Strategic Architectures for Accelerated Early-Game Progression in Screeps: An Engineering Analysis of RCL 1 to RCL 4

The transition of a decentralized autonomous colony within the Screeps execution environment from a nascent, resource-starved state to a high-throughput industrial powerhouse represents a paramount challenge in distributed simulation engineering. While the Screeps environment is fundamentally a real-time strategy simulation, the underlying engineering constraints mirror those of an embedded operating system or a high-frequency algorithmic trading platform. The primary constraints for an automated agent are not merely the in-game resources, but the rigid limits of Central Processing Unit (CPU) cycles, Heap persistence, and the computational overhead of memory serialization.[1] In the highly competitive landscape of the global control level (GCL) rankings, the margin between a rapidly expanding empire and a stalled colony often rests on the microscopic optimization of early-game progression.[1]

This comprehensive report provides an exhaustive architectural analysis of tick-perfect progression from Room Controller Level (RCL) 1 to RCL 4. It focuses on the mathematical optimization of pioneer creeps, the exact Return on Investment (ROI) of drop mining versus container mining at RCL 2, the spatial algorithms required for optimal extension placement, and the actionable code patterns necessary to construct a highly resilient "Bootstrapping Overlord." By abandoning legacy role-based scripts in favor of a deterministic, kernel-driven architecture, developers can achieve an unprecedented velocity of growth during the critical bootstrapping phase.

The Computational Mechanics of the Execution Environment

Before establishing the logic for early-game progression, the physical architecture of the server-side infrastructure must be accounted for. User code runs within a Node.js virtual machine isolate, managed by isolated-vm to ensure strict resource accounting and security sandboxing.[1] This environment operates on a discrete, deterministic tick-based lifecycle where the progression of time is measured in game ticks, tracked by the Game.time global property.[1]

The standard execution model involves two distinct stages: the player script calculation phase and the command processing phase.[1] During the calculation stage, the server loads the user's script, parses the persistent state, and executes the main loop. All commands issued during this time (e.g., creep.move(), creep.harvest()) are collected into a Redis-based intent queue for batch processing at the end of the tick.[1] If conflicting intents are scheduled, they are resolved according to predefined engine priorities.[2] Because intents are evaluated post-calculation, an

architecture must accurately predict the state of the world multiple ticks into the future.

The Serialization Wall and Heap Persistence

A critical vulnerability in early-stage bot development is the assumption that the execution environment is completely stateless between ticks.[1] In reality, the global scope—the V8 engine's Heap—persists across ticks unless the server explicitly tears down the isolate to reclaim resources or the user uploads new code.[1] The legacy architecture, prevalent in first-generation scripts, relies heavily on the Memory object, which is essentially a JSON string backed by MongoDB and indexed via Redis.[1]

Each tick, the server charges the user the CPU cost of parsing this string via JSON.parse(). As a colony expands, a monolithic reliance on Memory inflates the baseline CPU floor. For a multi-room empire, a 2MB memory file can consume upwards of 15ms of CPU merely to enter the execution loop, leaving insufficient processing time for advanced pathfinding, spatial analysis, and military coordination.[1]

To achieve tick-perfect speedrunning, the architecture must transition to a Kernel-driven, Heap-first model.[1] The "Bootstrapping Overlord" operates almost entirely within the global cache, storing instances of classes, CostMatrices, and pathing arrays in the Heap, and only serializing critical state transitions to Memory.[1] This paradigm shift reduces the JSON serialization tax by orders of magnitude, allowing the saved CPU to be reinvested into real-time market trading and aggressive expansion operations.

State Storage Feature	Legacy Approach (Procedural)	Bootstrapping Protocol (Kernel-Driven)	Impact on Performance
Primary Data Store	Memory object parsed every tick.	Global Heap Cache accessed instantly.	10x CPU reduction in JSON parsing.[1]
Data Lifecycle	Re-instantiated sequentially every tick.	Persistent classes updated via .refresh().	Minimizes V8 garbage collection overhead.[1]
Capacity Constraints	2MB hard cap on Memory string.	2MB Memory + ~256MB Heap + 100MB Segments.	Enables complex historical analysis and CostMatrix caching.[1]

Spatial Sensing	Direct Room.find() calls per creep.	Centralized Registry with $O(1)$ lookups.	Eliminates redundant spatial searches.[1]

The Bootstrapping Overlord: Foundational Architecture

The most successful architectural pattern identified for high-speed progression is the hierarchical decoupling of colony management into specific, modular layers. By structuring the AI into distinct tiers, developers ensure that high-level strategic goals do not interfere with the micro-optimizations required for individual unit performance.[1]

The Inversion of Control Hierarchy

In legacy systems, creeps execute independent, localized logic (e.g., a harvester scans the room, finds a source, and moves to it).[1] This "Search and Rescue" behavior leads to catastrophic redundant sensing.[1] If ten pioneer creeps execute room.find(FIND_SOURCES), the server processes ten identical spatial queries, resulting in $O(N \cdot M)$ complexity.

