Tractable Online Multi-Agent Coordination in Dense Multi-Level Warehouses

Decomposition, Localized Replanning,
and Compositional Safety–Liveness Guarantees

Architecture Working Document — March 23, 2026

Abstract

Coordinating tens of robots in a dense, multi-level warehouse requires solving an *online* variant of Multi-Agent Path Finding (MAPF) where goals arrive, change, and cancel during execution under exogenous disturbances. Optimal MAPF is NP-hard even in static settings [1]; the online variant compounds this with frequent, low-latency replanning. We formalize the problem, then argue for a two-axis decomposition: *vertical decomposition* separates demand modeling and carrier-level coordination from bot-level pathfinding, while *localized replanning* confines each bot-level replan to the affected subgraph of the current bot-level plan. We show that the resulting layered architecture admits a compositional correctness argument structured as an assume-guarantee composition [12]: a core safety invariant (collision-freedom) holds unconditionally at all reachable states; liveness (starvation-freedom) holds conditional on well-formed infrastructure [11] and bounded hardware execution, grounding out in physical facts rather than circular software assumptions.

1 Introduction

MAPF asks: given n agents on a graph, each with a start and goal, find collision-free paths for all agents. Real warehouse systems face a harder variant: goals arrive *online*, agents share bottleneck resources (elevators, narrow corridors), and safety invariants must hold continuously under exogenous disturbances—robot faults, elevator failures, demand cancellations, and battery alerts. An upstream *Demand Model* maintains a bipartite graph of active demands (unsatisfied SKU line items or carrier-pin requests) and carrier supplies (available inventory), exposing scored candidate sets to a *Carrier Coordinator* that jointly selects carriers and plans rearrangements. Demands may arrive, change, or cancel during execution. Replanning all agents from scratch at each event is untenable: even high-performance bounded-suboptimal solvers [4] incur prohibitive per-invocation cost at warehouse scale, and lifelong approaches such as RHCR [3] that amortize this cost still face full-replan overheads on large graphs.

We argue that tractability requires *reducing what must be solved and how often*, via two complementary strategies: (1) **vertical decomposition** factors the problem into layers with different change frequencies, and (2) **localized replanning** (bot-level plan only) confines each replan to the minimal affected subgraph. Section 2 formalizes the problem. Section 3 develops the vertical decomposition and composes the end-to-end assume-guarantee argument.

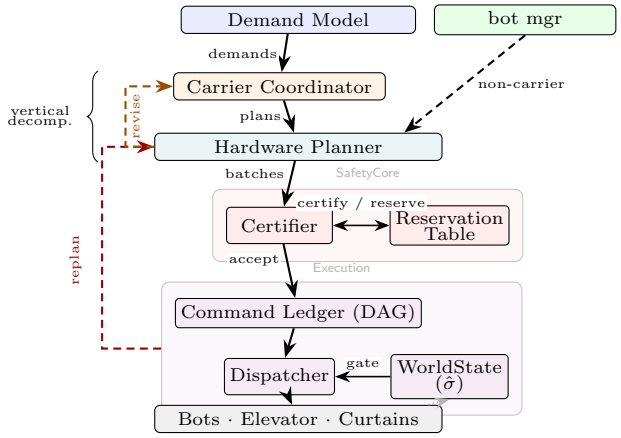

Figure 1: Layered architecture. The Demand Model maintains a demand-supply graph and emits constrained demand views; the Carrier Coordinator jointly selects carriers and plans rearrangements (stable, time-agnostic); the Hardware Planner expands plans into bot/elevator commands (volatile, time-aware). Non-carrier directives (charging, onboarding) flow via bot mgr and bypass the Carrier Coordinator. The Certifier checks proposed command batches against the space–time Reservation Table and atomically commits on acceptance. Certified commands enter the Command Ledger (DAG); the Dispatcher checks dependencies, preconditions, and a physical occupancy gate against WorldState ($\hat{\sigma}$) before issuing each command. Dashed red: localized replanning repairs the affected DAG subgraph; dashed orange: the Hardware Planner signals the Carrier Coordinator when carrier-level revision is needed.

Sections 4–5 detail the Carrier Coordinator and Hardware Planner internals respectively. Section 6 treats energy safety. Figure 1 summarizes the resulting layered architecture.

A key benefit of the decomposition is that it admits an *assume-guarantee* correctness argument [12]: each layer provides guarantees conditional on explicit assumptions about adjacent layers, and those assumptions are discharged by other layers' guarantees or by physical facts. Core safety (collision-freedom) is established by the execution engine alone, with no dependence on planning-layer correctness. Liveness (starvation-freedom) is conditional on well-formed graph topology and bounded hardware execution times—physical properties, not software guarantees. This structure makes each component's obligations explicit and locates open problems precisely.

A deployment-specific *Site Model* provides the static graph topology, module role designations (storage, autobahn,

balcony, elevator ingress/egress), and declarative traversal rules. The well-formedness predicates that underpin bot-level liveness (Section 5.7) are the Site Model's responsibility; a deployment passes validation only if its topology satisfies the required connectivity properties (Section 9).

2 Problem Formalization

Environment. A discrete set of modules $M = \bigcup_{\ell \in L} M_\ell$ across levels L, connected by directed adjacency $E \subseteq M \times M$. A shared elevator occupies $|M_{\text{elev}}| \leq 3$ modules that move rigidly between levels. The elevator is optional; a deployment with $|M_{\text{elev}}|{=}0$ is *level-isolated*: bots and carriers are confined to their assigned levels for the duration of that deployment (not as a fault response, but by design). For a level-isolated site the Site Model must satisfy a *per-level self-sufficiency predicate*: each level $\ell \in L$ contains at least one bot, at least one charging station ($M_{\text{ch}} \cap M_\ell \neq \emptyset$, further constrained below), and at least one balcony module. These are configuration-time obligations, validated alongside well-formedness (Section 9). For level-isolated deployments, cross-level arguments (assumption A3, the cross-level deadlock extension, and elevator dispatch gates) are vacuously satisfied. Light curtains K guard protected boundaries (e.g., access to modules beyond balconies). Charging modules $M_{\text{ch}} \subseteq M$ are dedicated docking locations distributed across levels (at least one per level); each serves one bot at a time. Each bot has a battery level $\beta(b) \in [0, \beta_{\max}]$; a per-deployment critical threshold $\beta_{\text{crit}} \in (0, \beta_{\max})$ marks the lowest battery at which the bot is still guaranteed to reach a charging station before exhaustion.

State. A system state is a tuple

$$\sigma = (\pi_B,\ \theta,\ \delta,\ \pi_C,\ \lambda,\ \chi,\ \iota,\ e_\ell,\ e_o,\ e_s,\ e_k,\ \kappa,\ \mathcal{T},\ \beta)$$

where $\pi_B : B \to M$ maps bots to modules,[1] $\theta : B \to \Theta$ is heading (required because charging preconditions depend on bot–station orientation alignment), $\delta : B \to \{\mathsf{f}, \mathsf{t}\}$ is driveable status ($\delta(b){=}\mathsf{f}$ after a fault; absorbing until external reset), $\pi_C : C \to M \cup B$ maps carriers to modules or carrying bots, $\lambda : B \to \{\downarrow, \uparrow\}$ is lift state, $\chi : B \to \{\mathsf{f}, \mathsf{t}\}$ is charging state (docked at a station and actively charging), $\iota : B \to \{\mathsf{f}, \mathsf{t}\}$ is initialization status ($\iota(b){=}\mathsf{t}$ once the two-phase setup handshake completes), $e_\ell \in L$, $e_o \in \{\mathit{idle}, \mathit{moving}\}$,[2] $e_s \in \{\mathit{normal}, \mathit{safety}\}$, $e_k \in \{\mathsf{f}, \mathsf{t}\}$ describe the elevator ($e_k{=}\mathsf{f}$ until the first WebSocket state message is received after coordinator startup; for level-isolated deployments these four components are absent from σ), $\kappa : K \to \{\mathit{unmuted}, \mathit{muted}, \mathit{breached}\}$ describes curtain state, $\mathcal{T}$ is the set of active carrier demands (unsatisfied, uncancelled demands from the Demand Model), and $\beta : B \to [0, \beta_{\max}]$ maps each bot to its current battery level.

[1] π_B maps bots to grid modules (occupancy cells), not sub-module positions. When a bot is docked at a charging station it is physically displaced within its module toward the charger; $\pi_B(b)$ remains the docking module throughout. Sub-module position is a hardware execution detail not captured in the formal model.

[2] The world model uses *idle_at_level* for $e_o{=}\mathit{idle}$; we abbreviate since e_ℓ already records the level.

