Architectural Paradigms for Early-Game Logistics: Solving the RCL 1 and RCL 2 Creep-to-Creep Transfer Bottleneck

The transition of an autonomous colony within the Screeps execution environment from a nascent, resource-starved state at Room Controller Level 1 (RCL 1) to a stable, infrastructure-supported economy represents one of the most critical challenges in distributed simulation engineering. While the fundamental mechanics of harvesting energy and upgrading controllers are conceptually straightforward, the physical constraints of the game's asynchronous, tick-based architecture often generate severe logistical bottlenecks that paralyze colony growth. One of the most pervasive inefficiencies observed in intermediate artificial intelligence codebases is the "Last Mile Problem," colloquially known as the "Fetch Loop," which manifests prominently during RCL 1 and RCL 2. In this highly specific scenario, a colony possesses a surplus of transporters (haulers) and abundant energy at its source nodes, yet worker creeps (builders and upgraders) consistently waste significant portions of their Time to Live (TTL) and Central Processing Unit (CPU) cycles traversing the map to fetch energy directly from distant source containers. This occurs because centralized infrastructure, such as a StructureStorage or spawn-adjacent containers, does not yet exist.

This comprehensive engineering report investigates the advanced architectural frameworks utilized by top-tier algorithms to entirely eliminate this inefficiency. By abandoning legacy "Search and Rescue" scripting paradigms in favor of deterministic "Command and Control" logistics brokers, autonomous agents can enable highly efficient, direct creep-to-creep energy transfers. The ensuing analysis will dissect the mathematical thermodynamics of early-game mining, the Application Programming Interface (API) constraints governing simultaneous actions, the implementation of heuristic "Mobile Container" models for open-offer scenarios, and the integration of mobile creeps as dynamic requesters within Gale-Shapley stable matching algorithms.

The Computational Mechanics and Constraints of the Early Game

To architect a sustainable solution to the early-game logistics bottleneck, the physical and mathematical constraints of the Screeps server infrastructure must first be rigorously defined. User code executes within a Node.js virtual machine isolate, managed by the isolated-vm library to ensure strict resource accounting and security sandboxing. This environment processes logic in a discrete, deterministic lifecycle where the progression of time is measured in game ticks.[1]

The Serialization Wall and Inversion of Control

A critical vulnerability in early-stage bot development is the over-reliance on the native Memory object for per-tick decision-making. The global Memory object in Screeps is not a live database connection; rather, it is a unified JSON string backed by MongoDB and synchronized via Redis.[1] Parsing this string via JSON.parse at the beginning of the script, and serializing it via JSON.stringify at the end, incurs a severe CPU tax.[1] Legacy architectures typically utilize a monolithic, procedural loop where each individual creep independently queries its environment utilizing functions like room.find(FIND_DROPPED_RESOURCES). If twenty workers simultaneously execute this search, the server processes twenty identical spatial queries, resulting in a devastating $O(N \cdot M)$ computational complexity that rapidly exhausts the player's CPU bucket.[1]

Top-tier architectures overcome this computational ceiling by implementing an Inversion of Control (IoC) hierarchy. A centralized "Overlord" or "Kernel" process performs a single, comprehensive spatial scan of the room, caches the state in the global V8 Heap, and subsequently assigns explicit task objects to individual "Zerg" (creep) units.[1] This fundamental decoupling is the prerequisite for advanced logistics. Individual workers should never autonomously "decide" where to fetch energy based on localized, greedy algorithms. Instead, a central broker must evaluate the global state of the colony and calculate the most efficient distribution vectors mathematically.[1]

Simultaneous Execution Pipelines and API Mechanics

A major architectural advantage of the Screeps engine that facilitates direct creep-to-creep interaction is the simultaneous execution pipeline. The server evaluates player intents in a bipartite structure: player script calculation occurs first, followed by command processing.[1] Actions that do not share underlying engine dependencies can be executed concurrently by the same creep within a single tick.[2]

