). Multi-resource packing for cluster schedulers. In Proceedings of the ACM SIGCOMM.

[15] Mao, H., Schwarzkopf, M., Venkatakrishnan, S. B., Meng, Z., & Alizadeh, M. (2019). Learning scheduling algorithms for data processing clusters. In Proceedings of the ACM SIGCOMM.

[16] Wang, Y., Zou, X., Lu, Z., Wei, K., & Rhee, M. (2026). PRESAGE: Uncertainty-aware predictive elastic scheduling for cloud analytics and GenAI inference. IEEE Access, 14, 146019–146040. https://doi.org/10.1109/ACCESS.2026.3734498

[17] Romera-Paredes, B., Barekatain, M., Novikov, A., Balog, M., Kumar, M. P., Dupont, E., Ruiz, F. J. R., Ellenberg, J. S., Wang, P., Fawzi, O., Kohli, P., & Fawzi, A. (2024). Mathematical discoveries from program search with large language models. Nature, 625, 468–475.

[18] Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., & Yao, S. (2023). Reflexion: Language agents with verbal reinforcement learning. In Advances in Neural Information Processing Systems 36 (NeurIPS).

[19] Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., Alon, U., Dziri, N., Prabhumoye, S., Yang, Y., Gupta, S., Majumder, B. P., Hermann, K., Welleck, S., Yazdanbakhsh, A., & Clark, P. (2023). Self-Refine: Iterative refinement with self-feedback. In Advances in Neural Information Processing Systems 36 (NeurIPS).

[20] Qin, Y., Rhee, M., Teng, D., Zhang, C., & Zi, B. (2026). Neuro-symbolic constraint verification for LLM-driven internet finance transaction execution. Scientific Reports. https://doi.org/10.1038/s41598-026-65233-w

[21] Zhang, H. (2026). A verification-guided multi-agent framework for reliable RTL generation and optimization. Transactions on Computer Science and Intelligent Systems Research, 13, 260–274. https://doi.org/10.62051/r6y8jj34

[22] Geng, S., Josifoski, M., Peyrard, M., & West, R. (2023). Grammar-constrained decoding for structured NLP tasks without finetuning. In Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP).

[23] Hui, B., Yang, J., Cui, Z., Yang, J., Liu, D., Zhang, L., Liu, T., Zhang, J., Yu, B., Lu, K., et al. (2024). Qwen2.5-Coder technical report [Preprint]. arXiv. https://arxiv.org/abs/2409.12186

[24] Perron, L., & Furnon, V. (n.d.). OR-Tools (Version 9.15) [Software]. Google. https://developers.google.com/optimization/

[25] goinfer. (n.d.). Single-binary local LLM inference runtime with JSON-Schema-constrained decoding (Version 0.23.0) [GitHub repository]. https://github.com/townsendmerino/goinfer