Actions and feedback asymmetry. The system issues two command classes to two distinct entity types. *Bot commands* (MOVE, ROTATE, LIFT, LOWER, PLUG, PLUG_INV) are serialized per-bot, non-cancellable once dispatched, and provide *completion-only feedback*—no intermediate position or progress is observable. PLUG and PLUG_INV dock and undock a bot at a charging module; they share the same certification, ledger, and dispatch semantics as all other bot commands.[3] *Elevator commands* (ELEV_MOVE) are serialized for the single shared elevator, non-cancellable once dispatched, and operate on a *continuously observed* resource: e_ℓ, e_o, and e_s are pushed in real time. Curtain state is likewise continuously observed. Both command classes coexist in the same plan structure (a DAG of committed commands; Section 5), but their feedback models, failure modes, and dispatch semantics differ: a bot fault affects one agent; an elevator fault ($e_s \leftarrow \mathit{safety}$) inhibits *all* elevator commands system-wide, and an in-flight ELEV_MOVE locks the sole shared cross-level resource for its duration. Command durations are uncertain; actual execution time is revealed only at completion.

Observation model. The coordinator maintains a state estimate $\hat{\sigma}$ (the *world state*) updated by hardware feedback. Bot positions in $\hat{\sigma}$ update only on command completion; elevator and curtain components update continuously. Between dispatch of a bot command and its completion, $\hat{\sigma}$ retains the bot's pre-dispatch position—the coordinator cannot observe intermediate progress. Crucially, a new command for bot b is dispatched only after b's previous command completes, so at every dispatch decision point $\hat{\sigma}$ agrees with σ on $\pi_B(b)$. The same holds for elevator and curtain state by continuous observation. Dispatch-time safety checks (Section 5.6) therefore operate on accurate state. This argument assumes *faithful localization*: the bot's physical position does not diverge from its last-known module between command dispatch and completion reporting. A silent localization failure—where the bot moves inconsistently with its commanded trajectory without reporting a hardware fault—would violate the agreement property and allow the occupancy gate to dispatch into a physically occupied module. Bounding the localization error envelope is noted as an open problem (Section 9).

Exogenous events. Between any two command completions the environment may inject: demand arrivals or cancellations, bot faults ($\delta(b) \leftarrow \mathsf{f}$; absorbing until external reset), elevator faults ($e_s \leftarrow \mathit{safety}$; absorbing until reset), or curtain breaches.

Core safety invariant $S(\sigma)$: (i) no two bots share a module; (ii) no two carriers share a module; (iii) elevator entry/exit only when $e_o{=}\mathit{idle}$, $e_s{=}\mathit{normal}$, and e_ℓ matches

[3] The hardware implements two docking variants—backward charging and forward charging—distinguished by whether the bot's heading at the docking module aligns with the charging station's pin orientation. The formal model collapses these to a single PLUG/PLUG_INV pair; the variant is a deployment-time constraint enforced by the Site Model's station-orientation configuration and does not affect certification, reservation, or dispatch semantics.

the accessing bot's level; (iv) curtain timing constraints ($\tau_{\text{mute\_max}}$, τ_{breach}) are not violated; (v) *elevator movement gate*: ELEV_MOVE may execute only when no bot has an in-flight MOVE whose source or target is in M_{elev} (no bot is mid-transition to or from the elevator surface).

$S(\sigma)$ is enforced unconditionally by the certifier and dispatcher; no planning-layer assumption is required.

Energy safety invariant $E(\sigma)$: $\beta(b) > 0$ for all $b \in B$ (no bot reaches zero battery). $E(\sigma)$ is qualitatively different from $S(\sigma)$: it depends on a site-model assumption (per-level charging sufficiency) and a policy assumption (critical-threshold calibration), not solely on the execution engine. Section 6 treats $E(\sigma)$ separately.

Liveness objective. Every demand $d \in \mathcal{T}$ that is not cancelled is eventually served (*starvation-freedom*).

The challenge: maintain $S(\sigma)$ and $E(\sigma)$ while achieving liveness under online arrivals, exogenous faults, non-cancellable commands, and completion-only bot feedback.

3 Vertical Decomposition: Carriers and Bots

This section presents the decomposition, establishes carrier-level deadlock freedom (§3.3) and conditional liveness (§3.4) as *proved properties* of the architecture—treating the Carrier Coordinator and Hardware Planner as black boxes with specified external behaviour—and then composes both layers into end-to-end assume-guarantee correctness (§3.5). Sections 4 and 5 then open each subsystem to detail internal mechanisms and bot-level safety and liveness properties.

In dense warehouse systems, robots (*bots*) retrieve mobile shelving units (*carriers*) from storage lanes and transport them to pick stations, potentially across levels via a shared elevator. The joint problem of deciding *which carriers to retrieve*, *where they move*, and *which bots execute those moves* is combinatorially explosive.

Li and Ma [2] formalize a related problem as Double-Deck MAPD and propose MAPF-DECOMP, which decomposes into: (1) a **carrier-level plan** determining the sequence of rearrangements (displacing blockers, freeing corridor space), and (2) a **bot-level plan** assigning bots and computing collision-free paths to realize each carrier move. We extend this decomposition with an upstream *Demand Model* that maintains a demand-supply bipartite graph—active demands (unsatisfied SKU line items or carrier-pin requests) matched to carrier inventories—and exposes scored candidate sets to a *Carrier Coordinator*. The Carrier Coordinator jointly selects carriers from these candidates using physical layout knowledge (blocking depth, temporary placement availability) and plans the resulting rearrangements, unifying carrier selection and rearrangement planning in a single component.

This exploits a key asymmetry: demand-level state and carrier-level plans are *structurally stable*, changing mainly when demands change (seconds to minutes) or when execution diverges (e.g., carrier position after a bot failure); bot-level plans are *volatile*—a single fault invalidates paths, requiring immediate repair. Crucially, carrier-level plans are *time-agnostic*: they specify which carriers move where and in what causal order, but carry no temporal commitments. Space–time reservations are created only when the bot planner expands carrier steps into concrete commands, so bot-level timing disruptions never invalidate the carrier plan—only the bot-level schedule needs repair.

Completeness trade-off. Hierarchical decomposition may sacrifice theoretical completeness: a carrier plan feasible in isolation may be unrealizable at the bot level (e.g., no collision-free path exists for the assigned bot). The architecture trades completeness for tractability; a repair loop (Section 5) recovers in practice: when bot-level expansion or certification fails, the Carrier Coordinator revises.

3.1 Dense storage as a distinct subproblem

Carrier retrieval from narrow, serial-access storage zones is equivalent to a puzzle-based storage (PBS) retrieval problem [6, 7], NP-complete in the general case. Treating it as a separate layer enables domain-specific heuristics (empty-cell invariants, displacement budgets) that would be lost in a monolithic formulation.

3.2 Multi-task coordination

When concurrent carrier demands share limited resources (balconies, levels, elevator), the Carrier Coordinator must coordinate their plans to avoid deadlock. Two coordination modes are natural: *strict-lexicographic*, where plans are computed sequentially by priority with no downward information flow (each plan reads only its own demand, its own candidates, and higher-priority claims), and ε-*cooperative*, where a higher-priority plan may read lower-priority demand data and accept bounded suboptimality to reduce joint cost. The canonical ε-cooperative instance is the *blocker-is-target* optimization (Section 4.3): P_i routes a displaced blocker to a lower-priority demand D_j's goal instead of a temporary location, merging the displacement step with D_j's presentation at the cost of a longer blocker route for P_i. P_j, planned later, sees the carrier at its goal in the projected post-P_i layout and selects it with zero extraction cost. This introduces a bounded upward information-flow dependency: P_i's output may depend on D_j's candidate set ($j > i$), so a change to D_j can invalidate P_i—the strictly-downward memoization cascade of Section 4.4 does not hold. Execution-level acyclicity is preserved: the dependency direction is lower-to-higher (P_j depends on P_i completing the delivery), the same direction as standard priority blocking. Both modes produce the same plan data structures and obey the same structural invariants (empty-cell guarantee, acyclic waits-for relation); the choice is a deployment-time policy. This paper assumes strict-lexicographic coordination except where ε-cooperative is noted explicitly. *Level-health plans* (returning undemanded carriers to storage) participate in the same priority ordering and are escalated when the Carrier Coordinator detects resource saturation (Section 3.4).

3.3 Deadlock freedom

The argument has two stages: per-level, then cross-level.

Invariant 1 (empty-cell guarantee): At every intermediate state during carrier-plan execution, the level retains

at least $k \geq 1$ free modules. Shirazi and Zolghadr [8] prove that with one free cell, a feasible retrieval sequence exists for any target—no reachable state is a dead end. The Carrier Coordinator enforces this constructively: before emitting each carrier movement step, it verifies that the step's source module becoming free and the destination module becoming occupied does not reduce the free-module count below k. This check is performed against the *projected* post-step layout (accounting for all previously emitted steps in the current planning cycle), not merely the physical state—so overlapping plans cannot collectively violate the invariant. The value of k is a per-level deployment parameter derived from the Site Model (Section 9).

Invariant 2 (priority-ordered planning): Plans are computed in strict priority order; plan P_i treats higher-priority placements as immutable occupied cells. Every dependency edge points from lower to higher priority, so the waits-for relation is acyclic by construction [5].