For example, a hauler creep can execute creep.moveTo(target) and creep.transfer(worker, RESOURCE_ENERGY) in the exact same tick.[2] However, the engine strictly dictates that a creep cannot execute transfer() multiple times to multiple targets in a single tick, nor can it execute conflicting intents like build() and harvest() simultaneously.[2] Crucially, if a worker creep is actively executing creep.build(site) and an adjacent hauler executes creep.transfer(worker, RESOURCE_ENERGY), both actions succeed in the backend processing phase. The worker utilizes its current internal energy store to apply build progress, and at the culmination of the tick, its capacity is replenished by the hauler's transfer intent.[2] This pipeline behavior is the mechanical foundation that allows for a continuous, uninterrupted stream of construction without requiring the builder to pause its operations to receive logistics support.

RCL 1: Pioneer Optimization and Metabolic

Stabilization

At RCL 1, the colony is severely restricted to a single Spawn structure with a maximum energy capacity of 300 units.[1] The standard energy source in an owned room generates 3,000 energy every 300 ticks, equating to an exact, unyielding output of 10 energy units per tick.[1] Because the 300-energy limit precludes the creation of highly specialized, massive units, the optimal strategy at this stage relies entirely on "Pioneer" creeps.

Mathematical derivation demonstrates that a pioneer body consisting of ``, costing exactly 250 energy, is the most efficient configuration available under the capacity cap.[1] These units act as universal agents, completing a cyclic, self-contained task of traveling to the source, harvesting, and returning to the Spawn or the Controller.[1]

The primary threat at this nascent stage is "Metabolic Collapse"—a deadlock scenario where energy is spent entirely on upgrading the Controller, leaving the Spawn empty and unable to replace aging pioneers as their 1,500-tick lifespans expire.[1] Top-tier logic bypasses this vulnerability by injecting a strict state override into the behavior of the initial units. The first two pioneers spawned are forced to prioritize the transfer() action directed at the Spawn until a safe, redundant buffer of 250 energy is consistently established.[1] Only after this metabolic stabilization is achieved does the Overlord permit the workforce to focus on Controller saturation, which mathematically requires precisely 6.5 pioneers (rounded to 6 or 7) to process the 10 energy-per-tick yield efficiently.[1]

The RCL 2 Paradigm Shift and the Last Mile Problem

Upon accumulating 200 energy in the Room Controller, the colony advances to RCL 2, unlocking 5 Extensions and expanding the maximum spawn energy capacity to 550.[3] This morphological inflection point dictates an immediate architectural shift from the "Universal Pioneer" to a highly decoupled "Miner and Hauler" paradigm.[1]

Drop Mining Versus Container Maintenance

At RCL 2, static miners equipped with 5 WORK parts and 1 MOVE part (costing exactly 550 energy) are deployed directly adjacent to energy sources. These units never move again, dedicating their entire existence to harvesting the maximum 10 energy per tick.[1] Because StructureContainer units require 250 energy to build and significant worker time to construct, top-tier architectures temporarily employ a "Drop Mining" methodology. The static miner continuously executes the harvest() intent, allowing excess energy to fall onto the terrain tile beneath it.[1]

The Screeps engine simulates thermodynamic decay for dropped resources. Energy left on the ground decays at a rate of exactly $\lceil amount/1000 \rceil$ per tick.[1] For a source yielding 10

energy per tick, this acts as a persistent 10% tax on the colony's gross income.[1] Specialized haulers composed entirely of CARRY and MOVE parts are spawned to ferry this energy back to the core base as rapidly as possible to mitigate this decay.[1] Eventually, builders construct containers beneath the miners, eliminating decay entirely in exchange for a physical maintenance cost of 0.1 energy per tick, as the container loses 5,000 hit points every 100 ticks.[1]