The Overlord architecture utilizes an Inversion of Control (IoC) pattern to mitigate this. The Inversion of Control model centralizes spatial queries at the Overlord level, passing direct intent commands to Zerg units to minimize redundant CPU usage, moving from the Overmind down through the Overseer, Directives, and finally the Overlord and its Zerg wrappers.

- **The Overmind:** Serves as the global root object, providing persistent storage, global heap caching, and a unified interface for cross-room analytics.[1] It hosts continuous "Analysts" (such as the Economy Analyst and Path Analyst) that poll the world state and convert raw API data into actionable Territory Intelligence.[1]
- **The Overseer:** Acts as the sensory brain of the colony, scanning for anomalous conditions or developmental milestones (such as an RCL increase) and instantiating Directives.[1]
- **The Directive:** Functional wrappers for room flags that define a strategic goal for a specific location.[1]
- **The Overlord:** Process managers that replace traditional "Role" scripts by generalizing a set of related tasks into a single biological process.[1] The Bootstrapping Overlord performs a single spatial scan of the room, registers the sources, and directly assigns task objects to its units.
- **The Zerg:** Extensions of the native Game.creeps object that are stripped of independent decision-making capabilities. They strictly execute the movement and intent commands provided by their governing Overlord.[1]

The Three-Phase Handshake and Orphan Adoption

A major vulnerability in automated spawning occurs due to the one-tick latency between a spawn intent and the actual instantiation of the creep object in memory. When spawnCreep() returns an OK code, the memory entry is generated immediately, but the creep does not appear in the Game.creeps object until the subsequent tick.[1] If a memory cleanup routine—designed to delete the memory of deceased creeps—executes during this window, it will erroneously delete the memory of the creep currently being spawned.[1] This creates an "Orphan Creep" that consumes resources but possesses no instructions.[1]

The Bootstrapping Overlord employs a "Three-Phase Handshake" to ensure deterministic lifecycle management and prevent this failure state [1]:

1. **Phase I: The Commitment (Tick N):** The Hatchery process identifies available energy and an idle spawn, calling spawnCreep(). Crucially, metadata, including the requesting Overlord's Process ID (PID) and the intended role, is written to the creep's memory configuration at the point of creation.[1]
2. **Phase II: The In-Utero Period (Tick $N+1$ to $N+M$):** The creep enters the physical spawning process (creep.spawning evaluates to true). The Hatchery registers the creep's name in a persistent "Pending Spawns" array in the Global Heap. The memory cleanup utility is strictly programmed to ignore any keys present in this array, averting accidental deletion.[1]
3. **Phase III: The Delivery (Tick $N+M+1$):** Once the spawning animation concludes and !creep.spawning evaluates to true, the Hatchery executes an adopt() method on the requesting Overlord, finalizing the transfer of the unit to the active workforce.[1]

In the event of a global reset or unexpected process crash, creeps may still become untethered. To resolve this, a Subreaper Process periodically scans the room for creeps whose memory.overlordPID does not match an active process in the Kernel table. The Subreaper algorithmically reassigns these orphans to the appropriate Overlord based on their body composition (e.g., assigning a heavy WORK creep to the Upgrade Overlord), ensuring that no energy investment is left idle.[1]

RCL 1: Pioneer Optimization and Controller Saturation

The primary objective at RCL 1 is to accumulate the 200 energy required to upgrade the controller to RCL 2, thereby unlocking 5 Extensions and allowing for greater morphological flexibility.[3] Because a brand new room begins with a maximum energy capacity of 300 (provided entirely by the lone Spawn), the engineering challenge lies in finding the optimal body composition and total count of "pioneer" creeps to achieve theoretical maximum extraction without stalling the economy.[3]

The Mathematics of Pioneer Harvesting

A standard energy source in an unreserved, neutral room regenerates 1,500 energy units every 300 ticks, translating to an average maximum yield of 5 energy units per tick.[4] However, the initial room is owned, meaning the source generates 3,000 energy per 300 ticks, equating to an exact output of 10 energy units per tick.[5]

The Screeps engine dictates that each WORK part harvests 2 energy units per tick when calling the harvest() function.[6] Therefore, a total of 5 WORK parts are required in the room to fully saturate the extraction rate of a single source.[7] Because the spawn capacity at RCL 1 is hard-capped at 300 energy, a single optimal 5-WORK creep (which would cost 550 energy) is impossible to spawn.[6]

Pioneer creeps must therefore be carefully balanced to maximize utility within the 300-energy budget. The engine defines the cost of parts as follows: WORK costs 100 energy, CARRY costs 50 energy, and MOVE costs 50 energy.[1]