Deferral, not deadlock: If plan P_j cannot find a feasible rearrangement in the remaining space, it is deferred until higher-priority work completes. Throughput degrades under saturation but no plan enters an unresolvable wait.

Required conditions: (1) the empty-cell invariant holds at plan start and is preserved by every step; (2) the priority ordering is total and stable within a planning cycle; (3) no agent outside the Carrier Coordinator claims temporary placement space. When any condition is violated (e.g., a bot fault drops a carrier at an unexpected module), the Hardware Planner detects the inconsistency between expected and actual carrier positions via bot-level feedback and signals the Carrier Coordinator, which re-evaluates from the current physical state (Section 4.1).

Cross-level extension: The elevator is a time-shared, renewable resource: a carrier plan uses it for a bounded-duration trip, then releases it. Critically, a bot exits the elevator onto the destination level before releasing the elevator for other trips—so the elevator is released regardless of destination-level resource availability (the bot does not hold the elevator while waiting for a balcony). The global priority ordering extends the acyclicity argument across levels: plan P_i operates on every level it touches before any lower-priority plan P_j. In level-isolated systems this extension is vacuous.

Caveat: If the destination level is saturated and the Carrier Coordinator must re-store a carrier to a *different* level to free a balcony, the re-storage plan itself requires the elevator, creating a cross-level dependency chain. The architecture relies on the per-level empty-cell invariant to ensure that same-level re-storage is always feasible—eliminating the need for cross-level re-storage in the critical path. The argument follows from the PBS feasibility result [8]: with $k \geq 1$ free cells in a connected storage zone, any carrier can be retrieved or placed. Re-storage is the reverse of retrieval—placing a carrier into a free cell—and is feasible under the same conditions. The remaining obligation is a Site Model configuration-time check: the storage zone on each level must be connected (accounting for one-way edges and zone restrictions) so that the PBS result applies. This reduces the cross-level deadlock concern to a verifiable graph property rather than an open theoretical problem.

3.4 Conditional liveness at the carrier level

Deadlock freedom is a safety property (no irrecoverable state); liveness requires a separate argument that progress occurs [10]. We establish starvation-freedom under five explicit assumptions:

- **A1** *Bounded active demands:* $|\mathcal{T}| \leq N$ at all times.
- **A2** *Starvation-free priority:* no demand remains lowest-priority indefinitely (e.g., priority aging).
- **A3** *Bounded elevator time:* each trip completes within T_{elev}; vacuous for level-isolated systems.
- **A4** *Bounded bot-level step realization:* each carrier movement step is realized by the bot planner within bounded time T_{step}.
- **A5** *Bounded presentation hold:* each carrier presentation at a balcony completes (all picks served, carrier released for storage) within bounded time T_{pick}; vacuous when no human pickers are present (liveness of non-presentation demands is unaffected).

Resource-freeing as an internal guarantee. A previous version of this argument required an external assumption ("resource-freeing scheduling") that an upstream layer would schedule carrier storage with sufficient priority to prevent scarce-resource saturation. The Carrier Coordinator internalizes this obligation: it has joint visibility into the demand view (which balconies are needed) and the physical state (which balconies are occupied by undemanded carriers). When a presentation plan requires a free balcony and none is available, the coordinator identifies the lowest-cost undemanded carrier at a balcony and generates a re-storage plan with priority at least as high as the blocked presentation. By the deadlock freedom argument above and bounded execution (A4), the re-storage plan completes within bounded time, freeing the resource. This eliminates the need for an external scheduling guarantee: resource-freeing is a structural consequence of the coordinator's joint optimization scope.

Saturation detection completeness. The internal guarantee requires that the coordinator *detects* saturation—it must observe that scarce resources are occupied by undemanded carriers. Every path to a carrier becoming undemanded at a balcony involves a state transition that fires a replanning trigger: pick completion or demand cancellation changes the demand view; a carrier placed temporarily by the coordinator arrives via a step completion signal. In each case, the coordinator's next planning cycle reads carrier positions from the world state (no internal placement memory that could become stale), observes the undemanded carrier at the scarce resource, and can escalate re-storage. Startup performs a full sweep with no assumption about prior state, eliminating the initial-state problem. The detection bound is therefore one coalesce window plus one planning-cycle latency after

the triggering event (Section 4.2 details the trigger classification).

Under A1–A2, every demand eventually becomes highest-priority. By the empty-cell invariant and PBS feasibility [8], a feasible carrier plan exists. Under A3, any required elevator trip completes within T_{elev}. Under A4, each carrier step is realized in bounded time. By the internal resource-freeing guarantee, scarce resources (balconies, autobahn capacity) do not remain permanently saturated. Under A5, each carrier presentation is bounded, ensuring that balcony occupancy is transient. Every demand in $\mathcal{T}$ is therefore eventually served. A1–A3 and A5 are external (physical or policy); **A4 is the critical cross-layer assumption**, discharged in Section 3.5 via bot-level properties.

Ma et al. [11] prove an analogous result—bounded task-completion on well-formed graphs—for lifelong MAPF; the structural pattern (infrastructure connectivity plus a progress-ensuring policy yields liveness) is shared.

Decomposition *compounds* with the two replanning strategies described in Sections 4–5: carrier-level full re-evaluation and bot-level localized repair operate at different frequencies and scopes, and a disruption at one layer does not necessarily propagate to the other.

3.5 Compositional guarantees

We compose the preceding decomposition with bot-level properties (established in Section 5) into end-to-end guarantees, structured as an *assume-guarantee composition* [12]: each layer provides guarantees conditional on assumptions about other layers; we show the assumptions are discharged by guarantees of other layers or by physical facts.

Claim (informal). Under assumptions A1–A5 (Section 3.4), the internal resource-freeing guarantee, well-formed infrastructure [11], and bounded command execution times: (a) the core safety invariant $S(\sigma)$ holds unconditionally at all reachable states; (b) the energy safety invariant $E(\sigma)$ holds conditional on per-level charging sufficiency (Section 6); and (c) every demand that is not cancelled is eventually served.

Exogenous events and conservative postures. Each exogenous event class preserves $S(\sigma)$: a bot fault does not release the faulted bot's reservation (it remains occupied at its last known module); an elevator fault sets $e_s = \textit{safety}$, inhibiting further elevator commands; a curtain breach propagates to $e_s = \textit{safety}$. In every case the system enters a conservative posture that maintains $S(\sigma)$; recovery is gated by external reset. These postures correspond to three distinct degradation levels. An elevator fault or curtain breach induces *level-isolated operation*: $S(\sigma)$ maintained; A3 violated; cross-level liveness lapses; same-level operations continue; $E(\sigma)$ sustained via per-level charging sufficiency. One or more bot faults induce *reduced-fleet operation*: $S(\sigma)$ and $E(\sigma)$ maintained; A4 weakened; throughput degrades with fleet loss. Simultaneous faults, livelock detection, or world-state reconciliation failure trigger a *safety hold*: all dispatch suspended; in-flight commands complete normally; reservations held conservatively; recovery gated on verified world state and operator confirmation.

Assume-guarantee structure. The dependency between layers is *not circular*:

- *Bot-level safety* (P1–P4 + dispatch gate) depends only on structural properties of the execution engine—not on the Carrier Coordinator.
- *Bot-level liveness* (T_{step} bounded) depends on well-formed infrastructure, physical T_{cmd}, and a bounded repair budget—not on Carrier Coordinator correctness.
- *Carrier-level liveness* depends on bot-level liveness (A4), external assumptions A1–A3 and A5, and the Carrier Coordinator's internal resource-freeing guarantee. An elevator fault violates A3 (bounded elevator time); liveness degrades for all cross-level demands while same-level demands continue to be served.

The assumptions ground out in physical facts (bounded hardware execution time, graph topology), external policy (stable load, fair priority, bounded pick duration), and the Carrier Coordinator's internal scheduling guarantee, not in other software guarantees. This is the assume-guarantee pattern of Abadi and Lamport [12]: component specifications are conjoined, and circular reasoning is avoided because the lowest-layer assumptions are physical invariants.

What remains for a formal proof. The argument above is structured but not mechanized. A formal proof requires: (i) a precise well-formedness predicate for the warehouse graph, extending [11] to multi-level topologies with elevator constraints; and (ii) a refinement mapping showing the decomposed system implements the monolithic specification.

4 Carrier Coordinator Internals

The properties established in Sections 3.3–3.4 depend on the Carrier Coordinator's internal mechanisms but do not describe them algorithmically. This section opens that black box: the planning cycle structure (§4.1), trigger classification (§4.2), carrier selection and rearrangement algorithm (§4.3), and input fingerprinting for memoization (§4.4). The Hardware Planner's internals—localized DAG repair, space–time reservations—follow in Section 5.

4.1 Planning cycle structure

The Carrier Coordinator does not maintain a persistent plan DAG that it surgically repairs (as the Hardware Planner does with committed commands). Instead, each planning cycle is a *full priority-ordered re-evaluation* of all active demands:

1. Read current carrier positions from the world state. The coordinator has *no internal placement memory*—it discovers the carrier layout fresh each cycle.
2. Read the current demand view from the Demand Model.
3. Compute carrier plans in strict priority order (Section 3.2), treating the current physical state as ground truth.

