Architectural Design and Engineering Protocols for High-Tier Military State-Machines in Autonomous Screeps Environments

The evolution of autonomous systems in the Screeps environment has moved beyond simple script-based reactions toward comprehensive military architectures that function as decentralized, yet highly coordinated, strategic entities. A top-tier military state-machine is defined not by the complexity of its individual unit code, but by the robustness of its hierarchical decision-making framework and its ability to mathematically quantify every tactical engagement. The paradigm shift required for modernization involves transitioning from legacy target-lock logic to an event-driven, hierarchical state-machine architecture that decouples strategic intent from tactical execution.

In the high-stakes ecosystem of the global control level (GCL) rankings, the margin between a successful siege and a total colony collapse often rests on a single tick of miscalculated healing throughput or a fatigue-induced break in squad cohesion. This report provides an exhaustive engineering breakdown of the protocols, mathematical models, and implementation steps necessary to develop and maintain a military system capable of competing at the highest tiers of autonomous warfare.

Foundational Command Hierarchy: The Overlord Framework

The most successful architectural pattern identified in the research is the hierarchical decoupling of colony management into specific layers: the Overmind, the Overseer, the Directive, and the Overlord. This structure ensures that high-level strategic goals do not interfere with the micro-optimizations required for individual unit performance.

The Overmind and Strategic Analysis

The Overmind serves as the global root object, providing persistent storage, global heap caching, and a unified interface for cross-room analytics. Within this layer, "Analysts" perform continuous polling of the world state, converting raw game data into actionable metrics. This sensing layer is critical because military state-machines cannot function effectively on raw sensory data alone; they require processed "Territory Intelligence".

Component	Responsibility	Performance Impact
Threat Analyst	Calculation of hostile DPT and heal capacity	High CPU, highly cacheable
Path Analyst	Multi-room routing and CostMatrix generation	Medium CPU, low cacheability
Economy Analyst	Monitoring tower energy and	Low CPU, medium cacheability

Component	Responsibility	Performance Impact
	lab throughput	
Strategic Selector	Priority-based task allocation across colonies	Medium CPU, low cacheability

The Overseer acts as the sensory brain of the colony, scanning for anomalous conditions such as an invasion, a structural breach, or a resource opportunity. Upon detection, the Overseer does not issue unit commands; instead, it instantiates Directives.

Directives as Goal Wrappers

Directives are functional wrappers for room flags that define a strategic goal for a specific location. By using flags as the basis for directives, the system achieves human-assisted automation: a developer can manually place a flag to trigger a siege, or the Overseer can automate this placement based on room scoring.
Directives are categorized by color codes, with "Red" typically reserved for military operations. A Red Directi[span_10](start_span)[span_10](end_span)ve on a hostile room acts as an attachment point for a Combat Overlord. This separation allows the system to remain modular; the logic for *deciding* to attack is entirely separate from the logic for *executing* the attack.

Overlords as Process Managers

The Overlord class replaces traditional "Role" systems by generalizing a set of related tasks into a single biological process. A Siege Overl[span_11](start_span)[span_11](end_span)ord handles the following lifecycle:

1. **Spawning:** Calculating the optimal body composition based on the target room's tower damage.
2. **Boosting:** Coordinating lab intents to apply T3 mineral compounds to the necessary parts.
3. **Logistics:** Ensuring the military unit reaches the target through safe highway routes.
4. **Tactics:** Driving the combat state-machine (Harass, Drain, Siege) once the creeps arrive.

Quantitative Defensive Logic: The Mathematical Perimeter

High-tier defense is a game of energy efficiency and burst-damage prevention. The military state-machine must transition between "Peacetime Maintenance," "Yellow Alert" (NPC or weak scouts), and "Red Alert" (Full-scale siege).

Tower Engagement Math and Energy Management

Towers are the primary tool for active defense, but they are economically expensive if used improperly. A tower's damage output decreases linearly as the distance to the target increases. The exact formula used by the Screeps engine is:
where Range is clamped between 5 and 20.

Range (r)	Condition	Attack (Hits)	Heal (Hits)	Repair (Hits)
r \le 5	Optimal Range	600	400	800

Range (r)	Condition	Attack (Hits)	Heal (Hits)	Repair (Hits)
5 < r < 20	Linear Falloff	600 - 30(r-5)	400 - 20(r-5)	800 - 40(r-5)
r \ge 20	Max Falloff	150	100	200

A high-tier defensive state-machine calculates the "Survivability Threshold" of a hostile creep before firing. If a hostile unit's heal-per-tick (HPT) exceeds the combined DPT of all towers at their current ranges, the state-machine should transition to a "Hold Fire" state to prevent energy draining. This is particularly critical because tower fire consumes 10 energy units regardless of effectiveness, making "Tower Draining" a viable strategy for attackers.

Passive Barrier Fortification and Layout

Structural defense relies on walls and ramparts. High-tier bots utilize the "Min-Cut" algorithm to determine the minimum number of ramparts needed to seal a base footprint. A common engineering error is placing ramparts too close to internal structures. Best practices dictate a 3-tile buffer to prevent rangedMassAttack from damaging internal buildings through the rampart.

RCL	Wall Limit (Hits)	Max Towers	Strategic Shift
3	N/A	1	Transition from spawn-defense to initial walling
5	Controller Level dependent	2	Introduction of Terminal energy sharing
7	Controller Level dependent	3	Introduction of Lab-boosted sorties
8	300,000,000	6	Absolute energy throughput and Nuke defense

During a Red Alert state, the machine prioritizes "Repair Spam" over "Attack" if tower fire cannot secure a kill. A room can sustain approximately 60,000 hits of repair per tick if the economy is sufficiently robust, making well-defended RCL 8 rooms nearly impossible to crack with unboosted forces.

Automated Safe Mode Trigger Protocols

Safe Mode is a non-renewable resource (one activation per level) and must be guarded by sophisticated trigger criteria. The "Fail-Safe" state should only be entered under the following conditions:

1. **Critical Breach:** A rampart protecting a core structure (Storage, Spawn, Terminal) has fallen below 10,000 hits and a hostile creep with WORK or ATTACK parts is adjacent.
2. **Pathfinding Threat:** The ThreatAnalyst finds a valid path from a hostile creep to the Spawn that does not pass through a rampart.
3. **Worker Attrition:** Hostiles have destroyed more than 50% of the colony's worker creeps, and energy sharing via Terminals is offline.
4. **Adjacency Check:** A hostile creep with WORK parts is adjacent to the room's Controller.

Offensive State-Machines: The Mechanics of Seizure

Offense in Screeps is significantly more resource-intensive than defense, requiring an estimated

3:1 economic advantage for success. A high-tier offensive state-machine operates through a series of tactical transitions: Scouting, Harassment, Draining, and Siege.

Scouting and Strategic Selection

Before an attack is launched, the Overseer places a Purple Scout Directive. The scout creep's goal is to record the following metadata in the global memory:

- **Wall Thickness:** The average and minimum hits of the enemy ramparts.
- **Tower Energy:** Whether the defender has an active tower-refilling script.
- **Boost Availability:** Presence of Labs and whether they are stocked with XGHO2 (Tough) or XLHO2 (Heal).

This information determines the state transition: if towers are unrefilled, transition to Tower Draining; if walls are thin, transition to Siege; if the opponent is high-tier, transition to Room Quarantining.

The "Drainer" State-Machine

Tower draining is the most cost-effective offensive strategy. The state-machine for a drainer unit follows a specific oscillating logic:

1. **Approach:** Move to range 1 of the room exit.
2. **Step In:** Move one tile into the hostile room, ideally at range \ge 20 from all towers.
3. **Soak:** Wait for the end-of-tick tower processing to apply damage.
4. **Step Out:** On the next tick, move back into the safe room.
5. **Recover:** Receive healing from a stationary healer in the safe room until hits are at 100%.

The goal is to force the defender to spend 10 energy per shot while the attacker's healing costs only the time (ticks) of the unit. Against bots with poor terminal logic, this will eventually trigger an economic collapse.

The Siege and Destruction FSM

A siege is initiated once the defender's energy is low or the attacker has achieved Tier 3 boosting capability. The machine spawns "Breakers" (boosted work creeps) and "Medics".

Role	Core Parts	Primary Boost	Tactical Goal
Breaker	TOUGH, WORK, MOVE	XZHO2, XGHO2, XH2O	Dismantling ramparts and spawns
Medic	TOUGH, HEAL, MOVE	XZHO2, XGHO2, XLHO2	Synchronized healing of the Breaker
Destroyer	TOUGH, ATTACK, MOVE	XZHO2, XGHO2, XUH2O	Elimination of hostile sortie creeps

The state-machine transitions from TRAVEL to SIEGE_TARGET once the breaker is adjacent to a wall. It is a best practice to use dismantle() over attack() for structures, as WORK parts provide 50 damage per part compared to the 30 damage of ATTACK parts.

Unit Coordination: The Quad Movement Engine

In the high-tier meta, single creeps are easily countered by random focus fire or specialized melee defenders. To overcome this, top-tier bots implement "Quads"—synchronized groups of

four creeps moving as a 2x2 unit.

Synchronized Movement and Traffic Resolution

A quad must move in lockstep. If any member lags due to fatigue, the formation breaks and the group's collective heal power is halved. Implementation of a quad movement engine requires a custom pathing solution:

1. **Leader Assignment:** One creep is designated as the "Master," usually the top-left unit of the quad.
2. **Custom CostMatrix:** The pathfinder considers any tile that would place *any part* of the 2x2 footprint in a wall to be impassable. This typically means marking every tile adjacent to a wall (top and left) as cost 255.
3. **Fatigue Synchronization:** The squad's move intent is only issued if *all* members have zero fatigue. If one member enters a swamp, the entire squad waits for that unit to recover.

Wavefunction Collapse for Shoving

Traffic jams are the primary cause of quad failure inside a base. To resolve this, high-tier bots use a variant of the "Wavefunction Collapse" algorithm for traffic management.

- **Intent Collection:** Every creep in the room submits an array of valid move targets (with their current position as the lowest priority).
- **Priority Tiers:** Military quads are given top-tier priority, followed by defenders, haulers, and then static harvesters.
- **Resolution:** The engine iterates through the highest-priority intents. If a quad's target tile is occupied by a lower-priority creep, the system "shoves" the worker into an adjacent valid tile, effectively clearing a path for the military formation.