- A unit with 1 WORK, 1 CARRY, and 1 MOVE (MWC) costs 200 energy.[6]
- A unit with 2 WORK, 2 CARRY, and 1 MOVE (MMWCC) would cost 350 energy, exceeding the limit.[6]
- A heavily optimized pioneer body configuration that strictly adheres to the 300-energy cap is ``, costing exactly 250 energy.[6]

To determine the true mathematical ROI of this 250-energy pioneer, an analysis of its operational cycle is required. Assuming a generalized travel distance, if the creep takes 5 ticks to travel to the source, spends 25 ticks harvesting (extracting 100 energy with its 2 WORK parts at a rate of 4 energy per tick), and 5 ticks returning to the Spawn or Controller, the total cycle consumes 35 ticks.[6] Over its standard ,1500-tick lifespan, it completes approximately 42 of these cycles. The gross total energy harvested is 2,100 units (42 cycles $\times$ 50 carry capacity).[6] Subtracting the initial 250-energy spawn cost yields a net positive return of 1,850 energy per pioneer.[6]

Controller Saturation and the Spawn Bottleneck

While extracting 10 energy per tick is the localized goal, the pioneer must also facilitate the upgrading of the room's Controller. The upgradeController() action costs 1 energy per tick per WORK part.[10] Thus, a pioneer with 2 WORK parts consumes energy at a rate of 2 units per tick—which perfectly balances the 4 units per tick it can harvest, creating a synchronized flow of throughput.[6]

A mathematical derivation determines the absolute optimal number of pioneers required to saturate the Controller without wasting spawn energy. If an optimized pioneer spends approximately 77% of its cycle actively performing productive tasks (harvesting and upgrading) and 23% of its time moving, its effective throughput is multiplied by 0.77.[6] To continuously

extract the 10 energy per tick available from the source, the colony requires:

$$\text{Optimal Count} = \frac{5 \text{ Required WORK parts}}{0.77 \text{ Time Efficiency}} = 6.49 \text{ creeps}$$

Therefore, exactly 6.5 (effectively rounded to 6 or 7 depending on distance) pioneer creeps are required to perfectly saturate an RCL 1 controller without wasting spawn time or energy.[6] Spawning 8 or 9 pioneers results in creeps idling at the source, wasting their Time to Live (TTL) and the energy used to construct them.[6]

However, achieving this theoretical maximum introduces a critical physiological bottleneck: the Spawn itself regenerates only 1 energy unit per tick naturally.[1] If the colony attempts to spawn 6 pioneers sequentially without the first pioneers returning energy to the spawn, the Hatchery enters an immediate deadlock. The Bootstrapping Overlord prevents this "Metabolic Collapse" by injecting a strict state override: the first two pioneers spawned must prioritize the transfer() action to the Spawn over the upgradeController() action until the localized energy economy stabilizes.[1] Only once the Spawn's internal buffer is consistently above 250 energy does the Overlord permit the remaining workforce to focus entirely on the Controller.

RCL 2: The Mining Paradigm Shift and Economic ROI

Upon accumulating 200 energy in the Controller, the room advances to RCL 2. This milestone unlocks 5 Extensions, expanding the room's maximum spawn capacity from 300 to 550 energy (300 from the Spawn + 50 from each of the 5 Extensions).[3] This is the most crucial morphological inflection point in the early game, as 550 energy is precisely the cost required to spawn a dedicated miner with 5 WORK parts and 1 MOVE part.[9]

At this stage, the "Universal Pioneer" model becomes a severe economic liability due to the inefficiency of moving WORK parts back and forth.[7] The Bootstrapping Overlord must pivot to a decoupled "Miner and Hauler" paradigm.[4] A static miner sits directly adjacent to the source, continuously harvesting, while specialized haulers (composed entirely of CARRY and MOVE parts) ferry the resources.[4] This shift introduces a profound architectural choice for the speedrunner: should the miner drop energy directly onto the ground (Drop Mining), or should the bot invest 250 energy into constructing a StructureContainer beneath the miner (Container Mining)?