Full re-evaluation is both correct and efficient for this

problem size. The coordinator manages at most N plans (bounded by A1) that are highly interdependent through shared resources (balconies, temporary placement space, the empty-cell invariant). A change to any plan can cascade through lower-priority plans via shared placement constraints, so localized repair would require global reasoning anyway.

Planning cycles are *serialized*: at most one cycle runs at a time; triggers arriving mid-cycle are queued and processed in the next cycle. This eliminates race conditions between concurrent cycles and makes the acyclicity argument (Section 3.3) straightforward—each cycle produces a total priority ordering atomically. Serialization is not a bottleneck: each cycle evaluates $O(N)$ demands with small candidate sets, and the computational cost is dominated by the rearrangement solver (§4.3), which operates per-level (carrier rearrangement is a within-level problem) without time reasoning.

4.2 Trigger tiering and coalescing

Triggers are classified into two tiers based on whether they represent an *unexpected* change to the coordinator's planning inputs or an *expected* confirmation that the plan is executing as intended. The distinction is architectural, not a policy choice: Tier 1 triggers invalidate inputs; Tier 2 triggers confirm them.

Tier 1—immediate (unexpected input changes): demand view changed (new, cancelled, or satisfied demand; priority change); carrier at unexpected position (fault, manual intervention); step expansion failed at the bot level; pick confirmed (remaining quantity updated); system startup or recovery. A Tier 1 trigger starts a planning cycle promptly; if a coalesce timer is running (from an earlier Tier 2 trigger), it is cancelled and the cycle starts immediately.

Tier 2—coalesced (expected state changes): step completed—the carrier moved to its expected position. Current plans remain valid; re-evaluation is an optimization opportunity (e.g., a newly freed temporary cell may unblock a deferred plan), not a correctness requirement. Multiple step completions within a bounded *coalesce window* W are batched into a single cycle. If no deferred plans exist, the batch updates the plan registry and skips re-evaluation entirely.

Coalesce mechanism. On each incoming trigger: if a cycle is already running, the trigger is queued (serialization). Otherwise, a Tier 1 trigger schedules an immediate cycle; a Tier 2 trigger starts (or joins) a coalesce timer of duration W. When the cycle runs, it drains all queued triggers, updates the plan registry for any step completions, and invokes full re-evaluation if any Tier 1 trigger is present or deferred plans exist.

Detection completeness. Every path to resource saturation involves a trigger from this table. Picks completing or demands being cancelled change the demand view (Tier 1, immediate); a carrier placed temporarily by the coordinator arrives via step completion (Tier 2, coalesced within W). The coordinator reads carrier positions from the world state at cycle start, so a persistent saturation condition is detected regardless of batching; transient saturation that self-resolves within W was not actionable. The detection bound is therefore one coalesce window plus one planning-cycle latency after the triggering event.

4.3 Carrier selection and rearrangement

Carrier retrieval from a serial-access storage lane at depth d requires displacing the $d-1$ carriers between the target and the lane's I/O point. The algorithm adapts the *escort pattern* of Shirazi and Zolghadr [8]: the "escort" is the empty cell that enables carrier movement. The *autobahn*—a Site Model role designation for a dedicated module column—is the physical escort column connecting the *storage lanes* to wings and balconies; the storage lanes (a separate Site Model role designation) flank the autobahn and are the designated home positions for undemanded carriers. The Carrier Coordinator queries both role designations to determine valid temporary-placement targets and carrier home positions. Each extraction has three phases: (1) displace blockers from positions $1..d-1$ to temporary locations on the autobahn or in other lanes; (2) extract the target carrier to the goal (e.g., a balcony); and (3) re-store displaced blockers in reverse order.

Blocker-is-target optimization (ε-cooperative only; Section 3.2). When a displaced blocker is itself a candidate for a lower-priority active demand, the coordinator routes it directly to that demand's goal instead of a temporary location—merging the displacement step with the presentation step. This requires ε-cooperative mode: the higher-priority plan reads a lower-priority demand's candidate set to make the routing decision, introducing an upward information-flow dependency incompatible with strict-lexicographic planning (Section 3.2). Under strict-lexicographic mode, blockers are always routed to temporary locations; the optimization is unavailable.

Selection cost function. For N active demands on a level, each with at most C candidates (filtered by the Demand Model), the coordinator evaluates combinations using a composite cost:

$$\text{cost} = \textstyle\sum_i \big(w_d \cdot (1 - s_i) + w_p \cdot r_i\big)$$

where s_i is the Demand Model's score for candidate c_i (higher is better) and r_i is the number of carrier movement steps required to extract c_i. The selection space is $O(C^N)$; with N bounded by A1 and $C \leq 5$ in practice, exhaustive evaluation is tractable because the rearrangement solver's per-invocation cost is low (no time reasoning, no collision checking). Standard pruning (blocking-depth dominance, cost-bound cutoff, infeasibility cutoff) reduces the constant. A greedy fallback—selecting per-demand in priority order—is available when C^N exceeds a configurable threshold.

Feasibility guarantee. Shirazi and Zolghadr [8] prove that a single free cell suffices for any retrieval in a connected graph. The storage-lane/autobahn topology satisfies connectivity (every storage module connects to the autobahn via its lane's I/O point), and the empty-cell invariant (Section 3.3) guarantees $k \geq 1$ free cells. Extraction is

therefore always feasible; the coordinator need not handle "stuck" states.

4.4 Input fingerprinting

Full re-evaluation (§4.1) may recompute plans whose inputs have not changed. A memoization key per plan short-circuits this redundant work:

$$\text{key}(P_i) = \mathcal{H}\big(d_i.id,\ d_i.prio,\ d_i.cands,\ \pi_C[refs_i],\ claims_{1..i-1}\big)$$

where d_i is the demand, $\pi_C[refs_i]$ captures the positions of all carriers referenced by P_i's candidate set and rearrangement plan, and $claims_{1..i-1}$ is the set of cells claimed by higher-priority plans.

If the key matches the stored key from the previous cycle, the stored plan (including its claimed cells) is reused without invoking the rearrangement solver. Under strict-lexicographic coordination, invalidation cascades *strictly downward* in priority: a change to P_i can invalidate $P_{i+1}, \ldots, P_N$ (their *claims* input may differ), but cannot invalidate any P_j with $j < i$—higher-priority plans are computed first and their inputs do not include lower-priority outputs. Under ε-cooperative mode (Section 3.2), the blocker-is-target optimization introduces a bounded upward dependency: P_i's routing may depend on lower-priority demands' candidate sets, so a change to D_j $(j > i)$ can invalidate P_i. The key for P_i must then include the lower-priority candidate overlap that influences blocker routing; the strictly-downward cascade property does not hold. In the worst case a highest-priority change recomputes all N plans; in the common case (no demand or layout change) all plans hit the cache at $O(N)$ hash cost and zero solver invocations.

5 Hardware Planner

The Hardware Planner translates carrier-level plans into certified bot and elevator commands, manages them in a DAG with space–time reservations, and enforces bot-level safety and liveness through the execution pipeline. Where the Carrier Coordinator performs full re-evaluation over a small plan set (Section 4), the Hardware Planner manages a large DAG of committed commands. Most exogenous events affect only a local region of this DAG—a bot fault invalidates that bot's path and its dependents, not agents in distant warehouse regions—so *localized repair* is the efficient response. Sections 5.6–5.7 then establish the safety and liveness properties of this pipeline.

5.1 Rolling-horizon approaches

Li et al. [3] introduce Rolling-Horizon Collision Resolution (RHCR) for lifelong MAPF: a windowed solver resolves conflicts within the next w timesteps, called every h steps $(w \geq h)$. This bounds per-invocation complexity and scales to 1,000+ agents. However, a temporal window does not distinguish in-flight (non-cancellable) commands from pending ones—a critical distinction when hardware commands cannot be retracted. Moreover, standard windowed approaches offer no completeness guarantee and may deadlock or livelock; Veerapaneni et al. [13] address this with WinC-MAPF, introducing completeness guarantees for windowed MAPF via heuristic updates from real-time search. Our architecture bounds replan scope via *causal dependency* rather than temporal windowing, as described next.

5.2 Plan-graph–based partial repair

A finer-grained approach represents the active plan as a directed acyclic graph (DAG) of committed commands with two edge types. *Dependencies* are causal sequencing edges: cancelling a command transitively cancels all dependents (closure). *Preconditions* are physical-state requirements (e.g., "elevator at level X") checked at dispatch time; they carry no cancellation semantics, keeping independently-certified plans independently redactable.

Three advantages over windowed replanning: (a) replan scope follows *causal dependency*, not temporal proximity; (b) in-flight commands are never disturbed, respecting non-cancellable semantics; and (c) independently-certified plans remain independently redactable (cross-plan coordination uses preconditions, not dependencies). The runtime dispatcher checks before issuing each command: dependency satisfaction, precondition satisfaction against the current state estimate $\hat{\sigma}$, and a physical occupancy gate (Section 5.6). Failed dependencies trigger cascading cancellation; failed preconditions route to the planner for repair—no cascade.