The Genesis of the Logistical Inefficiency

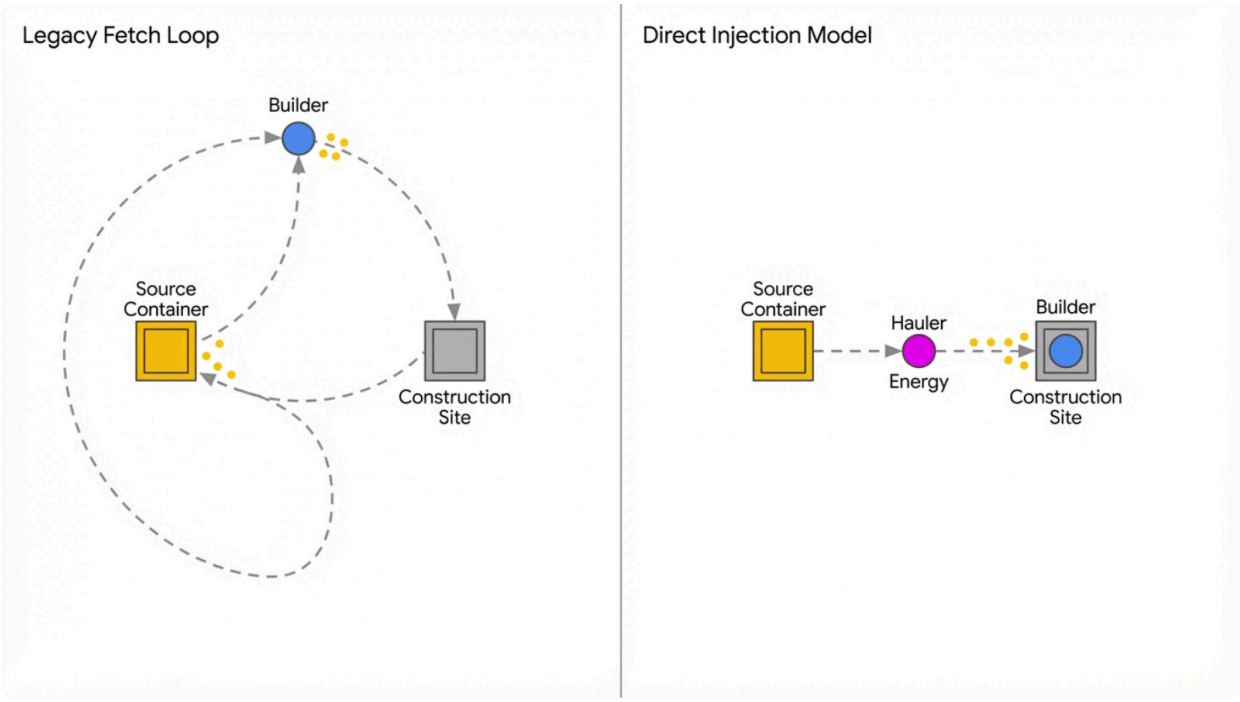
The specific problem identified in the architectural query—workers abandoning their construction sites to walk all the way to source containers—arises precisely during this volatile RCL 2 transition. The logistics network during this phase is structurally imbalanced, characterized by a massive disparity between source output and core storage capacity.

The inefficiency unfolds sequentially. First, static miners extract 10 energy per tick, rapidly filling the drop piles or newly built source containers. Second, a highly optimized fleet of haulers efficiently moves this energy to the Spawn and the 5 Extensions. Third, a "Demand Vacuum" occurs. Once the Spawn and Extensions are completely full (holding a mere 550 energy combined), the haulers have no valid target structures.[3] Because a central StructureStorage does not yet exist (as it is not unlocked until RCL 4), the haulers are forced into an idle state, despite carrying full loads of energy.[3]

Meanwhile, builder creeps are actively draining their internal energy reserves to construct critical infrastructure, such as the aforementioned extensions or the road network. When a builder's internal store reaches zero, legacy procedural logic instructs the creep to find the nearest source of energy. Because the Spawn and Extensions are strictly reserved for unit production and often restricted from worker withdrawal, the worker's logic targets the source containers. This initiates a catastrophic, highly inefficient transit cycle.[5]

If a builder creep is required to spend 30 ticks walking to a source container, 2 ticks withdrawing energy, and 30 ticks walking back to its designated construction site, its effective duty cycle plummets to unacceptable levels. The fundamental architectural solution is to redirect the currently idle haulers to deliver energy directly to the workers, effectively treating the workers as dynamic, mobile extensions of the colony's storage grid.