The Thermodynamics of Drop Mining Decay

Drop mining is the simplest methodology to program: a creep moves to the source and continuously calls harvest(), allowing the excess energy to fall onto the terrain tile. However, the Screeps engine simulates thermodynamic decay for dropped resources. Energy left on the ground decays at a rate of $\lceil \frac{\text{amount}}{1000} \rceil$ per tick.[5]

For a source producing 10 energy units per tick (the maximum yield of an owned room), the accumulation immediately crosses the threshold where at least 1 energy unit decays every single tick.[5] Over a 300-tick regeneration cycle, the source yields a gross of 3,000 energy, but the decay effectively acts as a persistent 10% tax.[5] This reduces the net yield of the source to a theoretical maximum of 9 energy units per tick, assuming haulers pick it up instantly.[5]

The Maintenance Economics of Container Mining

Container mining mitigates this thermodynamic decay entirely. A container holds up to 2,000 energy units, and any resources dropped by a creep standing on the same tile are automatically deposited inside the structure.[13] The fundamental advantage is that energy stored within a container does not undergo resource decay.[7]

However, the container structure itself is subject to physical decay, losing 5,000 hit points every 100 ticks.[5] To maintain the container's integrity, a creep must periodically use the repair() action, which restores 100 hit points per 1 energy unit consumed.[6] The baseline maintenance cost calculation is determined by the formula:

$$\text{Upkeep Cost} = \frac{5000 \text{ hits}}{100 \text{ ticks}} \times \frac{1 \text{ energy}}{100 \text{ hits}} = 0.5 \text{ energy per tick}$$

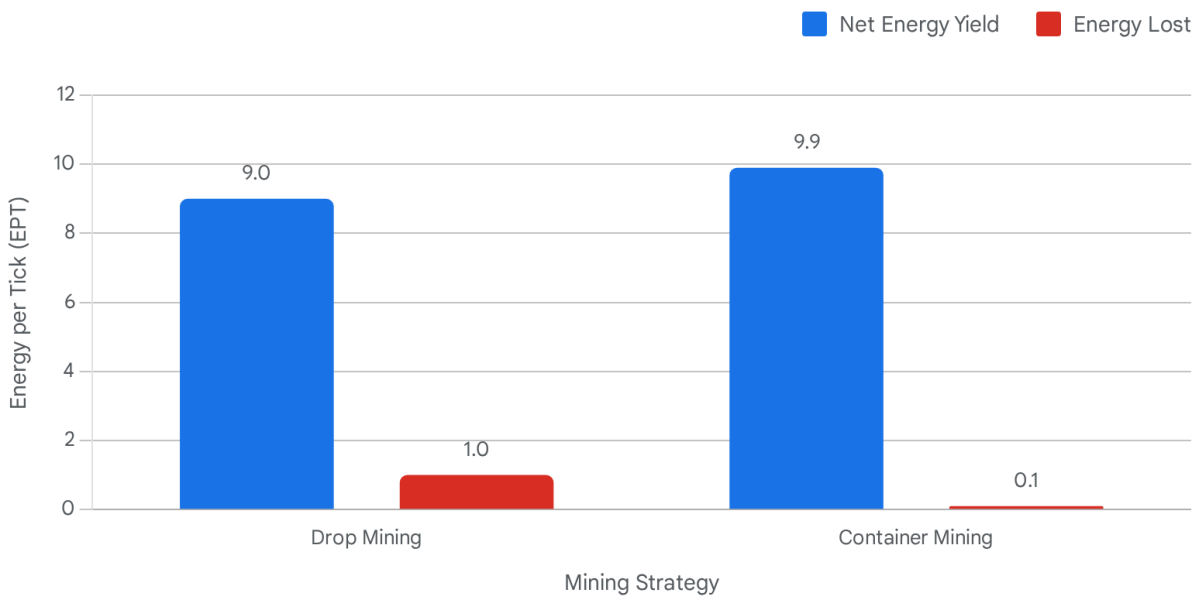
This 0.5 energy per tick represents the upkeep cost for containers located in neutral or remote rooms.[5] Crucially, in an owned room (which is the case for RCL 2 progression), the structural decay rate is significantly lower. The math adjusts such that the maintenance overhead drops to a mere 0.1 energy per tick.[5]

Mathematical ROI Comparison

Comparing the two methodologies mathematically reveals the strict superiority of Container Mining at RCL 2:

- **Drop Mining:** 10 (Gross Yield) - 1.0 (Decay Tax) = **9.0 Net Energy / Tick**.[5]
- **Container Mining:** 10 (Gross Yield) - 0.1 (Maintenance Tax) = **9.9 Net Energy / Tick**.[5]

Return on Investment: Container vs. Drop Mining at RCL 2



In an owned room, container mining reduces total energy loss from a 1.0 EPT decay tax to a 0.1 EPT maintenance cost, increasing overall net yield by 10%.

Data sources: Reddit (r/screeps)

By eliminating 90% of the energy loss, Container Mining readily justifies the initial 250 energy investment required to construct the structure.[13] A miner equipped with 6 WORK parts (instead of 5) can utilize its spare cycle to execute the repair() command directly on the container it stands on, eliminating the need for a dedicated maintenance creep.[1]

Furthermore, containers provide a massive, often overlooked "administrative overhead" reduction.[5] In a Heap-first architecture, the Overlord can cache the ID of the container in the global memory upon initialization. This allows haulers to target the container directly via Game.getObjectById(), bypassing the extremely expensive $O(N)$ CPU cost of executing room.find(FIND_DROPPED_RESOURCES) on every single tick.[5] In speedrunning scenarios, saving this CPU overhead prevents early-game timeouts and allows the bot to scale its operations seamlessly.