Heterogeneous commands in the DAG. The DAG contains both bot and elevator commands. The two edge types—dependencies and preconditions—arise precisely from this heterogeneity. Within a single plan segment (commands planned and certified together), *dependencies* express causal ordering: a bot's entry MOVE onto an elevator module must complete before the ELEV_MOVE departs, and the ELEV_MOVE must complete before the exit MOVE at the destination level. These commands share planning assumptions and should be redacted as a unit.

Preconditions enable independent plans to coordinate on *shared resources without lifecycle coupling*. When two independently-planned bots share an elevator trip (Section 8), the second bot's post-elevator command carries a precondition ("elevator at level X") rather than a dependency on the first bot's ELEV_MOVE. If the first bot's plan is cancelled, the second bot's precondition fails at dispatch time and routes to the planner for repair—a new elevator trip is planned—rather than cascading cancellation across plan boundaries. This separation is what makes independently-certified plans independently redactable.

5.3 Space–time reservation tables

A shared reservation table maps each (module, time-interval) pair to an owning command. Reservations are held until the command reaches a terminal state (completed or failed), *not* until its predicted completion time—preventing timing overruns from silently violating occupancy invariants.

Elevator modules (M_{elev}) are reserved by the *bot commands* that place bots on them—a bot's MOVE onto elevator module E_1 reserves $(E_1, [\cdot])$ in the table, just as any other module. The ELEV_MOVE itself is a *state-change command*: it transitions e_ℓ and e_o but does not occupy

additional modules. Its correctness is enforced by elevator serialization (at most one ELEV_MOVE at a time) and by the elevator movement gate ($S(\sigma)$ item (v), Section 5.6), both checked by the dispatcher at dispatch time—not by the reservation table.

5.4 Failure blast radius

The replan scope of a failure depends on which entity fails. A *bot fault* is local: the faulted bot's pending commands and their DAG dependents are cancelled; other bots' plans are unaffected. An *elevator fault* ($e_s \leftarrow$ *safety*) is effectively global for cross-level work: every pending ELEV_MOVE and its transitive dependents must be cancelled, potentially spanning multiple carrier plans. However, same-level plans that require no elevator transit continue undisturbed. In practice, most exogenous events are bot-level; elevator faults are rare but high-impact, and the architecture degrades to level-isolated operation until recovery.

5.5 Bot allocation and non-carrier task integration

The Hardware Planner's bot assignment is an instance of single-task, single-robot, time-extended assignment (ST-SR-TA in the Gerkey–Matarić taxonomy [14]): each carrier movement step or non-carrier task requires exactly one bot, and the chosen bot is occupied for the step's duration.

Priority-aware assignment. The Carrier Coordinator emits plans with a priority inherited from the demand ordering (Section 3.2). The Hardware Planner expands plans in strict priority order—highest first—so that the best-positioned available bot is assigned to the most urgent work. Within a single plan, independent steps (no *depends_on* relation) may be assigned to different bots in parallel; dependent steps are assigned as preceding steps complete.

Non-carrier task hierarchy. Carrier work and non-carrier tasks share a single bot pool, so the Hardware Planner enforces a total priority ordering: (1) critical charging ($\beta(b) \leq \beta_{\text{crit}}$ or FORCECHARGE)—non-preemptible once issued; (2) carrier work, ordered by Carrier Coordinator priority (level-health re-storage plans participate in this ordering and may be escalated to high priority under resource saturation, Section 3.4); (3) opportunistic charging—interruptible; the bot unplugs when carrier work arrives; (4) onboarding staging—new bot completing the setup handshake. This total ordering ensures that the most urgent work always claims the next available bot.

Fleet partitioning and effective fleet. At any instant, the bots on level ℓ partition into: bots executing carrier commands (B_c), bots in critical charging (B_{cc}; unavailable), bots opportunistically charging (B_{oc}; available after short unplug delay), bots in lifecycle transitions (B_{lc}; unavailable during handshake), and idle bots (B_{id}; immediately available). The *effective carrier fleet* is $B_{\text{eff}} = B_c \cup B_{\text{oc}} \cup B_{\text{id}}$; A4 (bounded step realization) requires $|B_{\text{eff}}| \geq 1$.

Minimum fleet predicate. For sustained liveness, each level ℓ requires $n_{\min}(\ell) \geq 2$ bots under normal operation: one may be charging while the other serves demands. With $r_{\text{charge}}/r_{\text{discharge}} \approx 3$–$5$, each bot spends roughly 20–25% of its duty cycle charging, so two bots sustain continuous carrier availability. Zou et al. [15] show that ignoring the charging duty cycle underestimates the required fleet by more than 15%; the minimum fleet must account for charging load explicitly. At higher demand rates requiring k simultaneous working bots, $n_{\min}(\ell) = k + \lceil k \cdot r_{\text{discharge}}/r_{\text{charge}} \rceil$. This predicate is a configuration-time Site Model obligation, validated alongside the per-level self-sufficiency predicate (Section 2).

Single-bot levels. When $|B_\ell| = 1$, all carrier work serializes and critical charging takes the level entirely offline for the charge cycle. Effective throughput reduces by the charging fraction: $\Theta_{\text{eff}} = (1 - \alpha)\,\Theta_{\text{single}}$ where $\alpha \approx r_{\text{discharge}}/(r_{\text{discharge}} + r_{\text{charge}}) \approx 0.2$–$0.25$. Safety ($S(\sigma)$) and energy safety ($E(\sigma)$) are unaffected. Liveness degrades: A4 is transiently violated during charging (no bot available for carrier steps), but every demand is eventually served with worst-case added latency bounded by one charge cycle. For level-isolated deployments with single-bot levels, this latency bound should be surfaced as a deployment parameter.

Onboarding and offboarding. Onboarding introduces a new bot at a specified module. The Hardware Planner verifies the module is unoccupied via the reservation table, waits for the setup handshake ($\iota(b) \leftarrow \mathtt{t}$), and adds the bot to the availability pool—during which the bot is unavailable. Offboarding reverses this: the Hardware Planner completes or redacts pending commands, lowers any lifted carrier (the Carrier Coordinator detects the position change via CARRIERPOSITIONCHANGED), routes the bot to a designated offboarding module, and removes it from the pool. Both transitions update the effective fleet; the minimum fleet predicate must account for expected lifecycle churn to prevent transient A4 violations during planned transitions.

5.6 Safety properties

The execution pipeline's safety rests on four independently grounded properties of the command ledger and dispatcher.

P1: Closure-based cancellation. Cancelling a command transitively cancels all DAG dependents (dependency edges only; preconditions do not propagate). No descendant executes without its prerequisites.

P2: In-flight immutability. Commands already dispatched are non-cancellable; they run to completion and retain reservations until reaching a terminal state. For bot commands, this locks one agent for a bounded duration. For an ELEV_MOVE, this locks the sole cross-level resource for the trip duration, affecting every plan that requires the elevator.

P3: Single-authority certification. Every command batch must pass through a single certifier that atomically verifies collision-free space–time reservations before committing. No planning layer may bypass certification. Replacement commands produced by a replan pass the same

certifier.

P4: Serialized mutation. The command ledger processes all mutations (certification, reservation commits, feedback, redaction) through a single-threaded event loop, eliminating race conditions between concurrent replan cycles.

Composition. After a partial replan: (a) surviving commands retain intact reservations; (b) in-flight commands complete normally; (c) cancelled commands' reservations are freed atomically; (d) replacement commands are certified against the surviving table. At no point do conflicting space–time claims coexist.

Runtime safety net. Certification proves safety under *predicted* timing. The dispatcher enforces safety under *actual* conditions via a centralized set of dispatch-time checks, varying by command type. For *all bot commands*, two preconditions are checked at dispatch time regardless of command type: (a) *initialization gate*—$\iota(b)$=t, ensuring no command is issued to a bot that has not yet completed its setup handshake; and (b) *driveable gate*—$\delta(b)$=t. In addition, PLUG_INV carries the precondition $\chi(b)$=t (the bot must be currently docked before an undock command can be issued), analogous to the lift-state precondition on LOWER. For *bot* MOVE *commands*, a physical occupancy gate checks module occupancy against $\hat{\sigma}$ before dispatch, closing the gap between certification-time predictions and runtime reality. For ELEV_MOVE *commands*, the dispatcher enforces four checks: (a) *elevator state known*—e_k=t (no elevator command is issued before the WebSocket subscription delivers the first state message; this guards against evaluating elevator preconditions against a default or stale value); (b) *elevator serialization*—at most one ELEV_MOVE is in-flight at a time; (c) *elevator state preconditions*—e_s=*normal*, e_o=*idle*, curtain state clear; and (d) *elevator movement gate* ($S(\sigma)$ item (v))—no bot has an in-flight MOVE whose source or target is in M_{elev}. Check (d) has a useful coordination side-effect: if a bot from another independently-certified plan is still mid-entry onto the elevator, the ELEV_MOVE is held until that bot completes boarding. The elevator naturally waits for active boarders, bounded by T_{cmd}. Bots that have not yet started boarding miss the trip; their post-elevator preconditions fail at dispatch and route to the planner for a new elevator trip via the standard repair path. All dispatch-time checks are centralized in the dispatcher. This mirrors the layered execution model of Hönig et al. [9]: a planned layer provides predicted safety; a runtime layer provides actual safety; their composition guarantees collision avoidance under execution uncertainty. Together, certification prevents *planned* conflicts, the dispatch gates prevent *runtime* conflicts, and reservation-lifetime semantics prevent *silent* conflicts from timing overruns.