Resolving the RCL 2 Logistics Bottleneck



In the Legacy Fetch Loop, builders waste up to 70% of their lifespan in transit. The Direct Injection Model utilizes the Logistics Broker to assign haulers to stationary workers, achieving near 100% building uptime.

Architectural Solutions for Creep-to-Creep Transfers

Top-tier codebases approach the direct-delivery problem through two primary architectural patterns. The first is a heuristic, state-machine driven approach known as the "Mobile Container" pattern, designed specifically for early-game open-offer scenarios. The second, more advanced method is the integration of a Global Logistics Broker utilizing bipartite matching algorithms.

Heuristic Approach: The "Mobile Container" Pattern

During the RCL 1 to RCL 2 transition, before a highly sophisticated logistics broker is fully operational or computationally viable within the CPU limits, developers often implement a transitional architecture known as the "Mobile Container" or "Direct Delivery" model.[7]

As documented in advanced field journals and community logic analyses, when a centralized StructureStorage is unavailable and spawn extensions reach maximum capacity, the standard hauler behavior of returning to an idle state creates the aforementioned bottleneck.[7] To resolve the scenario where "open offers exist with no requests," the hauler is explicitly instructed to

intercept active workers rather than waiting for structural deficits.

The logic flow for this heuristic model operates through a strict Finite State Machine (FSM):

1. **Surplus Detection:** The hauler verifies that its carry.energy is full and that no primary StructureSpawn or StructureExtension targets require filling.
2. **Target Acquisition Expansion:** If the structural target array evaluates to a length of zero, the hauler executes a fallback search for allied units utilizing FIND_MY_CREEPS. This search is heavily filtered to isolate specific roles (e.g., role == 'builder' or role == 'upgrader') and is sorted by available capacity (creep.store.getFreeCapacity() > 0).[7]
3. **Interception Routing:** The hauler calculates a path to the highest-priority worker and initiates a moveTo intent.
4. **Settle and Transfer:** Once the hauler achieves adjacency (range of 1), it effectively becomes a "mobile container." It continuously executes the transfer() intent directed at the worker until its own capacity is entirely depleted or the worker achieves maximum capacity.[7]

While this model is highly effective at keeping builders stationary and maximizing their construction duty cycles, it remains a heuristic workaround. It relies on individual haulers performing localized spatial searches, which inadvertently increases CPU overhead if multiple haulers are actively searching the Game.creeps object for workers simultaneously.

Logistics Model	Decision Maker	Target Types	CPU Complexity	Scalability
Legacy Fetch Loop	Worker (Pull)	Structures, Dropped Energy	High ($O(N \cdot M)$)	Poor (Fails at RCL 3+)
Mobile Container	Hauler (Push)	Structures, Allied Creeps	Medium ($O(H \cdot$)	Moderate (Viable RCL 1-4)
Logistics Broker	Kernel (Match)	Registered Requests (Any)	Low ($O(R \log R$)	Infinite (GCL 30+)

Enterprise-Grade Resolution: The Global Logistics Broker

The absolute pinnacle of Screeps logistics architecture entirely abandons independent creep decision-making. Instead, it utilizes a Global Logistics Broker managed by a stable matching

algorithm, specifically the Gale-Shapley method, executed within the global heap.[1]