RCL 3: Logistics Brokerage and The Gale-Shapley Algorithm

At 45,000 accumulated energy, the room advances to RCL 3. This unlocks an additional 5 Extensions (bringing the total to 10, or 800 spawn capacity) and, most importantly, the first StructureTower.[3] With containers established at the mining sites, the Bootstrapping Overlord must flawlessly manage the flow of energy from these providers to the various consumers (Spawns, Extensions, the Controller, and now the Tower).

The introduction of the Tower adds an unpredictable, high-priority consumer to the energy grid.[3] Legacy systems rely on role-based pulling—where haulers independently execute scans for energy deficits—which inevitably leads to the "Energy Racing" phenomenon.[1] Multiple haulers may observe the same dropped pile or empty extension and simultaneously path toward it. Because only the first to arrive can complete the transfer, the subsequent haulers waste dozens of ticks traversing the room fruitlessly, creating massive logistical inefficiencies.[1]

To counter this, a top-tier Bootstrapping Overlord implements a Global Logistics Broker utilizing the Gale-Shapley Stable Matching Algorithm (often referred to as the Stable Marriage problem).[1]

Stable Matching in Screeps

The logistics economy is algorithmically modeled as a bipartite matching problem between "Providers" (Containers, Miners) and "Requesters" (Spawns, Extensions, Towers, Upgraders).[1]

1. **Proposal Generation:** Each free hauler (the proposer) generates a heuristic preference list. This list ranks requests based on the urgency of the requester (e.g., active Towers under attack receive a weight of 10, empty Spawns receive 5, and Upgraders receive 1) multiplied by the inverse of the pathfinding distance.[1]
2. **Tentative Matching:** Haulers "propose" to their most preferred, available target.
3. **Conflict Resolution:** The requester maintains its current best proposal. If a closer or higher-capacity hauler proposes, the requester "rejects" the inferior proposal, forcing the rejected hauler to immediately propose to its next preference.[1]

This algorithm guarantees stability: no two haulers will ever race for the same energy, and critical infrastructure, such as a depleted Tower during an invasion, is refilled with absolute optimal efficiency.[1] By centralizing this logic, the CPU burden is shifted from N individual haulers to a single, highly optimized $O(N^2)$ operation per tick.

The Effective Store Ledger (S_{eff})

To prevent overlapping assignments over the temporal span of multiple ticks, the Logistics Broker implements a predictive "Effective Store" (S_{eff}) ledger.[1] Instead of merely querying the native structure.store property, the Overlord calculates future states:

$$S_{eff} = \text{Current Store} + \text{Incoming Reservations} - \text{Outgoing Reservations}$$

When the broker matches a hauler to a requester, it immediately adds a reservation to the ledger.[1] If a Tower requests 200 energy and a hauler is dispatched carrying 200 energy, the broker immediately updates the Tower's S_{eff} to full.[1] Subsequent iterations of the matching algorithm on following ticks will observe that the Tower's needs are theoretically met and will ignore the structure, even if the hauler is still 10 ticks away from executing the physical transfer.[1] This perfect predictive routing is the hallmark of a speedrunning logistics engine.

Wavefunction Collapse and Traffic Shoving

As the volume of logistics traffic increases, creeps frequently encounter bottlenecks in narrow corridors or around the Spawn. Standard moveTo() logic causes creeps to stop and recalculate their path, generating massive CPU spikes.[1] Top-tier architectures circumvent this using a variant of the "Wavefunction Collapse" algorithm for traffic management.[1]

Every tick, the engine collects move intents from all creeps.[1] The resolution engine sorts these intents by priority tiers. Military units and heavily loaded haulers receive top priority.[1] If a high-priority hauler's intended tile is occupied by a lower-priority creep (such as an idle builder or a static upgrader), the system "shoves" the lower-priority creep into an adjacent valid tile.[1] This ensures that the high-priority logistics supply chain is never physically interrupted by civilian traffic, maintaining optimal velocity.[1]

RCL 4: Spatial Geometry and Extension Placement

Achieving RCL 4 (135,000 energy) unlocks the Storage structure, which completely transforms the economy by providing a 1,000,000-unit buffer, permanently decoupling production from consumption.[3] However, the speedrun window *before* Storage is built requires meticulous management of the 20 available Extensions.[3] The primary bottleneck during RCL 3 and 4 is the physical travel time required for "filler" creeps to walk between the energy providers and the scattered Extensions.