Mathematics of Chemical Warfare: The Tier 3 Advantage

The delta between an unboosted creep and a Tier 3 boosted creep is an order of magnitude, not a marginal increase. A military state-machine must integrate lab logic as a prerequisite for any offensive state transition.

Stacking Boosts and Effective Hit Points

Tier 3 TOUGH boosts (XGHO2) reduce incoming damage by 70%. This effectively multiplies the hit points of that body part by 3.33.
For a Tier 3 boosted part:
This stacking effect is the only way a creep can survive the focus fire of six RCL 8 towers (6 \times 600 = 3,600 \text{ DPT}). Without boosts, even a 50-part creep would be destroyed in 2 ticks. With boosts, the effective HP of 13 TOUGH parts is 4,330, which allows a single medic to fully recover the damage of those towers in the same tick if they have 27 Tier-3 HEAL parts (27 \times 48 = 1,296 \text{ HPT} adjusted for tough parts).

Part Type	Unboosted Effect	T3 Boosted Effect	Multiplier
ATTACK	30 dmg	120 dmg	4.0x
RANGED_ATTACK	10 dmg	40 dmg	4.0x
HEAL	12 hits	48 hits	4.0x
WORK (Dismantle)	50 dmg	200 dmg	4.0x
MOVE	-2 fatigue/tick	-8 fatigue/tick	4.0x

The "Pre-healing" and Synchronization Logic

In the Screeps engine, intents are processed in batches. A major engineering challenge is the timing of heal() versus incoming attack(). If damage is applied first and reduces a creep to 0 hits, any subsequent heal intent on that creep will fail, as the unit is already dead.
Top-tier state-machines implement "Pre-healing": healers call heal() on their squad members every tick *regardless* of their current health. This ensures that if the creep takes damage during the tick resolution, it immediately begins recovery. In a quad, each member should prioritize healing the squad member with the lowest *projected* health based on tower focus fire predictions.

System Implementation: Coding a Modular Military FSM

The implementation of a top-tier state-machine requires a move away from "Role" modules toward "State Classes" that implement a standard interface. This allows the bot to decouple the *behavior* of the unit from the *entity* itself.

The State Class Pattern

Instead of a giant switch statement in a role file, each state is represented by an object with a run() and transition() method.

```
class EngageState {
    run(creep, context) {
        // Find best position using kiting logic
        if (creep.pos.isNearTo(context.target)) {
            creep.attack(context.target);
        } else {
            creep.moveTo(context.target);
        }
    }

    transition(creep, context) {
        if (creep.hits < creep.hitsMax * 0.5) return 'RETREAT';
        if (!context.target) return 'IDLE';
        return 'ENGAGE';
    }
}
```

This pattern prevents "obscure logic bugs" by making transitions explicit. A unit's state is stored

in its memory, and the main loop simply instantiates the corresponding state class and calls run().

Intent Order and Simultaneous Actions

A military unit should maximize its per-tick value by combining non-exclusive actions. For example, a single creep can perform move, attack, and rangedHeal in the same tick if it possesses the necessary parts.

Action A	Action B	Compatible?	Constraint
attack()	heal()	No	Same action pipeline
rangedAttack()	heal()	Yes	Different pipelines
move()	attack()	Yes	Standard kiting requirement
dismantle()	attack()	No	Same action pipeline
build()	repair()	No	Same pipeline

The state-machine's "Action Resolver" should verify these combinations every tick. High-tier bots use a "greedy" resolver that attempts to execute every available military intent, starting with the most critical (Healing) and ending with secondary effects (Ranged harassment).

Advanced Tactical Maneuvers: Nukes and Power Creeps

In the late game (RCL 8 and beyond), traditional creep combat reaches a stalemate due to repair throughput. Breaking this stalemate requires the integration of non-standard military assets: Nukers and Power Creeps.

Nuke Orchestration logic

Nukes take 50,000 ticks to land and are highly visible. They are rarely used to destroy units; instead, they are used for "Infrastructure Denial."

- **Safe Mode Resetting:** If a defender activates Safe Mode (20,000 ticks), a nuke launched *after* the activation will land after the Safe Mode ends, providing a 200-tick window where the defender cannot Safe Mode again.
- **Structural Shock:** Targeting the Terminal and Storage simultaneously forces the defender to spend vast amounts of energy on emergency repairs, often stalling their upgrader pipeline.

Nuke Metric	Value	Strategic Implication
Flight Time	50,000 ticks	Requires long-term forecasting
Center Damage	10,000,000 hits	Destroys everything but high-tier ramparts
Outer Damage	5,000,000 hits	Forces massive repair costs
Safe Mode Lock	200 ticks	Primary window for siege entry

Power Creeps as Force Multipliers

Power Creeps introduce abilities that disrupt standard state-machine calculations.

- **Exhaust:** Increases fatigue on hostiles, effectively "pinning" a quad in range of towers.

- **Jam:** Forces a creep to repeat its last action, disrupting kiting or retreating state transitions.
- **GeneratePortal:** Linking two Power Creeps allows for the near-instant projection of military force across sectors, bypassing the standard multi-room travel state.

A top-tier military machine must account for these "Debuff" states in its unit FSM. A unit that is Exhausted must immediately transition to an Emergency Shield state, requesting all available nearby healers to prioritize it until the effect wears off.

Implementation Steps for Top-Tier Military Modernization

The roadmap for implementing a modernized military state-machine within the grgisme/screeps codebase involves a systematic overhaul of the sensing and action layers.

Phase 1: The Sensing Layer (Tick 0 - 1000)

Establish the "Analyst" infrastructure. This involves creating singleton modules that monitor room vision and update global metadata.

1. **Hostile Catalog:** Maintain a list of all active players in range and their observed RCL/boost tiers.
2. **Combat Simulation:** Develop a module that can take two creep bodies and simulate their interaction over 10 ticks. This "Battle Simulator" is the backbone of offensive decision-making.
3. **Min-Cut Perimeter:** Automate base layout checks to ensure 100% rampart coverage with optimal buffers.

Phase 2: The Tactical Machine (Tick 1000 - 5000)

Move away from per-creep role logic and toward group-based state-machines.

1. **Quad Movement Engine:** Implement the 2-wide CostMatrix and cohesion-wait logic.
2. **Traffic Reconciler:** Integrate a wavefunction collapse resolution for move intents to prevent squad breakage.
3. **Pre-heal Logic:** Ensure all military healers call heal() on every tick, using projected damage as the targeting priority.

Phase 3: The Strategic Layer (Tick 5000+)

Implement the Overlord/Directive hierarchy for fully autonomous warfare.

1. **Expansion Quarantining:** Upon detecting a hostile expansion within range 2 of the colony, automatically place a Harassment Directive.
2. **Economic Denial:** Use the Tower Draining FSM to target high-energy players, forcing them into a deficit state.
3. **Nuke Interception:** Develop the "Emergency Repair" FSM that detects incoming nukes and prioritizes rampart height on those specific tiles.

Causal Implications of High-Tier Military Design

The engineering data suggests several deep-order insights into the nature of top-tier play. First, military success is fundamentally an economic problem. A bot with superior kiting code will still lose to a bot with superior lab-automation and energy-sharing logic, as the latter can afford to sustain infinite unboosted attrition or a finite T3 boosted steamroll.
Second, the game's deterministic nature means that the "best" move is usually discoverable through simulation. High-tier machines spend significant CPU on "What-If" scenarios: if the quad moves to tile (X,Y), what is the maximum damage it can take, and is the current HPT sufficient to survive?. If the simulation shows a survival chance of <100\%, the state-machine *must* transition to RETREAT. This "Knife-Edge" survival is what makes Screeps military logic an engineering discipline rather than a typical game AI challenge.
Finally, the trend in Screeps warfare is moving toward "Centralized Decision, Decentralized Execution." The strategic directives are issued from the colony center, but the tactical quads must have enough local autonomy to resolve traffic and intent conflicts in real-time.

Conclusions and Engineering Recommendations

The development of a high-tier military state-machine is an iterative process of quantification and abstraction. To satisfy the requirements for a modernized, top-tier system within the context of the user's focus, the following final recommendations are made:

1. **Quantify Defensive Thresholds:** Implement a room-wide DPT vs HPT check. Never fire towers if the enemy's boosted heal capacity exceeds the possible damage. This is the single most important fix to prevent tower draining.
2. **Standardize the Overlord Pattern:** All military operations should be encapsulated in Overlord objects that manage the entire lifecycle from lab-boosting to extraction. This eliminates "sloppy bugs" where units forget to boost or arrive with mismatched TTL.
3. **Master the Quad:** Single-unit combat is obsolete at RCL 8. The quad move engine is a non-negotiable prerequisite for high-tier sieging.
4. **Automate the Safe Mode fail-safe:** Implement path-based threat detection to trigger Safe Mode. Relying on a fixed "hit threshold" is too slow against T3 boosted dismantlers that can destroy a 10M rampart in fewer than 100 ticks.
5. **Implement Pre-healing Pass:** Modify the intent resolution system to call heal() at the start of the tactical loop, prioritizing projected health deltas over current health deltas. This negates the "Alpha-Strike" advantage of defenders.

By strictly adhering to these architectural and mathematical protocols, a Screeps bot transitions from a collection of defensive scripts into a dominant military force capable of project power across the global map. The future of autonomous warfare belongs to those who treat their codebase as a weapons-grade engineering system.