Establishing the Request and Provider Interface

In this advanced architecture, the colony's economy is modeled as a massive bipartite graph connecting "Providers" (entities holding excess energy) and "Requesters" (entities requiring energy to function).[1] Providers typically include miners, source containers, dropped energy piles, and tombstones. Requesters are traditionally defined as structures: spawns, extensions, towers, and the room controller.[1]

To solve the RCL 2 worker starvation problem natively within this system, the architecture must be fundamentally modified to allow mobile creeps to register as Requesters.[9] In advanced frameworks, such as the open-source Overmind architecture, LogisticsTarget or TransportRequestTarget interfaces are explicitly extended to accept Creep objects as valid, routable destinations alongside standard structures.[9]

State-Machine Driven Registration

In a brokered system, workers operate on a rigid Finite State Machine. When a builder completes a task, or when its internal energy drops below a specific operational threshold (e.g., < 20%), it transitions into an AWAITING_SUPPLY state.

Instead of autonomously moving toward a source, the builder halts its movement to preserve pathfinding stability and submits a TransportRequest to the central Logistics Broker. This request is an ephemeral object pushed into a global array, containing:

1. The Builder's unique string identifier.
2. Its current geometric X and Y coordinates.
3. The exact magnitude of energy required, calculated via creep.store.getFreeCapacity().
4. A priority score calculated dynamically based on the critical nature of its construction site. For instance, a builder assigned to a StructureTower construction site receives a higher priority score than a builder paving a generic road tile.

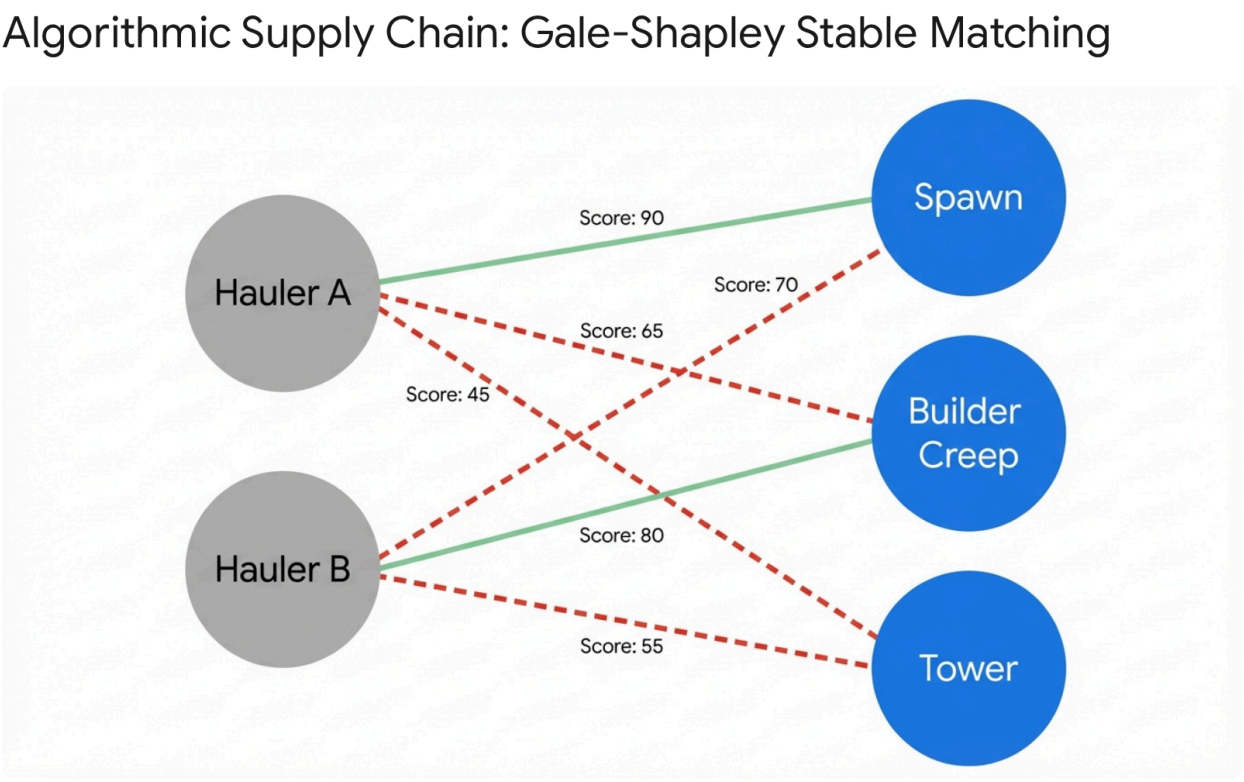
The Gale-Shapley Stable Matching Resolution