Distance Transform and Floodfill Automation

Top-tier bots completely automate base layout using advanced computational geometry rather than relying on hardcoded coordinates.[1] The Architect process runs a Distance Transform (DT) algorithm to map the room's topography.[16] This algorithm assigns an integer value to every walkable tile representing its Manhattan distance to the nearest wall.[16]

By filtering the matrix for values ≥ 3 or ≥ 5, the Architect guarantees a sufficiently large, unobstructed 5×5 or 13×13 footprint for the core base infrastructure (the Bunker

Stamp).[16] This prevents the bot from attempting to build core structures in fragmented corridors.

Following the Distance Transform, a Floodfill algorithm expands outward from the designated core (the eventual Storage and Terminal position) to map the logistical costs of all surrounding tiles.[16] Extensions are subsequently placed strictly on tiles with the lowest floodfill scores.[16] This algorithmic placement inherently minimizes the distance a filler creep must travel from the energy hub to the furthest extension.[16]

Minimizing Filler Travel Time: The EDA Metric

To truly minimize filler travel time, the spatial layout must optimize the Energy Deposited Average (EDA).[18] The EDA is calculated as:

$$\text{EDA} = \frac{\text{Total Energy Deposited}}{\text{Number of Deposit Ticks} + \text{Number of Move Ticks}}$$

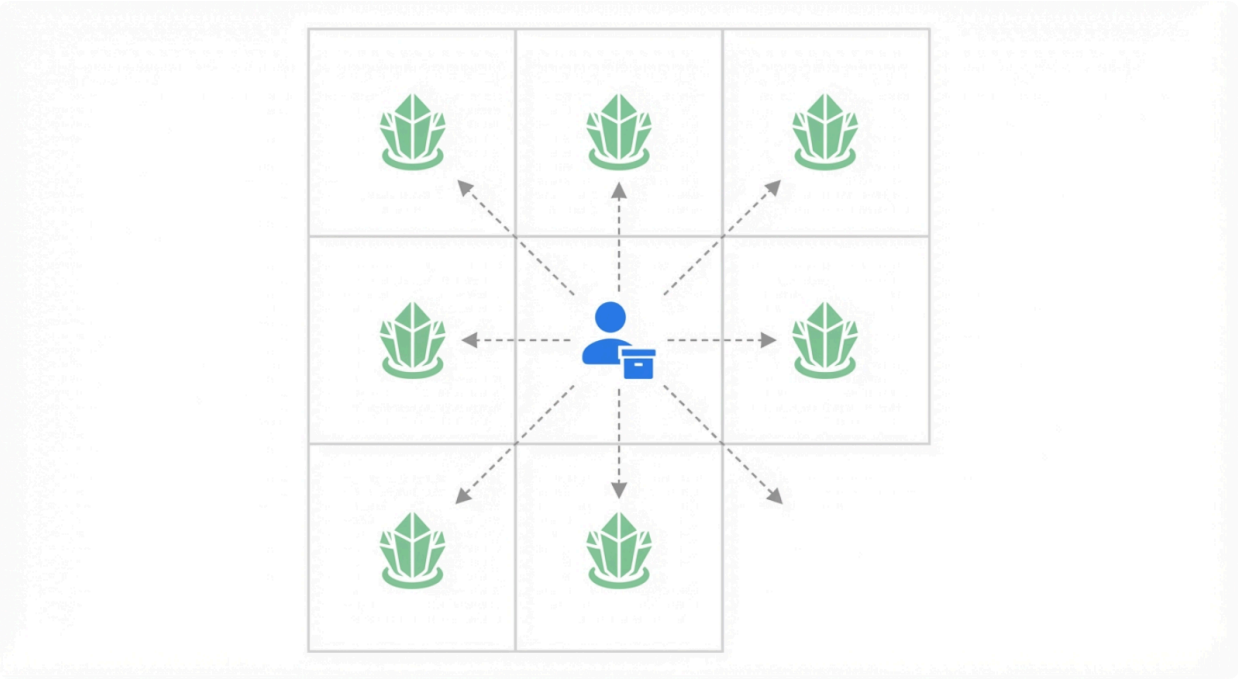
A naive, linear layout of extensions forces the filler creep to walk down a line, resulting in a low EDA and massive CPU waste from constant moveTo() intents.[18] The optimal theoretical placement strategy is the "Fast Filler" or "Flower Layout" (often termed the Dissi flower by the community).[14]

In a Fast Filler stamp, a single standing tile is surrounded by up to 8 interactive structures (Extensions, and eventually a Link or Container).[18] In practical early-game application (prior to unlocking Links at RCL 5), a filler creep enters a central tile surrounded by 7 Extensions.[3] The creep requires 2 ticks to navigate into the center.[18] Once positioned, it can execute the transfer() command to all 7 Extensions sequentially over 7 ticks without issuing a single move intent.[18]

Assuming RCL 3/4 extension capacity (50 energy per extension), depositing 350 energy takes exactly 7 ticks.[18]

$$\text{EDA} = \frac{350 \text{ energy}}{7 \text{ deposit ticks} + 2 \text{ move ticks}} = 38.8 \text{ Energy/Tick}$$

Optimized Fast Filler Spatial Layout for Extension Clusters



By positioning a filler creep in a central tile, it can deposit energy into up to 7 surrounding extensions sequentially, eliminating movement latency and achieving an EDA of 38.8.