Works cited

1. Screeps #1: Overlord overload - Ben Bartlett, https://bencbartlett.com/blog/screeps-1-overlord-overload/ 2. Encouraging more combat at high GCL | Screeps Forum, https://screeps.com/forum/topic/2809/encouraging-more-combat-at-high-gcl 3. Workflow tips

and prioritization for new players? | Screeps Forum, https://screeps.com/forum/topic/2556/workflow-tips-and-prioritization-for-new-players 4. QP/C: State Machines - Quantum Leaps, https://www.state-machine.com/qpc/srs-qp_sm.html 5. Damage order of operations with partialy damaged boosted tough and pre-healing | Screeps Forum, https://screeps.com/forum/topic/2801/damage-order-of-operations-with-partialy-damaged-boosted-tough-and-pre-healing 6. Screeps #12: Strategic Directives | Field Journal, https://jonwinsley.com/notes/screeps-strategic-directives 7. Screeps #14: Decision Making - Field Journal, https://jonwinsley.com/notes/screeps-decision-making 8. Screeps #3: State of the Automated Union | Ben Bartlett, https://bencbartlett.com/blog/screeps-3-state-of-the-automated-union/ 9. Automating Base Planning in Screeps – A Step-by-Step Guide, https://sy-harabi.github.io/Automating-base-planning-in-screeps/ 10. MATH.md - ryanrolds/screeps-bot-choreographer - GitHub, https://github.com/ryanrolds/screeps-bot-choreographer/blob/main/MATH.md 11. Quad (high volume creep attack) · Issue #69 · JonathanSafer/screeps - GitHub, https://github.com/jordansafer/screeps/issues/69 12. Boosts - Screeps Wiki, https://wiki.screepspl.us/Boosts/ 13. Offensive Strategy : r/screeps - Reddit, https://www.reddit.com/r/screeps/comments/7d6w2t/offensive_strategy/ 14. Great Filters - Screeps Wiki, https://wiki.screepspl.us/Great_Filters/ 15. Defending your room | Screeps Documentation, https://docs.screeps.com/defense.html 16. Why are the towers broken? :: Screeps: World General Discussions - Steam Community, https://steamcommunity.com/app/464350/discussions/0/1699415798767961303/ 17. Screeps #21: Patrolling the Perimeter - Field Journal, https://jonwinsley.com/notes/screeps-patrolling-perimeter 18. Best Attack Strategy? | Screeps Forum, https://screeps.com/forum/topic/2995/best-attack-strategy 19. How do you deal with healer attacks? : r/screeps - Reddit, https://www.reddit.com/r/screeps/comments/4z8bz3/how_do_you_deal_with_healer_attacks/ 20. StructureTower - Screeps Wiki, https://wiki.screepspl.us/StructureTower/ 21. Tower Damage | Screeps Forum, https://screeps.com/forum/topic/1097/tower-damage 22. Automated Base Planning | Screeps Tutorial - YouTube, https://www.youtube.com/watch?v=YcruUDbqa7E 23. Combat - Screeps Wiki, https://wiki.screepspl.us/Combat/ 24. Controller | Screeps Wiki - Fandom, https://screeps.fandom.com/wiki/Controller 25. Auto Safemode - Code for new players :: Screeps: World Help - Steam Community, https://steamcommunity.com/app/464350/discussions/5/152390648081885882/ 26. does anyone automate safe mode? if so what triggers it to activate? : r/screeps - Reddit, https://www.reddit.com/r/screeps/comments/662flg/does_anyone_automate_safe_mode_if_so_what/ 27. Boosted creep bodyparts - forum - Screeps, https://screeps.com/forum/topic/3034/boosted-creep-bodyparts 28. RCL 6 Defense From Duo | Screeps Combat Analysis - YouTube, https://www.youtube.com/watch?v=g-lvDUyaNmM 29. JonathanSafer/screeps: Screeps AI - GitHub, https://github.com/JonathanSafer/screeps 30. Creep | Screeps Wiki | Fandom, https://screeps.fandom.com/wiki/Creep 31. Screeps #25: Arena - Grouping Up - Field Journal, https://jonwinsley.com/notes/screeps-arena-grouping-up 32. Traffic Management | screeps-cartographer, https://glitchassassin.github.io/screeps-cartographer/pages/trafficManagement.html 33. screeps-cartographer, https://glitchassassin.github.io/screeps-cartographer/ 34. Cartographer is an advanced (and open source) movement library for Screeps - GitHub, https://github.com/glitchassassin/screeps-cartographer 35. screeps-cartographer/README.md

at main - GitHub,
https://github.com/glitchassassin/screeps-cartographer/blob/main/README.md 36. Overhealing vs. strictly healing past damage: fixing inconsistencies based off of intent order | Screeps Forum,
https://screeps.com/forum/topic/319/overhealing-vs-strictly-healing-past-damage-fixing-inconsistencies-based-off-of-intent-order 37. Best Practices for State Machine Implementation : r/Unity3D - Reddit,
https://www.reddit.com/r/Unity3D/comments/1b6973c/best_practices_for_state_machine_implementation/ 38. How to transition between states and mix states in a finite state machine?,
https://gamedev.stackexchange.com/questions/7906/how-to-transition-between-states-and-mix-states-in-a-finite-state-machine 39. A functional boilerplate for screeps.com with creeps based on the fsm pattern - GitHub Gist, https://gist.github.com/fd6aae59f4193c63c8c3f28b0aeb51e2 40. Screeps #20: The Great Purge - Field Journal,
https://jonwinsley.com/notes/screeps-great-purge 41. Simultaneous execution of creep actions - Screeps Documentation, https://docs.screeps.com/simultaneous-actions.html 42. Are any Actions of Screeps mutually exclusive to eachother? - Arqade - Stack Exchange,
https://gaming.stackexchange.com/questions/200912/are-any-actions-of-screeps-mutually-exclusive-to-eachother 43. PvP Game discussion | Screeps Forum,
https://screeps.com/forum/topic/473/pvp-game-discussion/35 44. Super Structures and Creeps Idea. | Screeps Forum, https://screeps.com/forum/post/11239 45. [Power] Power creep ideas | Screeps Forum, https://screeps.com/forum/topic/388/power-power-creep-ideas 46. Calculating the probability of winning in a fight given the characters' health, damage and healing? : r/math - Reddit,
https://www.reddit.com/r/math/comments/8typwz/calculating_the_probability_of_winning_in_a_fight/

Architectural Forensics and Modernization: A Deep-Dive Engineering Report on Screeps Architecture

1. Introduction: The Engineering Constraints of the Screeps Environment

Screeps is distinct from virtually every other programming game or simulation due to its strict, deterministic, and resource-constrained execution environment. While it presents as a strategy game, the underlying engineering challenge is akin to designing a real-time embedded operating system or a high-frequency trading algorithm. The primary constraint is not game currency (energy), but computational resources—specifically Central Processing Unit (CPU) time and memory serialization overhead. This report provides an exhaustive architectural analysis of the legacy patterns typical of the grgisme/screeps repository era (circa 2016-2018) and contrasts them with the high-performance, industrial-grade architectures required to compete in the 2024-2025 ecosystem.

The evolution of Screeps architecture has been driven by the "CPU Crisis." As players advance from a single room (RCL 1) to a multi-room empire (GCL 10+), the computational cost of managing creeps scales non-linearly. A naïve loop that iterates through all game objects will inevitably hit the hard CPU limit (typically 20ms-100ms per tick depending on shard and subscription status), causing the bot to "bucket crash." When the bucket empties, the bot skips ticks, leading to creep death, defense failures, and empire collapse.

Consequently, modern architecture is defined by **CPU efficiency per useful action**. Every function call, every memory access, and every object instantiation is scrutinized for its overhead. The shift from the "Legacy Loop" architecture to the "Kernel/Process" architecture represents a fundamental maturation of the community's engineering standards. This report will dissect these paradigms, offering a comprehensive roadmap for transforming a legacy codebase into a scalable, TypeScript-based, kernel-driven system.

1.1 The Physics of the Server: Isolates and Serialization

To understand why specific architectures are favored, one must understand the server's physical execution model. User code runs within a Node.js vm instance, often wrapped in isolated-vm for security and resource accounting.

- **The Isolate Lifecycle:** A common misconception among early players is that the code "restarts" every tick. In reality, the global scope (the Heap) persists across ticks. The server only tears down the isolate when it needs to free resources or when the user uploads new code. This "Global Reset" might happen every 10 ticks or every 10,000 ticks. Modern architectures exploit this by caching massive amounts of data in the Heap (global object), avoiding expensive database reads.
- **The Serialization Wall:** The Memory object is not a live database connection; it is a JSON string backed by MongoDB. At the start of every tick, the server essentially runs

JSON.parse() on your memory string. At the end, it runs JSON.stringify(). This serialization process is charged against the user's CPU limit. As a bot grows, a 2MB memory file can consume 10-15 CPU just to parse, leaving almost no time for actual creep logic. This physical constraint dictates that modern bots must minimize Memory usage in favor of Heap usage.

2. Forensic Analysis of Legacy Architectures (grgisme Era)

Analyzing the architectural patterns prevalent during the active period of the grgisme repository reveals a "First-Generation" approach to Screeps bot design. These patterns, while functionally correct for low-level rooms, become architectural dead-ends at scale.

2.1 The Monolithic Loop Pattern

The hallmark of legacy architecture is the synchronous, monolithic loop found in main.js. In this pattern, the code iterates sequentially through every creep, structure, and room, executing logic immediately.

Typical Legacy Structure:

```
module.exports.loop = function () {
    // Clear dead memory
    for(var name in Memory.creeps) {... }

    // Run Creeps
    for(var name in Game.creeps) {
        var creep = Game.creeps[name];
        if(creep.memory.role == 'harvester') {
            roleHarvester.run(creep);
        }
        if(creep.memory.role == 'upgrader') {
            roleUpgrader.run(creep);
        }
    }

    // Run Towers
    var towers = Game.rooms.find(FIND_MY_STRUCTURES, {filter:
{structureType: STRUCTURE_TOWER}});
    towers.forEach(tower => towerDefend(tower));
}
```

Architectural Deficiencies:

1. **Lack of Preemption:** This loop is brittle. If the roleHarvester logic encounters an edge case (e.g., pathfinding through a complex swamp) and consumes 50ms, the loop times out. The roleUpgrader logic at the end of the list simply never executes. The bot does not "know" it is running out of time; it simply crashes.
2. **Redundant Search Operations:** Legacy roles typically contain their own sensing logic. A harvester might run creep.room.find(FIND_SOURCES) every tick. If there are 10

harvesters, the bot performs 10 identical spatial searches per tick. This O(N) scanning is a primary source of CPU waste.
3. **Hard-Coded dependencies:** The logic is tightly coupled. The main loop knows about specific roles. Adding a new role requires modifying the main loop, violating the Open/Closed Principle of software design.

2.2 Direct Memory Dependency

Legacy bots treat Memory as their primary state store. Every decision is read from and written to Memory.

- *Example:* creep.memory.working = true;
- *Impact:* This forces the V8 engine to constantly interact with the Memory proxy object, preventing optimization. Furthermore, it encourages "state bloating," where developers store massive arrays (like pathfinding caches) in Memory, inflating the serialization cost discussed in Section 1.1.