At the culmination of the calculation phase every tick, the Logistics Broker compiles all active requests (from both structures and mobile creeps) and all available free haulers. The Gale-Shapley algorithm then executes a highly optimized "Propose and Reject" sequence [1]:

1. **Proposal Generation:** Each free hauler calculates a mathematical preference list of all active requests. This list weights requests based on urgency (Priority) multiplied by the inverse of the pathfinding distance.[1]
2. **Tentative Matching:** Haulers "propose" to their absolute most preferred, available target.
3. **Conflict Resolution:** If multiple haulers propose to the same builder, the builder mathematically "rejects" the less optimal hauler (e.g., the hauler that is further away or carrying insufficient energy), forcing the rejected hauler to immediately propose to its next

highest preference.[1]

Because the Gale-Shapley algorithm mathematically guarantees stability, no two haulers will ever race toward the same builder. This elegant resolution entirely eliminates the "Energy Racing" phenomenon that cripples the CPU limits of legacy bots, ensuring that the supply chain operates with absolute precision.[1]



The Logistics Broker resolves energy distribution by ranking targets. When an Extension and a Builder both request energy, haulers propose based on a combined score of priority and proximity, eliminating 'energy racing'.

Managing Predictive State and the Effective Store

A fundamental risk of registering mobile creeps as requesters within a centralized broker is the temporal latency between intent generation and task execution. If a builder requests 50 energy, and a hauler carrying 50 energy is dispatched from a location 15 tiles away, the builder will remain catastrophically idle for 15 ticks awaiting delivery.

To counteract this latency, the architecture utilizes the "Effective Store" (S_{eff}) ledger system.[1] The effective store is defined mathematically as the current physical store plus incoming

reservations, minus outgoing reservations.

$$S_{eff} = CurrentStore + IncomingReservations - OutgoingReservatio$$

When the broker matches the hauler to the builder, it immediately updates the builder's S_{eff} to indicate that the energy is "in flight".[1] Consequently, subsequent iterations of the matching algorithm on following ticks will observe that the builder's needs are theoretically met, preventing the broker from dispatching a redundant second hauler.[1]

Furthermore, to completely eliminate the 15-tick idle penalty, the worker's FSM must implement Predictive Requesting. Rather than waiting until its physical energy store reads exactly zero, the builder registers a TransportRequest when its energy drops below the projected consumption rate required during the hauler's estimated travel time.[1] If a builder consumes 5 energy per tick, and the nearest idle hauler is mathematically 10 tiles away, the builder preemptively registers a request the moment its internal store drops to 50 energy. The hauler is dispatched instantly and arrives precisely as the builder exhausts its final unit of internal energy, ensuring a flawless 100% duty cycle.

Handling the Interception Vector: Moving Targets and Kinetic Fatigue

While the Gale-Shapley broker elegantly solves the logical assignment problem, the physical execution phase presents a unique geometric challenge: intercepting a moving target. If a hauler executes creep.moveTo(builder) and the builder is simultaneously moving toward a new construction site, the standard native pathfinder is forced to constantly recalculate its route every tick. This endless recalculation generates massive CPU spikes, frequently causing the codebase to exceed its bucket allocation.[1]

Static Sinks and Path Caching

The most computationally efficient architectural solution is to artificially restrict the physical mobility of the requester creep during the rendezvous phase.