This layout drastically reduces the time required to empty a hauler, allowing the Hatchery to maintain near 100% uptime on spawns without succumbing to energy starvation.[18] Furthermore, by pre-calculating this "Bunker Stamp" layout, the bot completely bypasses the need for the expensive findClosestByPath() function during the refilling cycle, further dropping the CPU baseline.[19]

Actionable Code Patterns for the Bootstrapping Overlord

The transition from theoretical optimization to executable logic requires rigid, deadlock-resistant code patterns. A colony in the RCL 1 to 4 phase is highly susceptible to the "Death Spiral"—a catastrophic feedback loop where energy consumption by upgraders exceeds energy production, leaving the room unable to spawn replacement miners or haulers.[1]

Death Spiral Prevention: The Upgrader Trigger ($U_{trigger}$)

The most common failure in a speedrun is the over-production of upgraders, which rapidly

drains the spawn network. To prevent this, the Bootstrapping Overlord implements a strict mathematical gate known as the Upgrader Trigger ($U_{trigger}$).[1] The system continuously evaluates the Effective Store (S_{eff}) of the room against a Critical Energy Buffer (T_{crit}).[1]

The T_{crit} is mathematically defined as the exact energy cost required to instantly replace the entire mining and hauling fleet, plus an overhead buffer for emergency road repair or defensive operations.[1] The state-machine logic dictates that a dedicated Upgrader is only added to the spawn queue if the following condition is met:

$$S_{eff} > T_{crit} + (\text{Cost of Upgrader})$$

By enforcing this boolean check at the beginning of every single tick, the Overlord guarantees that the metabolic core of the colony (the harvesting and hauling fleets) is never starved of spawn priority.[1] If energy drops below the threshold, upgraders are allowed to die naturally, preserving the core economy until the buffer is restored.

The State Class Implementation

To manage the complexity of pioneer units that must seamlessly transition between harvesting, building, and upgrading depending on the tick, the Bootstrapping Overlord abandons giant procedural switch statements in favor of the State Class pattern.[1]

Each discrete behavior is encapsulated in a class object containing a run() and a transition() method.[1]

1. **Run:** Executes the exact API intent (e.g., creep.harvest(source) or creep.build(site)).
2. **Transition:** Evaluates environmental conditions and returns the string identifier of the next state.[1]

For example, a filler creep evaluating its state will check creep.store.getUsedCapacity(). If the capacity reaches 0, the transition() method returns 'WITHDRAW', shifting the creep's behavior toward the nearest container. If the capacity reaches its maximum, it returns 'DEPOSIT', directing the creep toward the extensions. This explicit transition mechanism prevents edge-case logic bugs where creeps oscillate between targets, freeze due to conflicting intents, or attempt to execute commands they lack the resources to complete.[1]

Intent Combination and Action Resolvers

A top-tier execution engine maximizes per-tick value by combining non-exclusive actions.[1] The game API permits certain actions to occur simultaneously within the same tick provided they utilize different internal pipelines.[1] For example, a single pioneer creep can execute move() and

build() concurrently, allowing the bot to construct roads while traversing them.[21]

The Overlord utilizes an "Action Resolver" module that parses the intent queue before finalizing the tick.[1] If a hauling creep is moving across a damaged road tile and possesses a spare WORK part, the Resolver automatically injects a repair() intent into the execution batch.[21] This "Repair-on-Transit" pattern ensures infrastructure health is maintained at zero additional CPU or pathing cost.[21] This level of concurrent execution is vital during the high-throughput hauling phases of RCL 3 and 4, ensuring that the logistics network does not degrade while the core processors are focused on maximizing controller throughput.

Pre-Spawning and Duty Cycle Optimization

To achieve true tick-perfect progression, the Bootstrapping Overlord must eliminate the "latency" of creep death. If the system waits for a miner to die before spawning a replacement, the source will idle for the duration of the new miner's spawn time and travel time. To counter this, the Overlord monitors the Time to Live (TTL) property of all active creeps.

The system calculates the required pre-spawn window based on the physical distance to the operational site and the number of body parts (as each part takes 3 ticks to spawn). If a miner's TTL is less than or equal to SpawnTime + TravelTime, the replacement is immediately pushed to the top of the Hatchery queue.[1] This ensures that the replacement creep arrives at the source on the exact tick the predecessor expires, maintaining a 100% duty cycle on resource extraction and controller upgrading.