5.7 Conditional liveness: reducing A4 to a site-model property

Carrier-level liveness (Section 3.4) is conditional on A4 (bounded bot-level step realization) and A5 (bounded presentation hold); resource-freeing scheduling is guaranteed internally by the Carrier Coordinator (Section 3.4). We identify sufficient conditions under which A4 holds, reducing it to a *verifiable property of the deployed graph topology*—but we do not discharge it: the well-formedness predicate has not been stated for multi-level warehouse graphs, and formal verification against the deployed topology remains open.

Bot-level deadlock freedom. Ma et al. [11] prove bounded task-completion for lifelong MAPF on *well-formed* graphs (every agent can reach a non-task endpoint from any location). Okumura et al. [5] provide deadlock-freedom via PIBT on biconnected graphs. Under either condition, the bot planner finds a collision-free path for each carrier step. The well-formedness predicate must be verified against the deployed module topology (including one-way edges and zone designations); this is a Site Model configuration-time check (Section 9). **A4 therefore holds conditional on the Site Model passing well-formedness validation**—it is reduced to a graph-structural property, not a runtime guarantee.

Bounded execution. Each bot command completes within a physically bounded time T_{cmd} (a hardware property). On a well-formed graph with $|M|$ modules, a collision-free path of length at most $|M|$ exists for any start–goal pair (every simple path visits each module at most once), so each carrier step requires at most $O(|M|)$ bot commands. The step-realization bound is therefore $T_{\text{step}} \leq O(|M|) \cdot T_{\text{cmd}} + R \cdot T_{\text{cert}}$, where T_{cert} is certification latency and R is the maximum number of repair attempts per step.

Repair budget. The certify→reject→repair→re-certify loop at the bot level has no structural termination guarantee: if exogenous events arrive during repair, the input state may change before completion, restarting the cycle. We bound this by observing that for any *fixed* world state, a deterministic planner produces a deterministic output, so the loop terminates in one iteration absent further events. The relevant bound is therefore $R \leq 1 + \lceil f \cdot T_{\text{plan}} \rceil$, where f is the exogenous event rate and T_{plan} is the single-iteration planning time. Liveness requires $f \cdot T_{\text{plan}} < 1$ on average; when this condition is violated, the system cannot keep pace with disturbances and should enter a degradation posture.

6 Energy Safety: $E(\sigma)$

The energy safety invariant $E(\sigma)$—no bot shall ever reach zero battery—is qualitatively different from the core safety invariant $S(\sigma)$. $S(\sigma)$ items (i)–(v) are space–time properties enforced unconditionally by the Certifier and Dispatcher; $E(\sigma)$ is an *energy-domain* property whose guarantee rests on an infrastructure assumption rather than on the execution engine alone.

Site-model assumption (charging sufficiency). Every deployment's Site Model must satisfy: for each level $\ell \in L$, the charging modules $M_{\text{ch}} \cap M_\ell$ are sufficient that even if a strict majority of the fleet is stranded on ℓ indefinitely by an elevator fault ($e_s \leftarrow$ *safety*), every stranded bot can maintain $\beta(b) > 0$ by time-sharing the available stations. Time-sharing is feasible because the charge rate

$r_{\text{charge}} \gg r_{\text{discharge}}$: a bot recovers charge substantially faster than it depletes it, so a modest number of stations sustains a large stranded population through sequential access. This is a configuration-time predicate, validated at deployment alongside the well-formedness check for bot-level liveness (Section 5.7).

Battery Monitor and critical-priority path. An external Battery Monitor observes $\beta(b)$ continuously. When $\beta(b) \leq \beta_{\text{crit}}$, it generates a charging task at the highest system priority—above all carrier work. The Hardware Planner preempts pending carrier commands for the affected bot and routes it to the nearest available charging station. PLUG and PLUG_INV participate in the Command Ledger and the same certification path as any other bot command; they cannot be overtaken by lower-priority work once issued. The critical threshold margin is calibrated against worst-case travel distance on the level, so the charging route always completes before $\beta(b) = 0$.

Argument for $E(\sigma)$. Consider any reachable state. *Normal operation.* For any bot approaching β_{crit}, the Battery Monitor issues a highest-priority charging task before the threshold is crossed. By the critical threshold margin, the bot reaches a station and docks before exhaustion. Critical charging tasks are non-preemptible; the Hardware Planner cannot subordinate them to carrier demand.

Elevator stranding. An elevator fault may strand bots on a level for an unbounded duration. By the charging sufficiency assumption, the affected level has enough stations for time-sharing. Each bot's Battery Monitor independently detects its critical threshold; the Hardware Planner schedules station access in priority order. Because $r_{\text{charge}} \gg r_{\text{discharge}}$, each bot occupies a station for a bounded interval, then vacates for the next, sustaining $\beta(b) > 0$ across the stranded population.

Interaction with liveness. Bots engaged in critical charging are temporarily unavailable for carrier-step realization, which may transiently violate A4. Carrier-level liveness degrades for the stranding duration but resumes once bots return from charging; the liveness degradation is bounded by the charge cycle (charge/discharge ratio and station count), not permanent starvation. $E(\sigma)$ holds throughout.

Placement in the assume-guarantee structure. $E(\sigma)$ is grounded in: (a) the Battery Monitor's critical-priority preemption (software), (b) the critical threshold margin calibrated against hardware travel bounds, and (c) the per-level charging sufficiency assumption (Site Model). Item (c) is a configuration-time obligation of the Site Model, not a runtime guarantee—the same structural role as well-formed infrastructure for bot-level liveness (Section 5.7). The lowest-layer assumptions (charge/discharge ratio, travel distance bounds, station layout) are physical facts, keeping the assume-guarantee chain free of circularity.

7 Worked Example: Demand Preemption

We trace a concrete scenario through the full layer stack.

Setup. Demand D_2 (MEDIUM) is mid-execution: carrier C_5 has been extracted, with blocker C_9 temporarily at autobahn module A_6. A bot carries C_5 toward the balcony (in-flight). New demand D_1 (HIGH) arrives; the Carrier Coordinator selects carrier C_1, whose extraction requires A_6. Figure 2 traces the resulting cascade.

	Layer	Action
1	Demand M.	D_1 (HIGH) arrives; demand view updated.
2	Carrier C.	Selects C_1 for D_1; plans D_1 first (highest priority); claims A_6.
3	Carrier C.	D_2's plan conflicts (C_9 at A_6). Replans D_2: reroute $C_9 \to A_8$.
4	HW Plan	D_2's pending cmds target stale A_6. Requests DAG redaction.
5	Exec	Cancels pending D_2 cmds (closure). In-flight MOVE ($C_5 \to B_2$) unaffected.
6	HW Plan	Expands D_1 + revised D_2 plans into batches.
7	Safety	Certifies against surviving reservations. Accepts.
8	Exec	Dispatches D_1 and revised D_2 commands.

Figure 2: Replan cascade when high-priority D_1 preempts in-progress D_2. The Carrier Coordinator selects the carrier and replans in a single evaluation cycle (steps 2–3). In-flight commands (step 5) are unaffected; safety is maintained by recertification (step 7).

Properties. *Localized scope*: only D_2's pending commands are affected. *Carrier-level stability*: only D_2's temporary placements are revised; D_2's primary extraction (in flight) is untouched. *Safety*: cancelled reservations are released before new ones are certified (P3, P4). *Deadlock freedom*: D_1 plans first; D_2 is rerouted, not blocked. *Degradation*: if no reroute exists, D_2 is deferred.

The same cascade structure applies to demand cancellations (carrier plan teardown, re-storage via the Carrier Coordinator's level-health mechanism) and bot faults (carrier-position discrepancy triggers carrier-level replan from the new layout). In every case, replan scope is bounded by the DAG closure of the affected commands, and the safety argument of Section 5.6 applies to the replacement commands.

8 Worked Example: Cross-Level Shared Elevator Trip

We trace a cross-level scenario to illustrate how bot and elevator commands coexist in the DAG, how shared trips use preconditions rather than dependencies, and how the elevator movement gate ($S(\sigma)$ item (v)) provides coordination.