2.3 The "Role" Abstraction

The grgisme repo likely structures code around "Roles"—individual scripts defining the behavior of a single unit type (role.harvester.js, role.builder.js).

- **The Problem of Independence:** In this model, creeps are autonomous agents. A harvester "decides" to harvest. A builder "decides" to build. This lack of centralized coordination leads to inefficient emergent behavior, such as five builders racing to repair a single road tick, or harvesters clogging a source access point because they didn't coordinate their slots.
- **File Structure Limitations:** The legacy environment required a flat file structure. This discouraged modularity and led to massive files (e.g., a 2000-line main.js or utils.js). It prevented the use of modern design patterns like Strategy or Command, which thrive in deeply nested, class-based directory structures.

3. The Modern Toolchain: From Scripting to Engineering

The transition from 2017-era architectures to 2025 standards begins outside the game, in the development environment. The complexity of modern bots (often exceeding 20,000 lines of code) necessitates a toolchain comparable to enterprise software projects.

3.1 The TypeScript Imperative

While grgisme likely utilizes JavaScript (ES5/ES6), **TypeScript** has become the de facto standard for serious Screeps development. The arguments for TypeScript in this domain extend beyond simple type safety.

3.1.1 Refactoring Confidence

As a Screeps bot evolves, its internal data structures change frequently. A transition from storing sourceId to storing a MiningSite object in memory would introduce silent runtime failures in

JavaScript. TypeScript catches these contract violations at compile time, allowing for aggressive refactoring of the architecture without fear of deploying broken code to a live server where debugging is difficult.

3.1.2 Interface Enforcement

Screeps relies heavily on "Memory Contracts"—the agreement that a creep's memory will contain specific fields (e.g., _move, task). TypeScript interfaces formally define these contracts:

```
interface CreepMemory {
    role: string;
    room: string;
    working: boolean;
    _trav?: TravelData; // Interface for external library
}
```

This ensures that different subsystems (e.g., the Logistics Manager and the Movement System) interact with the creep's memory consistently.

3.2 The Bundling Revolution: Rollup vs. Webpack

Legacy bots uploaded raw files. Second-generation bots used Webpack. The current standard is **Rollup**.

Feature	Webpack	Rollup	Benefit for Screeps
Output Format	Complex wrapper (simulation of CommonJS)	Flat scope hoisting	**Lower CPU Overhead.** Rollup executes code directly in the closure scope, avoiding the function call overhead of Webpack's module loader shim.
Tree Shaking	Good	Excellent	**Reduced Parse Time.** Smaller code size means the server spends less CPU compiling the script on global reset.
Configuration	Complex, aimed at browsers	Simple, aimed at libraries	**Ease of Use.** Easier to configure for the specific "upload to flat file" requirement of the Screeps server.

Architectural implication: The use of Rollup allows developers to use a deep, nested directory structure locally (e.g., src/colonies/hatchery/spawning/SpawnRequest.ts) while deploying a single, highly optimized main.js to the server. This decouples the development experience from the deployment constraints.

3.3 Static Analysis and Linting

Modern toolchains integrate **ESLint** and **Prettier**. In the Screeps context, linting rules are often configured to forbid specific patterns known to be slow, such as Array.prototype.forEach (which is slower than a for loop in V8) or inadvertent global variable leakage. This static analysis acts as a first pass of optimization before the code even reaches the profiler.

4. The Kernel Architecture: Treating the Bot as an Operating System

To solve the "Monolithic Loop" problems identified in the grgisme analysis, modern bots adopt a **Kernel** (or Operating System) architecture. This is the single most important architectural shift for a high-level bot.

4.1 The Kernel Responsibility

The Kernel is a piece of code that runs every tick and manages the execution of "Processes." It abstracts the concept of the Game Loop into a managed environment.

Core Responsibilities:

1. **Resource Management:** It tracks the Game.cpu.bucket and Game.cpu.getUsed().
2. **Scheduling:** It determines which tasks run based on priority.
3. **Error Isolation:** It wraps processes in try/catch blocks. If the "RoadBuilder" process crashes due to a bug, the Kernel catches the error, logs the stack trace, and allows the "TowerDefense" process to still run. In the legacy model, a crash in one role stops the entire loop.

4.2 The Process Model

A "Process" is a class that encapsulates a specific unit of logic and its state.

- **PID (Process ID):** Every running task has a unique ID.
- **Priority:** Integers determining execution order (e.g., Critical = 0, Logistics = 5, Scouting = 10).
- **Sleep/Suspend:** A process can tell the Kernel "Wake me up in 50 ticks." This is crucial for CPU saving. If a Spawner process finishes spawning a creep, it knows it won't be needed for 3 ticks (spawning animation). It sleeps, removing itself from the scheduler and saving the CPU overhead of checking it.

Comparison: Legacy vs. Kernel Loop

Legacy Loop (grgisme)	Kernel Architecture
Iterates all creeps every tick.	Iterates only *active* processes.
Timeouts crash the loop.	Kernel suspends low-priority tasks when CPU is low.
Logic is strictly synchronous.	Supports long-running tasks via "coroutines" (generators).
Hard to debug (anonymous functions).	PIDs trace exactly which logic block failed.

4.3 Implementing Priority Queues

The Kernel relies on a Priority Queue data structure. Since JavaScript is single-threaded, this queue is usually implemented as an array of arrays, indexed by priority.

```
class Kernel {
    processTable: Map<PID, Process>;
    scheduler: PriorityQueue;

    run() {
        while (this.cpuAvailable()) {
            const process = this.scheduler.getNext();
            if (!process) break;
            try {
                process.run();
            } catch (e) {
                this.handleError(e, process);
            }
        }
    }
}
```

This ensures that **Defense** (Priority 0) always runs, even if the bucket is empty, while **Wall Construction** (Priority 9) halts to conserve resources.

5. Memory Engineering: Heap-First Architecture

The standard Screeps advice "keep memory small" is often misinterpreted. The goal is not just small Memory, but *infrequent* usage of the persistent Memory object.

5.1 The Global Cache Pattern

The most performant bots today move almost all active state to the global heap.

- **Mechanism:** On the very first tick of an isolate (Global Reset), the bot parses Memory and hydrates a set of rich JavaScript objects (e.g., RoomManager instances) stored in global.
- **Tick Operation:** For the next 100-5000 ticks, the bot reads and writes to these global objects. Access is instant (nanoseconds).
- **Persistence:** Critical state changes (e.g., a creep dying, a flag moving) are flushed back to Memory at the end of the tick. Ephemeral data (e.g., cached paths, heatmaps) is never written to Memory.

Benefits over Legacy:

- **Zero Serialization Cost:** Temporary data doesn't pay the JSON.stringify tax.
- **Rich Objects:** global can store class instances, functions, and circular references. Memory can only store JSON-compatible trees.
- **Garbage Collection Control:** By reusing the same global objects tick-after-tick, the bot reduces the rate of memory allocation, putting less pressure on the V8 Garbage Collector.

5.2 RawMemory and Segments

For advanced data (like historical market data or huge cost matrices), the 2MB Memory limit is insufficient. Modern bots utilize RawMemory.segments (up to 100MB of storage).

- **Asynchronous Access:** Segments must be requested in one tick and read in the next. The Kernel architecture handles this asynchronous complexity elegantly by having a "MemoryFetcher" process that requests a segment and wakes up the consumer process when the data arrives.

5.3 Inter-Shard Memory (ISM)

As an empire expands to GCL 10+, it will likely span multiple shards (e.g., Shard 0, Shard 1, Shard 2) to find free room slots.

- **The Challenge:** Memory and global are isolated per shard. Shard 0 cannot see Shard 1's variables.
- **The Mechanism:** InterShardMemory allows string communication.
- **Architecture:** A "Portal Manager" process serializes creep intent (e.g., "Creep A is moving to Shard 1 to colonize") and writes it to ISM. The Kernel on Shard 1 reads ISM, deserializes the intent, and spawns a "Ghost Process" to pick up the creep when it arrives. This coordination is impossible in the simple grgisme loop architecture.

6. Hierarchical Control: The Overlord Pattern

Moving away from the "Role" pattern is crucial for coordinating complex behaviors like swarming or precise logistics. The **Overlord** pattern, popularized by the *Overmind* bot, is the industry standard for high-level control.

6.1 Concept: Inversion of Control

In the legacy role pattern, the creep asks: "What should I do?" In the Overlord pattern, the Overlord commands: "You do this."

Structure:

1. **Colony:** The root object for a room. It owns the Controller, Spawns, and Sources.
2. **Overlord:** A virtual manager responsible for a specific *objective*.
    - *MiningOverlord:* Responsible for Source A.
    - *UpgradeOverlord:* Responsible for the Controller.
    - *DefenseOverlord:* Responsible for protecting the room.
3. **Zerg (Creep Wrapper):** An extension of the Creep object that executes Tasks.

6.2 The Logic Flow

1. **Instantiation:** The MiningOverlord for Source A initializes. It checks the room memory. It sees Source A has 3 open access spots.
2. **Creep Check:** It looks at its list of assigned creeps. It sees only 2 miners alive.
3. **Spawn Request:** It sends a request to the Colony's SpawnHatchery: "I need a miner body, priority High."
4. **Task Assignment:** For the 2 existing miners, it assigns a HarvestTask(SourceA).
5. **Execution:** The creeps execute the task. They do not run pathfinding logic or target selection logic; they strictly follow the Overlord's pointer.

6.3 Advantages over Roles

- **CPU Efficiency:** The MiningOverlord runs the logic "Find Source A" *once*. In the legacy model, every miner ran find(FIND_SOURCES) individually. For 100 creeps, this is a massive reduction in spatial lookups.
- **Coordination:** The Overlord knows the global state. It won't send 5 creeps to a source that only has 1 open slot. It perfectly distributes the workforce.
- **Prioritization:** If the colony is under attack, the Colony manager can suspend the UpgradeOverlord completely, freeing up CPU and energy for the DefenseOverlord.

7. Navigation and Cartography: Solving the Traveling Salesman Problem

Movement is arguably the most CPU-intensive subsystem in Screeps. A creep moving from Room A to Room B (50 tiles) running moveTo every tick will consume massive resources recalculating the path, checking for obstacles, and avoiding other creeps.

7.1 Path Caching

Modern bots **never** calculate a path more than once.