Conclusion: The Engineering of Autonomous Sovereignty

The pursuit of speedrunning the early game in Screeps demands a fundamental departure from reactive, script-based logic toward proactive, systems-level engineering. The transition from RCL 1 to RCL 4 is not merely a matter of waiting for energy to accumulate; it is a rigorous exercise in computational thermodynamics, CPU cycle management, and spatial optimization.

By adopting the Bootstrapping Overlord architecture, developers centralize decision-making, drastically reducing the CPU overhead caused by redundant sensing and pathfinding. At RCL 1, the precise calculation of 6.5 pioneer creeps ensures maximum controller saturation without triggering a spawn deadlock. As the colony breaches RCL 2, the implementation of container mining provides a mathematically proven 10% increase in net energy yield by eliminating resource decay.

Furthermore, the integration of the Gale-Shapley algorithm for logistics routing and the Distance Transform for "Fast Filler" extension placement guarantees that every unit of energy harvested is routed to the controller with absolute efficiency. Through the rigorous application

of $U_{trigger}$ thresholds and explicit State Class patterns, the colony is immunized against the economic death spirals that plague early-game development. Ultimately, treating the codebase not as a collection of game scripts, but as an industrial-grade operating system, is the definitive path to achieving tick-perfect sovereignty in the Screeps environment.

Works cited

1. 2026-02-18 - Screeps Research Combined.pdf
2. Understanding game loop, time and ticks - Screeps Documentation, accessed February 19, 2026, https://docs.screeps.com/game-loop.html
3. Control | Screeps Documentation, accessed February 19, 2026, https://docs.screeps.com/control.html
4. Remote Harvesting - Screeps Wiki, accessed February 19, 2026, https://wiki.screepspl.us/Remote_Harvesting/
5. How important are containers? : r/screeps - Reddit, accessed February 19, 2026, https://www.reddit.com/r/screeps/comments/5l5wd2/how_important_are_containers/
6. RC1.0 Design: Basic Harvesting | Screeping - WordPress.com, accessed February 19, 2026, https://screeping.wordpress.com/2021/05/01/episode-4-1-rc1-basic-harvesting/
7. Remote mining article by slowmotionghost · Pull Request #60 · screeps/docs - GitHub, accessed February 19, 2026, https://github.com/screeps/docs/pull/60/files
8. Screeps #1: The Game Plan | Field Journal, accessed February 19, 2026, https://jonwinsley.com/notes/screeps-game-plan
9. Spawned size max of creeps dependent on room level? : r/screeps - Reddit, accessed February 19, 2026, https://www.reddit.com/r/screeps/comments/4xbzo0/spawned_size_max_of_creeps_dependent_on_room_level/
10. Questions on roles | Screeps Forum, accessed February 19, 2026, https://screeps.com/forum/topic/913/questions-on-roles
11. What's the best strategy for gathering energy? : r/screeps - Reddit, accessed February 19, 2026, https://www.reddit.com/r/screeps/comments/4x8z65/whats_the_best_strategy_for_gathering_energy/
12. Energy - Screeps Wiki, accessed February 19, 2026, https://wiki.screepspl.us/Energy/
13. Static Harvesting - Screeps Wiki, accessed February 19, 2026, https://wiki.screepspl.us/Static_Harvesting/
14. How do you manage extensions? : r/screeps - Reddit, accessed February 19, 2026, https://www.reddit.com/r/screeps/comments/5vtnnj/how_do_you_manage_extensions/
15. CPU Optimization | Screeps Forum, accessed February 19, 2026, https://screeps.com/forum/topic/1614/cpu-optimization
16. Automating Base Planning in Screeps – A Step-by-Step Guide, accessed February

19, 2026, https://sy-harabi.github.io/Automating-base-planning-in-screeps/
17. Automated Base Planning Guide & Resources - screeps - Reddit, accessed February 19, 2026, https://www.reddit.com/r/screeps/comments/1o2r56j/automated_base_planning_guide_resources/
18. Compact Extension Arrays | Screeps Forum, accessed February 19, 2026, https://screeps.com/forum/topic/136/compact-extension-arrays
19. How can i store all my extensions/spawns in order? | Screeps Forum, accessed February 19, 2026, https://screeps.com/forum/topic/2426/how-can-i-store-all-my-extensions-spawns-in-order
20. Automatic Base Building - Screeps Wiki, accessed February 19, 2026, https://wiki.screepspl.us/Automatic_base_building/
21. Faster Controller Levelling : r/screeps - Reddit, accessed February 19, 2026, https://www.reddit.com/r/screeps/comments/5jia18/faster_controller_levelling/