When a worker's energy falls below the predictive threshold and it submits a request to the broker, it enters a STATIC_BUILD state. The worker calculates the optimal geometric location to construct the maximum number of sites without moving (for example, the exact center of a 3×3 extension cluster).[1] Once positioned, the worker anchors itself to that specific coordinate and explicitly refuses to issue further moveTo() commands until its construction targets are completely exhausted or it requires an unavoidable repositioning.

Because the worker is now a mathematically static sink, the assigned hauler can safely utilize deep Path Caching.[1] The Logistics Network calculates the optimal path from the hauler to the static worker exactly once using the highly efficient C++ PathFinder.search() backend.[1] This generated path array is then serialized into a direction string (e.g., "33214") and cached in the volatile V8 Heap.[1] On subsequent ticks, the hauler blindly follows this cached string, reducing the per-tick movement CPU cost to a negligible fraction of a millisecond.

Once adjacent, the hauler executes the transfer() intent. If the hauler possesses surplus energy after completely filling the builder's capacity, it updates its status in the Logistics Broker, which immediately executes a rapid matching iteration to redirect the hauler to the next nearest priority sink.

Traffic Management and Recursive Shoving

Introducing heavy logistics haulers directly into the active construction zones of a developing RCL 2 base creates incredibly dense traffic scenarios. Base corridors are frequently designed to be 1-tile wide, leading to inevitable collisions between haulers bringing energy and pioneers or workers navigating to new sites. Relying on standard moveTo() logic causes creeps to stop and wait indefinitely when their path is blocked, crippling the entire supply chain.[1]

Top-tier architectures resolve this kinetic friction via Centralized Traffic Management, utilizing a "Shove" algorithm or Bipartite Graph Optimization.[1]

1. **Intent Registry:** Before executing any native movement commands, all creeps register their intended destination tile with the centralized Traffic Manager.[1]
2. **Hierarchy Assignment:** The manager evaluates and assigns strict priorities. A fully loaded hauler executing a direct delivery to a critical builder is assigned a higher priority weight than an idle worker.[1]
3. **Recursive Displacement:** If a high-priority hauler's intended path requires a tile currently occupied by a low-priority worker, the Traffic Manager intercepts the worker's logic loop. It overrides the worker's intent and forces it to issue a move() command to an adjacent, non-blocking tile.[1] Advanced systems use recursive methods like vacatePos to ensure the shoved creep doesn't simply push into another occupied tile, unspooling the collision safely.[1]
4. **Positional Swapping:** If a hauler and a worker are moving in direct opposition within a 1-wide corridor, the engine mathematically identifies the head-on collision. It intercepts both intents and issues simultaneous move() commands directed at each other's coordinates. The Screeps backend explicitly permits creeps to exchange places seamlessly in a single tick without throwing an ERR_NO_PATH or ERR_INVALID_TARGET error, completely neutralizing the collision.[1]

This unified, algorithmic traffic resolution ensures that the direct delivery of energy remains fluid and does not introduce secondary deadlocks into the colony's infrastructure.

The CPU and Memory Economics of Direct Transfer

Implementing direct creep-to-creep transfers via a Logistics Broker and traffic manager requires a fundamental paradigm shift in how bot memory is managed. Executing matching algorithms and recursive displacement logic natively via the persistent Memory object would instantly deplete the player's 20-300 CPU limit due to the sheer volume of JSON parsing required.[1]

Heap-First Data Persistence

To sustain the intense computational load of tracking mobile requesters and evaluating Gale-Shapley matrices every single tick, the architecture must transition entirely to a "Heap-First" methodology.[1]

Upon the initialization of the Node.js isolate (which occurs after a Global Reset or code upload), the script hydrates rich JavaScript class instances—such as LogisticsNetwork, TransportRequest, and Zerg wrappers—directly into the global scope.[1] For the remainder of the isolate's lifespan, which may last thousands of ticks, the broker operates exclusively in volatile RAM.