- **Algorithm:** When a creep needs to go to a target, the path is calculated using PathFinder.search.
- **Serialization:** The resulting path (array of positions) is compressed into a string (e.g., "uulddr" for up-up-left-down-down-right) and stored in the heap cache.
- **Execution:** On subsequent ticks, the creep blindly follows the direction string. It does not look at the map. It does not check for walls. It assumes the path is valid.
- **Error Handling:** If the creep is blocked (fatigue or obstacle), it increments a "stuck" counter. If "stuck > 3", it recalculates the path.

7.2 The CostMatrix

The default moveTo avoids walls but treats swamps as cost 5 and plains as cost 1. Advanced bots manipulate the **CostMatrix** to define "Virtual Roads."

- **Skirting:** Bots increase the cost of tiles near Source Keepers (SKs) to 255 (unwalkable) to prevent civilian creeps from suiciding into enemies.
- **Preferring Roads:** They lower the cost of Road structures to 1 (or even 0.5 effectively in heuristics) to force pathfinding to stick to established infrastructure.

7.3 Traffic Management: The "Shove" Algorithm

In a dense base (Bunker), creeps constantly cross paths. Standard moveTo makes them stop and wait. **The Best Practice:** Use a library like screeps-cartographer or Traveler that implements **Shoving**.

1. Creep A (Hauler) needs to move onto a tile occupied by Creep B (Upgrader).
2. The Movement System sees the collision.
3. It checks priorities. Hauler > Upgrader.

4. It commands Creep B to move to a random free adjacent tile *this tick*, effectively pushing it out of the way.
5. Creep A moves into the slot. *Result:* Zero tick delay for high-priority logistics.

8. Logistics and Supply Chain Management

The economy of Screeps is a flow problem. Energy is generated at Sources and consumed at Controllers, Spawns, and Towers. The "Role" system (Push) is inefficient compared to the "Request" system (Pull).

8.1 The "Transport Request" Pattern

Instead of Haulers scanning for dropped energy (Pull), structures advertise their needs.

- **Providers:** Sources, Containers, and Links register themselves as "Providers" if they have energy.
- **Requesters:** Spawns, Towers, and Upgraders register as "Requesters" if they need energy.
- **The Broker:** A LogisticsManager runs every tick. It matches Providers to Requesters using a stable matching algorithm (minimizing distance).
- **Assignment:** It assigns a TransportTask to a specific Hauler. "Go to A, Pick up 50, Go to B, Drop off 50."

8.2 Link Balancing

At RCL 5+, Links allow instant energy transfer. A LinkNetwork process must actively balance the network.

- **Logic:** It identifies the "Hub Link" (near Storage) and "Source Links" (near Sources).
- **Action:** When a Source Link is full, the network automatically fires transferEnergy to the Hub Link. This removes the need for long-distance haulers entirely, significantly saving CPU (creep movement costs 0.2 CPU; Link transfer costs constant small CPU).

9. Military Architecture: From Skirmish to War

Legacy bots treat combat as an afterthought (role.attacker). Modern bots treat it as a state machine.

9.1 Tower Defense Logic

Towers are CPU-heavy active structures.

- **Naive:** towers.forEach(t => t.attack(closestEnemy))
- **Optimized:** room.find(FIND_HOSTILE_CREEPS) is called *once* by the DefenseManager.
    - It calculates the "Threat Potential" of each enemy (does it have HEAL parts? ATTACK parts?).
    - It calculates the potential damage of *all* towers combined.
    - If TotalDamage > EnemyHealPower, all towers fire at that single target (Focus Fire).
    - If not, towers conserve energy or target non-healers. This logic defeats standard "Tank/Healer" pairs.

9.2 Squad Coordination

For attacking, individual creeps are useless against defended rooms.

- **Formation:** Bots use a "Squad" object that controls 4 creeps (e.g., a "Quad").
- **Movement:** The Squad calculates a path for the *formation*. It issues move commands to all 4 creeps to keep them strictly adjacent. If one creep fatigues, the whole squad waits.
- **Healing:** Creeps in the squad auto-heal the most damaged member, effectively sharing a single health pool.

9.3 Safe Mode Automation

A grgisme bot might rely on the user to activate Safe Mode. An autonomous bot monitors the hits of critical structures (Spawn, Storage, Towers). If hits < maxHits, it instantly triggers Safe Mode. This prevents a bot from being wiped out while the player is asleep.

10. Empire Management: Automated Planning

The crowning achievement of a modern bot is the ability to place its own structures.

10.1 The Bunker Stamp

High-level players use a fixed layout called a **Bunker**.

- **Design:** A diamond shape containing all Extensions, Spawns, and Towers, tightly packed with Road/Rampart interleaving.
- **Implementation:** The bot has a hardcoded map of the bunker (e.g., BunkerLayout).
- **Placement:** When the room is claimed, the bot runs a DistanceTransform to find the center point (maximum distance from walls). It then "stamps" the layout onto the room coordinates.
- **Construction:** A ConstructionManager checks the RCL. If RCL increases, it checks the stamp to see what new structures are allowed and places the Construction Sites automatically.

10.2 Remote Mining

To fuel the bunker, the bot must exploit neighbor rooms.

- **Scouting:** A ScoutOverlord sends single-move-part creeps to adjacent rooms to check for Sources and threats.
- **Reservation:** A Reserver creep locks the controller of the remote room to double the energy regeneration rate.
- **Hauling:** The LogisticsManager calculates the distance (Distance * 2 * Capacity) to determine exactly how many Hauler parts are needed to drain the remote source without wasting CPU on idle haulers.

11. Migration Roadmap for grgisme/screeps

This section outlines the concrete steps to transform the legacy repository into a modern

powerhouse.

Phase 1: The Foundation (Week 1)

1. **Abandon the Flat Structure:** Do not attempt to refactor the existing main.js. Create a new git branch or repository.
2. **Install the Toolchain:**
    - Initialize a package.json.
    - Install typescript, rollup, rollup-plugin-typescript2.
    - Install @types/screeps for API definitions.
    - Configure tsconfig.json for strict null checks.
3. **Deploy the Starter:** Clone the **screeps-typescript-starter** or similar template to get the build pipeline working. Verify you can upload a "Hello World" to the private server or main server.

Phase 2: The Kernel Implementation (Week 2)

1. **Build the Loop:** Create src/main.ts that instantiates a Kernel class.
2. **Memory Management:** Implement the Heap Cache pattern. global.Empire = new Empire(). Hydrate from Memory on start; save to Memory on end.
3. **Error Mapper:** Install screeps-sourcemaps to ensure that when your TypeScript code errors, the stack trace points to the .ts file, not the compiled .js bundle.

Phase 3: The First Overlord (Week 3)

1. **Mining System:** Don't port the role.harvester. Write a MiningOverlord class.
2. **Logic:** Implement the logic to scan a room, identify sources, creating MiningSite objects in the Heap, and spawn creeps to fill them.
3. **Wrappers:** Create a Zerg class to wrap Creep and add convenience methods (creep.isFull, creep.travelTo).

Phase 4: Logistics and Upgrading (Week 4)

1. **Logistics:** Implement the Request/Provider system. Stop creeps from searching for energy.
2. **Upgrading:** Create an UpgradeOverlord that requests energy from the Logistics system and dumps it into the controller.

Phase 5: Advanced Systems (Week 5+)

1. **Cartographer:** Integrate screeps-cartographer to replace your custom movement logic.
2. **Room Planner:** Implement DistanceTransform to automatically place extensions.
3. **Market:** Write a TerminalOverlord to sell excess energy for credits.

12. Conclusion

The Screeps architecture landscape has matured from simple scripts to complex systems

engineering. The legacy patterns observed in the grgisme era—monolithic loops, direct memory access, and role-based independence—are fundamentally incompatible with the CPU and memory constraints of high-level play in 2025.

By adopting the **Kernel/Process** model, transitioning to **TypeScript/Rollup**, and implementing **Heap-based Caching**, a bot can achieve orders of magnitude better performance. This allows the player to focus on high-level strategy (Empire expansion, War) rather than fighting the limitations of the execution environment. The path forward is not a refactor, but a re-engineering of the bot's very soul, treating it not as a collection of scripts, but as a distributed operating system controlling a swarm of autonomous agents.

Data Summary & Key Metrics Table

Metric	Legacy Architecture (grgisme)	Modern Architecture (Overmind/OS)	Improvement Factor
Code Structure	Flat Files / Spaghetti Loop	Nested Modules / Kernel / Process	High Maintainability
Language	JavaScript (Dynamic)	TypeScript (Static)	Refactoring Safety
Memory Access	Direct Memory I/O per tick	Heap Cache (global)	~10x CPU reduction
Creep Logic	Independent Roles (Pull)	Overlord/Task Assignment (Push)	Coordination Efficiency
Movement	moveTo (Recalc every tick)	Cached Path + Traffic Shoving	~5x CPU reduction
Build Tool	None / Grunt	Rollup + Tree Shaking	Fast Deploy / Small Code
Scalability	Caps at ~RCL 5 / GCL 3	Scales to GCL 30+	Infinite Scaling

The adoption of these best practices is the only viable path to competitiveness in the modern Screeps World.