Setup. Demand D_1 requires carrier C_3 (level 1) to be presented at balcony B_2 (level 2). A non-carrier charging need requires bot b_4 (level 1) to charge at a station on level 2. The hardware planner determines that both can share a single elevator trip. Plan segment P_1 (for D_1) is

	DAG structure (two plan segments)
P_1	MOVE$_1$ (bot b_1 onto elev mod E_1) → ELEV_MOVE (L_1→L_2) → MOVE$_2$ (exit E_1 at L_2) → MOVE$_3$ (→ B_2). ELEV_MOVE.*deps* = [MOVE$_1$]. MOVE$_2$.*deps* = [MOVE$_1$], *precond* = ELEV_AT_L2.
P_2	MOVE$_4$ (bot b_4 onto elev mod E_2) → MOVE$_5$ (exit E_2 at L_2) → MOVE$_6$ (→ charger). MOVE$_5$.*deps* = [MOVE$_4$], *precond* = ELEV_AT_L2. No cross-batch *deps*.
	Dispatch sequence (nominal)
1	MOVE$_1$, MOVE$_4$ dispatched in parallel (bots board elevator).
2	Both complete. Dispatcher evaluates ELEV_MOVE: elevator movement gate checks no bot has in-flight MOVE to/from M_{elev}. Gate passes. ELEV_MOVE dispatched.
3	Elevator arrives at L_2. MOVE$_2$ and MOVE$_5$: deps satisfied, precondition ELEV_AT_L2 satisfied. Occupancy gate passes. Both dispatched.
	Failure: P_1 cancelled while bots aboard
4	D_1 cancelled. P_1 redacted: ELEV_MOVE, MOVE$_2$, MOVE$_3$ cancelled (pending). MOVE$_1$ already completed—b_1 remains on E_1.
5	P_2 *unaffected*—no deps cross the boundary.
6	MOVE$_5$.*precond* ELEV_AT_L2 fails at dispatch (no ELEV_MOVE exists). Returned to planner.
7	Planner issues new ELEV_MOVE (L_1→L_2) in a fresh batch P_3, certified against surviving reservations. Both b_1 (orphaned, on E_1) and b_4 (on E_2) ride. Carrier Coordinator's level-health mechanism generates a re-storage plan for b_1's orphaned carrier.

Figure 3: Cross-level shared elevator trip (two plan segments). P_2 uses preconditions, not dependencies, for elevator state; when P_1 is cancelled (step 4), P_2 is unaffected and recovers via the standard precondition-failure repair path (steps 6–7). The elevator movement gate (step 2) naturally waits for all actively-boarding bots before dispatch.

certified first; plan segment P_2 (for the charging need) is certified independently. Figure 3 traces the resulting DAG and failure scenario.

Properties. *Safety*: the elevator movement gate prevents dispatch while any bot is mid-transition; b_1 and b_4 are fully aboard before departure. *Independence*: P_2 has no lifecycle coupling to P_1; cancellation of P_1 does not cascade. *Recovery*: the precondition-failure repair path produces a new elevator trip; no special shared-trip recovery mechanism is needed. *Orphan handling*: b_1, stranded on the elevator after D_1 cancellation, rides the recovery trip; the Carrier Coordinator's level-health mechanism plans re-storage of the orphaned carrier.

9 Open Problems

Formal bot-level liveness composition. A4 is discharged informally via well-formed infrastructure [11] and PIBT [5], but formally composing carrier-level and bot-level guarantees—especially under cross-level transit where rearrangement, elevator scheduling, and pathfinding interact—remains open. The elevator complicates the argument: it is a shared resource whose availability is not a graph-structural property but a scheduling outcome, and the elevator movement gate (item (v)) introduces dispatch-time holds that are not captured in existing well-formedness predicates. WinC-MAPF [13] offers completeness guarantees for windowed bot-level planning and may provide a path toward discharging A4 formally.

Exogenous event rate and trigger frequency. The repair budget (Section 5.7) requires $f \cdot T_{\text{plan}} < 1$ on average at the bot level. At the carrier level, the two-tier trigger classification and coalesce mechanism (Section 4.2) reduce cycle frequency under sustained throughput while preserving detection completeness. What remains open: characterizing the maximum tolerable Tier 1 trigger rate as a function of re-evaluation cost and fleet size, and calibrating the coalesce window against detection latency requirements.

Resource-freeing proof tightening. The Carrier Coordinator internalizes the resource-freeing scheduling obligation (Section 3.4): when a scarce resource (e.g., a balcony) is saturated, the coordinator escalates re-storage priority and plans the freeing operation jointly with the blocked presentation. Detection completeness is argued by exhaustive case analysis over replanning triggers (Section 4.2), and planning cycles are serialized (Section 4.1), eliminating inter-cycle consistency concerns. What remains open: the induction step showing that escalated re-storage plans satisfy the same deadlock-freedom invariants as demand-driven plans (they enter the same priority-ordered pipeline, but the argument has not been mechanized), and a formal bound on re-storage plan feasibility under the empty-cell invariant.

Non-carrier task path: charging. Non-carrier operations bypass the Carrier Coordinator and reach the Hardware Planner via a separate path. For charging, a Battery Monitor generates tasks at two priority levels: *critical* (battery below threshold or forced charge) at highest priority above all carrier work, and *opportunistic* (idle bots) at lowest priority, interruptible when carrier work arrives. Battery exhaustion violates $E(\sigma)$ and creates an immovable obstacle blocking critical resources; the site model guarantees sufficient charging stations for sustainable operation (Section 6). The critical interaction with A4 is that the number of bots simultaneously charging must leave enough bots available for carrier-step realization; the site-level station sufficiency assumption and the critical-threshold margin together bound this interaction. Onboarding and offboarding task paths remain open.

Fleet sizing and the minimum viable fleet predicate. The minimum fleet predicate (Section 5.5) bounds

per-level bot count for sustained liveness. Two issues remain open: (i) the bound assumes uniform demand distribution across levels; non-uniform patterns may require non-uniform fleet placement, and the interaction between demand routing and fleet distribution is unexplored; and (ii) for elevator-equipped deployments, bots may transit between levels to rebalance the fleet in response to demand shifts, but fleet-rebalancing elevator trips compete with carrier transit for elevator time, and the priority interaction has not been formalized. The single-bot-per-level case is correct but degrades liveness during charging; whether $n_{\min}(\ell) \geq 2$ should be a hard configuration constraint or a soft recommendation with an explicit latency-degradation acknowledgment is a deployment policy decision to be surfaced in the site-model validation report.

Demand Model and Carrier Coordinator interface maturity. The Demand Model maintains a demand-supply bipartite graph and emits constrained demand views with scored candidate carrier sets; the Carrier Coordinator jointly selects carriers and plans rearrangements. The liveness argument assumes that this pipeline produces a well-formed demand stream satisfying A1–A2. Two interface questions remain open: (i) the cross-level transit walkthrough (the Carrier Coordinator fully decomposes cross-level carrier movements into per-module steps, including routing through elevator ingress and egress modules; the Hardware Planner expands the mechanism-specific steps into commands—the full-stack interaction has not been stress-tested end-to-end); and (ii) the joint carrier selection algorithm (Section 4.3 sketches the escort-based extraction and $O(C^N)$ selection; pruning strategies and the greedy fallback require empirical validation against realistic topologies).

Sustained overload. When A1 is violated, the liveness guarantee lapses. Practical mitigations (admission control, demand shedding, operator escalation) are domain-specific and outside the formal argument.

Well-formedness predicate and site model. Bot-level liveness relies on well-formed infrastructure [11]; carrier-level liveness relies on PBS feasibility [8] (sufficient free cells in a connected graph). Both results assume graph-connectivity properties that must hold for the *deployed* module topology, which may include one-way edges (e.g., wing exits toward autobahn only) and designated zones (storage, autobahn, balcony, elevator ingress/egress). A deployment-specific *site model* defines this topology and its constraints. The site model also declares whether the deployment is *elevator-equipped* ($|M_{\text{elev}}| > 0$) or *level-isolated* ($|M_{\text{elev}}|=0$). For level-isolated sites the per-level self-sufficiency predicate (Section 2) must be validated: each level must have at least one assigned bot, at least one charging station, and at least one balcony; this is a configuration-time check, not a runtime guarantee, and must be included in the validation suite alongside well-formedness. Five issues remain open: (i) the well-formedness predicate must be stated precisely for warehouse graphs with one-way edges and zone designations, extending the original definition [11]; (ii) the predicate must be validated at configuration time by the site model, so that liveness is guaranteed only for topologies that pass validation; (iii) the PBS free-cell guarantee must account for modules that are zone-restricted (e.g., balcony modules are not valid temporary carrier storage); (iv) the interaction between one-way edges and the Carrier Coordinator's displacement routing (blockers may not be returnable via the same path) requires analysis to ensure that rearrangement plans remain feasible under directional constraints; and (v) for level-isolated sites, the per-level well-formedness predicate must extend the single-level result to cover all levels simultaneously, confirming that independent per-level liveness arguments compose without cross-level interference.