When a builder submits an energy request, it simply pushes a lightweight object reference into a global array. The hauler logic iterates over this array in nanoseconds. Only absolutely critical persistent state changes, such as a creep dying, an active combat engagement, or the room successfully leveling up, are flushed back to the Memory JSON string via JSON.stringify() at the end of the tick.[1] By strictly segregating transient logistics data from persistent memory, the architecture eliminates the serialization tax, freeing the necessary CPU cycles to execute the sophisticated interception and matching algorithms required for top-tier play.[1]

Conclusion: The Engineering of Autonomous Sovereignty

The comprehensive analysis of top-tier codebases reveals that solving the "Last Mile" problem of workers walking to distant source containers requires a complete architectural overhaul of the bot's foundational logic flow. Heuristic fixes, such as the "Mobile Container" pattern, provide adequate relief during the early stages of RCL 2, but true efficiency is only realized through mathematical optimization.

To successfully implement direct hauler-to-worker transfers and achieve tick-perfect progression, developers must adhere to the following strict engineering protocols:

1. **Abandon Search and Rescue:** Creeps must be stripped of independent decision-making. They must not autonomously scan for resources using expensive $O(N \cdot M)$ spatial queries like room.find(). All energy logic must be centralized within a Logistics Broker.

2. **Implement Stable Matching:** The system must utilize algorithms like Gale-Shapley to pair energy providers and requesters, weighing both geometric proximity and strategic priority to definitively eliminate energy racing.
3. **Register Mobile Requesters:** The architectural definition of a "Requester" within the logistics network must be explicitly extended to include Creep objects, specifically builders and upgraders whose internal capacity drops below a predictive threshold.
4. **Anchor the Sinks:** Workers must be programmed to act as mathematically stationary targets during the resupply phase, allowing haulers to utilize highly efficient, cached paths to reach them without triggering CPU-heavy path recalculations.
5. **Utilize the Global Heap:** All Transport Requests, pathing arrays, and process queues must be stored in the global Node.js environment to bypass the MongoDB serialization tax inherent to the native Memory object.

By treating the Screeps environment not merely as a game of individual scripts, but as a distributed operating system requiring rigorous resource allocation and precise intent management, a colony can achieve 100% duty cycles on its construction units. The flawless implementation of direct creep-to-creep logistics during the vulnerable RCL 1 and RCL 2 phases serves as the metabolic engine that rapidly accelerates the colony, bypassing the early-game bottlenecks to establish the automated, high-throughput empire of the endgame.

Works cited

1. Screeps Speedrunning Architecture Analysis.pdf
2. Simultaneous execution of creep actions - Screeps Documentation, accessed February 21, 2026, https://docs.screeps.com/simultaneous-actions.html
3. Control | Screeps Documentation, accessed February 21, 2026, https://docs.screeps.com/control.html
4. Static Harvesting - Screeps Wiki, accessed February 21, 2026, https://wiki.screepspl.us/Static_Harvesting/
5. Screeps - Guide to Logistic Creeps for Energy Transport - YouTube, accessed February 21, 2026, https://www.youtube.com/watch?v=tOmonUwf6pI
6. Need help storing energy in containers | Screeps Forum, accessed February 21, 2026, https://screeps.com/forum/topic/2394/need-help-storing-energy-in-containers
7. Screeps #8: Logistics Overhaul - Field Journal, accessed February 21, 2026, https://jonwinsley.com/notes/screeps-logistics-overhaul
8. code help for sorting creeps by energy | Screeps Forum, accessed February 21, 2026, https://screeps.com/forum/topic/971/code-help-for-sorting-creeps-by-energy
9. Releases · bencbartlett/Overmind - GitHub, accessed February 21, 2026, https://github.com/bencbartlett/Overmind/releases
10. Overmind - Ben Bartlett, accessed February 21, 2026, https://bencbartlett.com/overmind-docs/