Works cited

1. How does CPU limit work | Screeps Documentation, https://docs.screeps.com/cpu-limit.html 2. Optimizations roadmap | Screeps Blog, https://blog.screeps.com/2017/06/optimizations/ 3. Caching Overview | Screeps Documentation, https://docs.screeps.com/contributed/caching-overview.html 4. Releases · bencbartlett/Overmind - GitHub, https://github.com/bencbartlett/Overmind/releases 5. What is your organizational structure for your creeps? : r/screeps - Reddit, https://www.reddit.com/r/screeps/comments/5epp14/what_is_your_organizational_structure_for_your/ 6. Modifying object prototypes - Screeps Documentation, https://docs.screeps.com/contributed/modifying-prototypes.html 7. CPU Optimization | Screeps Forum, https://screeps.com/forum/topic/1614/cpu-optimization 8. Global Objects | Screeps Documentation, https://docs.screeps.com/global-objects.html 9. Hierarchical Script Modules | Screeps Forum, https://screeps.com/forum/topic/993/hierarchical-script-modules 10. Grunt Screeps with folders - Reddit, https://www.reddit.com/r/screeps/comments/8pa6uk/grunt_screeps_with_folders/ 11. Starter kit for TypeScript-based Screeps AI codes. - GitHub, https://github.com/screepers/screeps-typescript-starter 12. Screeps Typescript Starter: Introduction, https://screepers.gitbook.io/screeps-typescript-starter 13. JavaScript vs TypeScript:

A Comprehensive Comparison | by Navindu Amerasinghe | Jan, 2026 | Medium, https://medium.com/@navinduamerasinghe/javascript-vs-typescript-a-comprehensive-comparison-c87a97397b37 14. TypeScript vs JavaScript - A Detailed Comparison - Refine, https://refine.dev/blog/javascript-vs-typescript/ 15. Screeping | World Domination via JavaScript, https://screeping.wordpress.com/ 16. Why I use Rollup, and not Webpack | by Paul Sweeney | Medium, https://medium.com/@PepsRyuu/why-i-use-rollup-and-not-webpack-e3ab163f4fd3 17. Module bundling - Screeps Typescript Starter - GitBook, https://screepers.gitbook.io/screeps-typescript-starter/in-depth/module-bundling 18. ryanrolds/screeps-bot-choreographer - GitHub, https://github.com/ryanrolds/screeps-bot-choreographer 19. Screeps after one year - Pedantic Orderliness, https://www.pedanticorderliness.com/posts/screeps 20. CPU - Screeps Wiki, https://wiki.screepspl.us/CPU/ 21. A generalized solution for heap and memory caching in Screeps - GitHub, https://github.com/glitchassassin/screeps-cache 22. OOP Ideas for Screeps/JS, https://screeps.com/forum/topic/2777/oop-ideas-for-screeps-js 23. Find & Fix Node.js Memory Leaks with Heap Snapshots - Halodoc Blog, https://blogs.halodoc.io/fix-node-js-memory-leaks/ 24. API - Screeps Documentation, https://docs.screeps.com/api/ 25. InterShardSegments for each shard rather than a shared one. | Screeps Forum, https://screeps.com/forum/topic/2026/intershardsegments-for-each-shard-rather-than-a-shared-one 26. Great Filters - Screeps Wiki, https://wiki.screepspl.us/Great_Filters/ 27. README.md - bencbartlett/Overmind - GitHub, https://github.com/bencbartlett/Overmind/blob/master/README.md 28. Screeps #1: Overlord overload | Ben Bartlett, https://bencbartlett.com/blog/screeps-1-overlord-overload/ 29. How to Cache paths? : r/screeps - Reddit, https://www.reddit.com/r/screeps/comments/6yr4em/how_to_cache_paths/ 30. Dijkstra flood fill, path to all destinations | Screeps Forum, https://screeps.com/forum/topic/2405/dijkstra-flood-fill-path-to-all-destinations 31. Cartographer is an advanced (and open source) movement library for Screeps - GitHub, https://github.com/glitchassassin/screeps-cartographer 32. Bunkers · bencbartlett/Overmind Wiki - GitHub, https://github.com/bencbartlett/Overmind/wiki/Bunkers 33. Creeps | Screeps Documentation, https://docs.screeps.com/creeps.html 34. Best Attack Strategy? | Screeps Forum, https://screeps.com/forum/topic/2995/best-attack-strategy 35. Unusual body part combinations | Screeps Forum, https://screeps.com/forum/topic/2744/unusual-body-part-combinations 36. arXiv:1706.02789v1 [cs.AI] 8 Jun 2017, https://arxiv.org/pdf/1706.02789 37. Defending your room | Screeps Documentation, https://docs.screeps.com/defense.html 38. Automating Base Planning in Screeps – A Step-by-Step Guide, https://sy-harabi.github.io/Automating-base-planning-in-screeps/ 39. Tips for multi-room architecture? : r/screeps - Reddit, https://www.reddit.com/r/screeps/comments/6uzqkb/tips_for_multiroom_architecture/ 40. Screeps #5: Refactoring for Remote Mining - Field Journal, https://jonwinsley.com/notes/screeps-refactoring-remote-mining 41. How do I get sourcemaps my original TypeScript code using rollup and terser?, https://stackoverflow.com/questions/68187532/how-do-i-get-sourcemaps-my-original-typescript-code-using-rollup-and-terser

The Genesis Protocol: Architecting Autonomous Colony Bootstrapping in Screeps

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 the most significant architectural hurdle in contemporary distributed simulation engineering. While the Screeps environment is often perceived through the lens of a strategy game, a forensic analysis of its underlying mechanics reveals it to be a real-time, resource-constrained distributed operating system. The primary bottlenecks for any automated agent are not game-specific currencies like energy, but the hard limits of Central Processing Unit (CPU) cycles and the heavy overhead of memory serialization.

This report delineates the "Genesis Protocol," an exhaustive architectural framework for autonomous colony bootstrapping and lifecycle management. It addresses the transition from Room Controller Level (RCL) 1 (Survival Mode) to RCL 4 (Storage Mode), focusing on the critical evolution of the worker unit, the mathematical prevention of "Death Spirals," and the systems-level handshaking protocols required to maintain continuous operational logic within a Kernel/Process architecture.

The Engineering Constraints of the Screeps Environment

To architect a bootstrapping protocol that avoids systemic collapse, one must first account for the physical execution model of the Screeps server. User code runs within a Node.js virtual machine isolate, often managed by isolated-vm for resource accounting. This environment is strictly deterministic and operates on a discrete tick-based lifecycle.

The Isolate Lifecycle and Heap Persistence

A common misconception in early-game development is that the code "restarts" every tick. In reality, the global scope—the V8 Heap—persists across ticks unless the server explicitly tears down the isolate to reclaim resources or the user uploads new code. This "Global Reset" might happen every 10 ticks or every 10,000 ticks. Modern architectures, such as the one implemented in the *grgisme/screeps* repository, exploit this persistence by caching massive amounts of data in the global object, bypassing the expensive JSON.parse() costs associated with the Memory object.

The Serialization Wall

The Memory object is not a live database connection; it is a JSON string backed by MongoDB. At the start of every tick, the server runs JSON.parse() on the user's memory string, and at the end, it runs JSON.stringify(). This process is charged against the user's CPU limit. As a bot expands, a 2MB memory file can consume upwards of 15ms of CPU just for serialization,

leaving almost no time for actual creep logic or pathfinding. This constraint dictates that the Genesis Protocol must favor ephemeral heap-based structures for task management, only persisting critical "intent" states to the Memory object.

Feature	Legacy Approach (Procedural)	Genesis Protocol (Kernel-Driven)	Impact on Performance
State Storage	Primary reliance on Memory object.	Primary reliance on Global Heap Cache.	10x CPU reduction in JSON parsing.
Data Lifecycle	Re-parsed and re-processed every tick.	Persistent classes updated via refresh().	Minimizes garbage collection overhead.
Memory Limit	2MB hard cap on Memory string.	2MB Memory + ~256MB Heap + 100MB Segments.	Enables complex historical analysis.
Sensing	Direct Room.find() calls per creep.	Centralized Registry with O(1) lookups.	Eliminates redundant spatial searches.

The Kernel Architecture: Managing the CPU Crisis

The Genesis Protocol utilizes a Kernel-driven architecture to solve the "Monolithic Loop" failures typical of early-generation bots. In legacy patterns, the code iterates sequentially through every creep, structure, and room. This structure lacks preemption; if a harvester's pathfinding logic encounters a complex swamp and consumes 50ms, the bot times out, and critical defense or hatchery logic never executes.

The Process Model and Priority Scheduling

The Kernel abstracts the game loop into a managed environment, treating the bot as an operating system. Logic is encapsulated into "Processes"—classes with unique IDs (PIDs) and integer-based priorities.

1. **Scheduling:** The Kernel determines which tasks run based on urgency. In a bootstrapping room, "Hatchery Refilling" (Priority 0) always runs, while "Scouting" (Priority 9) can be halted if the CPU bucket is empty.
2. **Error Isolation:** Processes are wrapped in try/catch blocks. If a "Road Builder" process crashes due to a site being blocked, the Kernel logs the error and allows the "Tower Defense" process to continue.
3. **Process Suspension:** A process can tell the Kernel to "wake it up" after a specific duration. For example, the "Spawner" process knows a creep takes 3 ticks per body part to produce and can suspend itself, saving CPU overhead.

The Evolution of the Worker: Polymorphism and Hierarchy

As a colony levels up, the role of the worker must evolve from a "Universal Unit" to a group of "Specialized Agents." The challenge in a Kernel/Process architecture is implementing this polymorphism without violating the Overlord -> Zerg command pattern.

RCL 1: The Universal Bootstrapper (Survival Mode)

At RCL 1, the colony is limited to a single Spawn with 300 energy capacity. The morphology of the worker is constrained by this budget, typically resulting in a body. At this stage, specialization is an inefficiency. A dedicated harvester at RCL 1 would waste ticks waiting for a hauler, and a dedicated upgrader would sit idle while the spawn is empty.
The Genesis Protocol implements the "Universal Worker" at RCL 1, utilizing a prioritized task chain:

1. **Harvesting:** The unit moves to a source to harvest energy.
2. **Self-Logistics:** If the unit has energy, it checks if the Spawn needs energy; if so, it fulfills that request.
3. **Controller Pushing:** If the Spawn is full, the unit moves to the Controller to upgrade it.

RCL 4: Functional Specialization (Storage Mode)

Upon reaching RCL 4 and constructing a StructureStorage, the economic landscape shifts. The colony gains a 1,000,000-unit buffer, allowing for the decoupling of production and consumption. At this level, the "Universal Worker" becomes an economic liability. Specialized units can be produced with optimized bodies that maximize "Part-Tick Value".

Unit Type	Optimized Body (RCL 4)	Functional Goal
Static Miner	$$	100% extraction of a 3000-energy source.
Logistics Hauler	$$	Distance-efficient energy transport.
Upgrader	$$	High-throughput GCL advancement.
Builder	$$	Rapid infrastructure deployment.

Implementing Polymorphism via the Zerg Wrapper

To maintain the Overlord -> Zerg pattern while allowing for specialized behaviors, the Genesis Protocol utilizes a "Task-Based Wrapper" model. The Zerg class is an extension of the standard Creep object that executes specific Task objects assigned by an Overlord.
Polymorphism is achieved through **Inversion of Control (IoC)**:

- **The Overlord commands:** "You are now a Miner." It assigns a HarvestTask(sourceID).
- **The Zerg executes:** It follows the pathing and intent logic defined in the Task.
- **Dynamic Re-tasking:** When a Builder finishes all construction sites, its BuilderOverlord identifies it as isIdle. The UpgradeOverlord can then "adopt" this unit and assign it an UpgradeTask, transforming its behavior without requiring a change in role string or morphology.