Startup reconciliation. On system startup (including restart after a crash), command dispatch is gated until the following conditions hold simultaneously: (1) every deployed bot has completed the two-phase setup handshake ($\iota(b)$=`t` for all $b \in B$); (2) for elevator-equipped deployments, the elevator WebSocket subscription has delivered its first state message (e_k=`t`; this condition is vacuous for level-isolated deployments); and (3) the carrier map π_C has been reconstructed by cross-referencing $\lambda(b)$ values against carrier positions from hardware ground truth. Until all applicable conditions hold, neither $S(\sigma)$ nor $E(\sigma)$ can be verified against a complete state and no commands are dispatched.

Two-phase handshake. Each bot, on power-on or fault-recovery, publishes a setup message containing its complete physical state: position, orientation, battery $\beta(b)$, lift state $\lambda(b)$, charging state $\chi(b)$, driveable status, and $last\_cmd\_id(b)$ (the command ID of the last successfully completed command before the current session, or `0xFF...F` on first boot). The coordinator sends a setup-confirmation acknowledgment; the bot sets $\iota(b) \leftarrow$ `t` and begins accepting operational commands. Prior to this, the bot silently drops all operational commands.

Last command ID and restart safety. If the coordinator dispatched command c to bot b before b faulted, and b's setup message reports $last\_cmd\_id(b) \neq id(c)$, then c was not completed. The coordinator must treat c as failed and re-plan; the command cannot be assumed complete even if b's reported position appears consistent with completion. The interaction between reservation cleanup (reservations held until terminal state) and this restart-recovery path requires specification.

In-session restart and cleanup commands. A bot restarting mid-session must first reach a neutral state before sending its setup message: if $\chi(b)$=`t`, it issues PLUG_INV first; if $\lambda(b)=\uparrow$, it issues LOWER first. These cleanup commands are accepted while $\iota(b)$=`f`. Whether cleanup commands require new certification, what happens if a cleanup command itself fails, and the reservation-table implications of cleanup in the faulted-bot FMEA path (F-05, F-47, F-48) remain open.

Startup inconsistencies. World-state reconstruction may encounter two cases requiring explicit policy. (*A*) If

$\lambda(b)=\uparrow$ but no carrier is found at $\pi_B(b)$, the carrier was dropped at an unknown location; treat bot b as faulted until manually resolved. (B) If a bot does not send a setup message within timeout T_{init}, the coordinator must either exclude the non-reporting bot from the fleet and proceed with reduced capacity, or hold all dispatch until operator confirmation. Both policies remain open design decisions.

Observation model and localization error bounding. The observation model (Section 2) establishes that $\hat{\sigma}$ agrees with σ on bot positions at every dispatch decision point, by combining completion-only feedback with per-bot command serialization. This argument assumes faithful localization: the bot's physical position does not diverge from its last-known module between dispatch and completion reporting. A silent localization failure—where the bot moves inconsistently with its commanded trajectory without reporting a hardware fault—would violate the agreement property: the occupancy gate would check $\hat{\sigma}$ and find the source module free, dispatching into a physically occupied module. Bounding the localization error envelope, characterizing under what physical conditions silent divergence is possible, and identifying runtime monitoring capable of detecting it remain open.

Light curtain and elevator dispatch interaction. Curtain timing parameters ($\tau_{\text{mute\_max}}$, τ_{breach}) impose hard real-time constraints on traversal commands near protected regions. Curtain state is also a dispatch precondition for ELEV_MOVE (a breach propagates to e_s = *safety*), coupling curtain timing to elevator scheduling. How certification-time duration bounds relate to curtain timing, how overruns near protected regions are handled, and how the elevator movement gate interacts with curtain-induced dispatch holds requires further analysis.

Elevator movement gate under temporal uncertainty. The elevator movement gate ($S(\sigma)$ item (v)) naturally holds the ELEV_MOVE while any bot is actively boarding. Under temporal uncertainty, this hold is bounded by T_{cmd} (single command duration). However, the interaction between this hold and planned departure timing—particularly for shared trips where multiple bots must board within a planned window—is under-specified. How much timing margin to allocate for late boarders before accepting that they will miss the trip is an open calibration question.

Runtime occupancy gate: temporal calibration and scheduling stability. The dispatcher's physical occupancy gate prevents dispatch into a physically occupied module even when certification-time predictions have expired (Section 5.6). The gate is correct by construction, but its effect on scheduling stability is uncharacterized. When a command duration overrun causes the gate to block a downstream command, other commands depending on that module's vacancy are also held—potentially cascading across multiple independently-certified plans. Two related questions are open: (i) characterizing the expected gate-trigger rate as a function of command duration variance, so that the fraction of dispatches that block can be bounded under realistic timing distributions; and (ii) analyzing whether cascading holds can destabilize the overall schedule—that is, whether a single overrun can cause a chain of blocks whose aggregate delay exceeds any pre-committed timing budget. A temporal calibration strategy—adding padding between certified reservation intervals to reduce gate-trigger frequency at a bounded throughput cost—is needed but not yet specified.

Repair loop: remaining gaps. Section 5.7 bounds the repair loop under the condition $f \cdot T_{\text{plan}} < 1$, but the degradation posture when this condition is violated (hold affected bots, continue unaffected operations, alert operator) is not yet specified. Event coalescing at the bot level—batching multiple exogenous events that arrive during a single planning cycle into one repair iteration—would reduce f effectively but requires analysis to ensure that coalesced events do not mask safety-relevant state changes.

Cross-level elevator-balcony deadlock under saturation. Same-level re-storage feasibility is argued in Section 3.3 via the PBS result [8], reducing the concern to a Site Model connectivity check. What remains open: verifying that the connectivity requirement (storage zone connected under one-way edges and zone restrictions) is satisfiable for practical warehouse topologies, and that the empty-cell count k is sufficient to absorb concurrent re-storage operations from multiple demands on the same level.

Summary. Each component has strong empirical grounding: MAPF-DECOMP [2] scales to hundreds of agents; RHCR [3] achieves real-time lifelong coordination at warehouse scale; PIBT [5] provides deadlock-freedom on well-formed graphs. Their synthesis—vertical decomposition with DAG-based localized replanning and single-authority certification—offers a practical, composable path to tractable online multi-agent coordination with explicit safety and conditional liveness guarantees.

References

[1] J. Yu and S. M. LaValle, "Structure and intractability of optimal multi-robot path planning on graphs," in *Proc. AAAI*, 2013, pp. 1443–1449.

[2] B. Li and H. Ma, "Double-deck multi-agent pickup and delivery: Multi-robot rearrangement in large-scale warehouses," *IEEE RA-L*, vol. 8, no. 7, 2023.

[3] J. Li, A. Tinka, S. Kiesel, J. W. Durham, T. K. S. Kumar, and S. Koenig, "Lifelong multi-agent path finding in large-scale warehouses," in *Proc. AAAI*, 2021.

[4] J. Li, W. Ruml, and S. Koenig, "EECBS: A bounded-suboptimal search for multi-agent path finding," in *Proc. AAAI*, 2021.

[5] K. Okumura, M. Machida, X. Défago, and Y. Tamura, "Priority inheritance with backtracking for iterative multi-agent path finding," *Artificial Intelligence*, vol. 310, 2022.

[6] K. R. Gue and B. S. Kim, "Puzzle-based storage systems," *Naval Research Logistics*, vol. 54, no. 5, pp. 556–567, 2007.

[7] T. Raviv, Y. Bukchin, and R. de Koster, "Optimal retrieval in puzzle-based storage systems using automated mobile robots," *Transportation Science*, vol. 57, no. 2, pp. 424–443, 2023.

[8] E. Shirazi and M. Zolghadr, "An item retrieval algorithm in flexible high-density puzzle storage systems," *Applied System Innovation*, vol. 4, no. 2, article 38, 2021.

[9] W. Hönig, S. Kiesel, A. Tinka, J. W. Durham, and N. Ayanian, "Persistent and robust execution of MAPF schedules in warehouses," *IEEE RA-L*, vol. 4, no. 4, 2019.

[10] B. Alpern and F. B. Schneider, "Defining liveness," *Information Processing Letters*, vol. 21, no. 4, pp. 181–185, 1985.

[11] H. Ma, J. Li, T. K. S. Kumar, and S. Koenig, "Lifelong multi-agent path finding for online pickup and delivery tasks," in *Proc. AAMAS*, 2017, pp. 837–845.

[12] M. Abadi and L. Lamport, "Conjoining specifications," *ACM Trans. Prog. Lang. and Systems*, vol. 17, no. 3, pp. 507–535, 1995.

[13] R. Veerapaneni, M. S. Saleem, J. Li, and M. Likhachev, "Windowed MAPF with completeness guarantees," in *Proc. AAAI*, 2025.

[14] B. P. Gerkey and M. J. Matarić, "A formal analysis and taxonomy of task allocation in multi-robot systems," *Int. J. Robotics Research*, vol. 23, no. 9, pp. 939–954, 2004.

[15] B. Zou, X. Xu, Y. Gong, and R. de Koster, "Evaluating battery charging and swapping strategies in a robotic mobile fulfillment system," *European J. Operational Research*, vol. 267, no. 2, pp. 733–753, 2018.