This separation ensures that high-level strategic goals (like rushing RCL 4) do not interfere with the micro-optimizations required for individual unit performance.

The Death Spiral Prevention Mechanism

A "Death Spiral" is a catastrophic feedback loop where a colony's energy consumption exceeds its production to the point that it cannot spawn the units required to harvest more energy. This

typically occurs when a bot aggressively spawns Upgraders that drain the Spawn's energy before the Mining infrastructure is saturated.

The Metabolic Tier: Critical Survival

The Genesis Protocol implements a "Tiered Priority Model" for the Hatchery spawn queue to ensure "Metabolic Maintenance" is never compromised.

1. **Metabolic Tier (Critical):** Small "Bootstrapper" harvesters and "Fillers" that distribute energy to extensions.
2. **Defensive Tier (Urgent):** Combat units and tower-refilling creeps.
3. **Economic Tier (Standard):** Standard miners, haulers, and upgraders.
4. **Strategic Tier (Low):** Pioneer units and remote exploiters.

The Safety Check and Emergency Mode

At the beginning of every tick, the Hatchery performing a **Safety Check** evaluates two primary variables: the room's energyAvailable and the count of active harvesters. If the system detects zero active creeps capable of harvesting energy, it enters **"Emergency Mode"**.
In Emergency Mode, the Hatchery logic:

- **Overrides the Queue:** It bypasses the current top request if that request's cost exceeds the current energyAvailable.
- **Generates a Bootstrapper:** It spawns a minimal unit costing exactly **300 energy**. This cost is critical because a Spawn automatically regenerates 1 energy per tick until it reaches its 300-capacity.
- **Metabolic Restoration:** The Bootstrapper harvests energy to refill extensions, allowing the room to "bootstrap" itself back to a functional level where it can afford standard-size units.

The Mathematical Upgrader Trigger

Spawning a dedicated Upgrader is the primary cause of energy starvation. To prevent this, the Genesis Protocol defines a precise formula for spawning Upgraders based on **"Energy Surplus"** and **"Effective Store"**. The system should never spawn an upgrader if the energy required for metabolic units is not secured.
The "Effective Store" (S_{eff}) of a structure accounts for resources currently "in flight":
The trigger for a dedicated Upgrader spawn (U_{trigger}) is defined as:
Where:

- T_{crit} is the "Critical Energy Buffer," typically calculated as the cost of spawning a full set of replacement Miners and Haulers plus 5,000 ticks of road maintenance.
- Flow_{net} is the net change in energy per tick, calculated by the "Economy Analyst":

By using this formula, the bot ensures that the Hatchery is only drained when there is a documented surplus, effectively preventing the starvation of the metabolic core.

Construction Site Prioritization: The Genesis Build Order

In a fresh room with 50+ construction sites, the sequence of construction determines the

"Velocity of Growth." Building extensions before sources are secured leads to idle capacity, while building roads before containers leads to massive energy loss from decay.

Phase 1: Source Containers (The Logistics Foundation)

The Genesis Protocol establishes **Source Containers** as the highest priority construction sites. This is driven by the physics of resource decay and the ROI of static mining.

- **Decay Mechanics:** Dropped energy decays at a rate of \lceil amount / 1000 \rceil per tick. For a source producing 10 energy per tick, the accumulation of dropped energy results in significant loss.
- **ROI Analysis:** A container in a remote room decays by 5,000 hits every 100 ticks, costing 0.5 energy per tick to maintain. The energy saved from preventing dropped-resource decay "significantly exceeds" this maintenance cost.
- **Enabling Specialization:** Containers allow for "Static Harvesting," where a miner (5 WORK) sits on a container and harvests at maximum rate, while a hauler picks up full batches of energy. This is more efficient than a single worker moving back and forth.

Phase 2: Extensions (Morphological Scaling)

Following the stabilization of the energy loop, the priority shifts to Extensions. Extensions do not increase energy production; they increase **Spawn Bandwidth** and unit efficiency.

- **RCL 2 to 3 Transition:** Increasing spawn capacity from 300 to 550 allows for larger creep bodies. At RCL 3, where capacity reaches 800, a single hauler with 10 CARRY parts can meet the demand of a 50-tile remote source, whereas RCL 2 would require two haulers.
- **CPU Optimization:** Spawning larger, more efficient creeps reduces the total headcount (N), thereby reducing the CPU overhead of pathfinding and intent processing (O(N)).

Phase 3: Defensive Readiness and Infrastructure

Once the economic engine is self-sustaining, the protocol prioritizes defensive structures and roads.

Structure	Build Priority	Economic Justification
Source Container	1 (Critical)	Prevents energy decay; enables static mining.
Controller Container	2 (High)	Stabilizes upgrader energy supply; reduces movement ticks.
Extensions	3 (High)	Increases morphology efficiency and reduces total unit count.
Towers	4 (Urgent)	Provides active defense and reduces Safe Mode reliance.
Roads	5 (Standard)	Reduces MOVE part requirement and fatigue; increases speed.
Storage	6 (Strategic)	Provides a 1M unit buffer for market trade and sieges.

The Bootstrap Handoff: The Orphan Adoption Protocol

The transition from "Emergency Mode" (Hatchery-driven survival) to "Control Mode" (Overlord-driven execution) is a critical juncture. A failure in this handoff results in "Orphan Creeps"—units that consume resources but perform no logic—and "Zombie Processes"—overlords waiting for creeps that no longer exist in memory.

The Three-Phase Handshake

The Genesis Protocol implements a "Three-Phase Handshake" to pass control of a unit from the Hatchery service to the requesting Overlord.

1. **Phase I: The Commitment (Tick N):** The Hatchery identifies an available spawn and calls spawnCreep(). Crucially, it attaches metadata to the creep's memory object at the point of creation, including the **Overlord PID** and the intended role.
2. **Phase II: The In-Utero Period (Tick N+1 to N+M):** The creep is physically spawning. During this time, the Hatchery tracks its name in a "Pending Spawns" list in the global heap.
3. **Phase III: The Delivery (Tick N+M+1):** Once !creep.spawning is detected, the Hatchery executes the registration handshake. It calls the adopt() method on the requesting Overlord process, passing the creep's handle. The Overlord then adds the unit to its "Zerg" array and begins task assignment.

The Memory Cleanup Pitfall

A major challenge in TypeScript-based bots is the 1-tick delay in creep instantiation. When spawnCreep() returns OK, the memory entry is created immediately, but the Game.creeps object is not updated until the next tick. If a memory cleanup script (which deletes data for dead creeps) runs between these two events, it will delete the memory of the creep currently being spawned. The Genesis Protocol solves this by instructing the cleanup script to ignore any names present in the Hatchery's **Pending Spawns** heap list.

The Orphan Adoption (Subreaper) Logic

In scenarios where a bot recovers from a global reset or a process crash, creeps may exist without a controlling Overlord. These are classified as "Orphan Creeps." The Genesis Protocol includes a **Subreaper Process** within the Colony Overseer to handle re-parenting.

- **Detection:** Every 10 ticks, the Overseer scans all living creeps and compares their memory.overlordPID against the active process table.
- **Categorization:** Orphans are categorized by body parts (e.g., WORK parts indicate a potential Worker, ATTACK indicates a Defender).
- **Adoption:** The Subreaper assigns the orphan to the most appropriate Overlord. For instance, a WORK creep in a room with a deficit of upgraders is re-parented to the UpgradeOverlord.

This ensures that no unit—representing a significant energy and production investment—is left idle, maintaining 100% duty cycle across the colony.

Algorithmic Orchestration: The Global Logistics Broker

The economy of a Screeps colony is essentially a flow problem. Energy is produced at sources and consumed at controllers, spawns, and towers. The Genesis Protocol moves away from "Role-based" logistics (where creeps independently pull resources) to a "Request-based" broker managed by stable matching algorithms.

Standardizing the Transport Request Interface

To coordinate the supply chain, all game objects register their needs with the **Logistics Broker** using a standardized interface.

- **Providers:** Structures with energy (Sources via Miners, Containers, Links).
- **Requesters:** Structures needing energy (Spawns, Towers, Controller).
- **Buffers:** High-capacity structures (Storage, Terminal) that act as either depending on the colony's delta.

The Gale-Shapley Stable Matching Algorithm

The core of the broker is the matching algorithm. While a "Greedy" approach (nearest creep to highest priority) is simple, it fails to achieve global optimality. The Genesis Protocol utilizes a variation of the **Gale-Shapley Algorithm** (the Stable Marriage problem).

1. **Proposal:** Each free hauler (proposer) proposes to its most preferred request based on a heuristic ranking.
2. **Tentative Matching:** Each request (receiver) maintains its current best proposal and rejects inferior ones.
3. **Heuristic Ranking:** Preference is calculated using: *Where Priority is Towers (10) > Spawns (5) > Upgraders (1).*

This "Command and Control" approach ensures that haulers are never racing for the same pile of energy, and critical structures like Towers are refilled instantly during an attack.

The Energy Reservation Ledger

To prevent "Energy Racing," the broker implements a reservation ledger. When a hauler is matched to a requester, the broker calculates the **Effective Store** :
If a hauler is carrying 200 energy to a Tower, the ledger immediately updates the Tower's "Effective Store" to full, even if the hauler is 10 ticks away. Subsequent matching logic will not dispatch redundant units to that Tower.

Resource Engine Optimization: Remote Mining Sovereignty

The ultimate goal of the bootstrapping phase is to transition from local mining to a distributed, multi-room resource engine. This represents the primary inflection point in a colony's developmental trajectory.

Doubling Yield through Room Reservation

Each energy source in the Screeps world is governed by a 300-tick regeneration cycle. In unreserved rooms, sources provide 1,500 energy. The Genesis Protocol utilizes CLAIM parts to execute the reserveController intent, which doubles the capacity to 3,000 units per cycle (10 energy per tick).
To minimize the cost of the expensive, short-lived CLAIM part, the protocol uses **"Buffer Cycling"** :

- The reservation buffer can store up to 5,000 ticks.
- A reserver is only spawned when the buffer falls below a safety threshold: This ensures maximum yield with minimum unit uptime.

Part-Count Balancing vs. Headcount

Top-tier resource engines utilize **Part-Count Balancing** to determine hauler requirements based on Actual Demand.
Instead of spawning a fixed number of creeps, the engine calculates the total CARRY parts needed for a route:
Example: For a reserved source (10 e/t) at 50 tiles, 20 CARRY parts are required. At RCL 3, this is fulfilled by two haulers; at RCL 8, by a single efficient unit.

Infrastructure: Road-Repair-on-Transit

To maintain the remote road network without the overhead of dedicated repair units, the Genesis Protocol equips haulers with a single WORK part. As the hauler moves back and forth, it checks the road tile it stands on. If the hits are below 100%, it issues a repair intent. This "Repair-on-Transit" pattern ensures road health at zero additional CPU or pathing cost.

The Architect process: Automated base planning

A high-performance bot cannot rely on manual placement of structures. The Genesis Protocol integrates an **Architect Process** that utilizes computational geometry to identifies the optimal base layout.

The Distance Transform Algorithm

To identify the center of the colony, the Architect runs a **Distance Transform** algorithm. This calculates the distance of every walkable tile from the nearest wall or exit. By targeting tiles with a distance value of \ge 3, the bot ensures enough open space for a compact "Bunker Stamp" or "Flower Layout".
Once a candidate center point is found, a **Floodfill** algorithm categorizes surrounding tiles by accessibility to the Storage and Terminal. Because logistics costs are a product of distance and frequency, the Hatchery and Labs are placed as close to the central storage hub as possible.

The Bunker Stamp

The Genesis Protocol favors a fixed layout called a "Bunker." This diamond-shaped

configuration contains all Extensions, Spawns, and Towers, interleaved with Ramparts and Roads. The use of a standardized stamp allows the bot to predict pathing costs before structures are built, eliminating the need for findClosestByPath calls and saving massive amounts of CPU during the high-intensity bootstrapping phase.

Military Architecture and Defensive Logic

The military capability of a colony is an emergent property of its economic stability. A bot with superior combat code will still lose if its hatchery is starved or its towers run dry.

Quantitative Defensive Thresholds

High-tier defense is a game of energy efficiency. The Genesis Protocol implements a "Survivability Threshold" calculation before firing Towers.

- **Threat Analysis:** The DefenseManager calculates the "Threat Potential" of hostile units (e.g., HEAL parts, T3 boosts).
- **Hold Fire Logic:** If a hostile unit's boosted heal-per-tick (HPT) exceeds the combined damage-per-tick (DPT) of all towers at their current range, the towers enter a "Hold Fire" state.
- **Tower Draining Prevention:** This prevents attackers from executing a "Drainer State-Machine"—oscillating in and out of the room to force the defender to waste 10 energy per shot on an unkillable target.

Automated Safe Mode Trigger

Safe Mode is a non-renewable resource and must be guarded by sophisticated trigger criteria. The "Fail-Safe" state is entered under specific path-based threats :

1. **Critical Breach:** A rampart protecting a core structure (Spawn, Storage) falls below 10,000 hits and a hostile unit is adjacent.
2. **Pathfinding Threat:** The ThreatAnalyst identifies a valid path from a hostile to the Spawn that does not pass through a rampart.

Conclusion: The Biological Imperative of Autonomy

The implementation of the "Genesis Protocol" transforms a Screeps bot from a collection of reactive scripts into a distributed, autonomous organism. By adopting a Kernel/Process architecture, the developer moves the point of control from the individual unit to the colony hierarchy, allowing for global optimization and resilience against systemic failure.

The evolution of the worker unit through Zerg task-wrapping enables a seamless transition from the "Survival Mode" of RCL 1 to the specialized "Storage Mode" of RCL 4. The mathematical precision of the "Safety Check" and "Effective Store" calculations effectively eliminates the "Death Spiral," ensuring that the metabolic core of the hatchery is always fueled. Furthermore, the rigorous prioritization of infrastructure based on ROI, the implementation of stable matching in logistics, and the enforcement of the three-phase registration handshake ensure that every unit of energy and every CPU cycle is translated into imperial expansion.

The path forward for an automated empire lies in viewing the codebase not as a sequence of instructions, but as a weapons-grade engineering system. Through the integration of

Gale-Shapley matching, recursive stoichiometry, and automated base planning, the Genesis Protocol provides the architectural blueprint for absolute sovereignty in the persistent, competitive world of Screeps. Autonomy is no longer just a feature; it is the fundamental metabolic requirement for survival and dominance.

Works cited

1. OOP Ideas for Screeps/JS, https://screeps.com/forum/topic/2777/oop-ideas-for-screeps-js 2. Control | Screeps Documentation, https://docs.screeps.com/control.html 3. The SIMPLEST Screeps Tutorial - LearnCodeByGaming.com - Learn Code By Gaming, https://learncodebygaming.com/blog/the-simplest-screeps-tutorial 4. role.worker - rasmusbergpalm/screeps - GitHub, https://github.com/rasmusbergpalm/screeps/blob/master/role.worker 5. Getting Started - Screeps Wiki, https://wiki.screepspl.us/Getting_Started/ 6. How important are containers? : r/screeps - Reddit, https://www.reddit.com/r/screeps/comments/5l5wd2/how_important_are_containers/ 7. RC1.0 Design: Basic Harvesting | Screeping - WordPress.com, https://screeping.wordpress.com/2021/05/01/episode-4-1-rc1-basic-harvesting/ 8. RC2.0 Design : Construction and Dedicated Creeps | Screeping - WordPress.com, https://screeping.wordpress.com/2021/05/07/rc2-0-design-construction-and-dedicated-creeps/ 9. Screeps #1: The Game Plan | Field Journal, https://jonwinsley.com/notes/screeps-game-plan 10. Orphan process - Wikipedia, https://en.wikipedia.org/wiki/Orphan_process 11. (PDF) Supervised Learning For Orphan Adoption Problem In Software Architecture Recovery - ResearchGate, https://www.researchgate.net/publication/311795858_Supervised_Learning_For_Orphan_Adoption_Problem_In_Software_Architecture_Recovery

Chemical Supremacy: Automated Laboratory Systems and Strategic Boosting Protocols in Screeps Infrastructure

The transition of a decentralized autonomous colony within the Screeps environment from the mid-game energy economy to the late-game chemical hegemony represents the most significant architectural hurdle for any automated system. While the early stages of development are characterized by the acquisition of energy and the expansion of Room Controller Levels (RCL), the attainment of RCL 6 introduces the mineral economy, and RCL 7 and 8 introduce the complex laboratory systems required for high-level competition. The fundamental goal of these systems is the automated production of Tier 3 (T3) compounds, which can amplify creep performance by factors of up to four hundred percent, transforming a standard unit into a specialized agent capable of dismantling entire unboosted armies. This report provides an exhaustive engineering analysis of the best practices for laboratory automation, mineral logistics, and strategic boosting, contrasting the legacy patterns of the grgisme/screeps era with the high-performance, process-driven architectures of the modern ecosystem.

The Physical Architecture of Laboratory Systems: Spatial Logic and Geometric Optimization

The deployment of laboratory structures is governed by a rigid geometric constraint: a laboratory performing a reaction must be within a range of two squares of its two reagent sources. This simple rule necessitates a highly optimized spatial layout to maximize throughput while minimizing the logistics overhead of the "Scientist" creep responsible for resource shuffling.

Geometric Optimization and Layout Paradigms

In a 10-lab environment—the maximum allowed at RCL 8—the objective is to designate two "Source" or "Input" labs that can supply reagents to the remaining eight "Sink" or "Output" labs. If the source labs are centrally located, the output labs can be clustered around them such that all eight outputs are within range two of both inputs. This allows for a massive parallelization of the runReaction command, producing up to 80 units of mineral compound every 10 to 160 ticks, depending on the reaction type and cooldown period.
Two primary layouts have emerged as the industry standard: the "Flower" and the "Diamond." The Flower layout centers two labs and surrounds them in a hexagonal or circular pattern, which is often favored for its accessibility to logistics creeps. The Diamond layout, often integrated into a 13x13 or 15x15 "Bunker" stamp, aligns labs in a grid that is easier to protect with ramparts but may introduce slight pathing inefficiencies for the Scientist. A variation of these, the "Figure-8" layout, facilitates scaling from RCL 6 (3 labs) to RCL 8 (10 labs) by adding

labs in a predictable progression that maintains range-two integrity.

Layout Feature	Flower Cluster	Diamond Grid	Figure-8 Progression
Input Lab Placement	Centralized / Dual-Hub	Grid-Aligned	Centralized / Linear
Output Lab Capacity	8 (Parallel)	8 (Parallel)	1 to 8 (Scalable)
Scientist Access Tiles	1 to 2	2 to 4	1 to 2
Rampart Coverage	Circular / Tight	Square / Efficient	Variable
Recommended Use	Compact Bases	Bunker-Style Planning	Progressive Growth

Algorithmic Lab Placement: Distance Transform and Floodfill

For a top-tier bot, manual placement of labs is an architectural dead-end. The system must utilize automated room planning algorithms to identify the optimal lab site during the "Colony Boot" phase. The primary algorithm used is the Distance Transform, which calculates the distance of every walkable tile from the nearest wall or exit. By targeting tiles with a distance value of at least 3, the bot ensures there is a 5x5 or larger open space suitable for a lab cluster. Once a candidate center point is found, a Floodfill algorithm is executed to categorize surrounding tiles by their accessibility and proximity to core structures like the Terminal and Storage. Because the Scientist creep must move frequently between the Terminal and the Labs, the "Logistics Cost" (the product of distance and transfer frequency) must be minimized. The most efficient systems place the lab cluster immediately adjacent to the Terminal, often sharing a single standing tile for the Scientist creep to minimize move intents.

The Science Overlord: A Process-Driven Architectural Framework

The transition from legacy "Role" based scripts to a "Kernel/Process" architecture is nowhere more critical than in laboratory management. A "Science Overlord" acts as a high-level manager that orchestrates the execution of "Science Processes," abstracting the complexity of chemical stoichiometry away from the general creep logic.

The Kernel Model and Lab Processes

In the modern kernel architecture, the Science Overlord does not iterate through creeps; instead, it manages a persistent state machine for each room's laboratory complex. This state machine tracks the "Lab Order"—a data structure containing the target product, the required amount, and the current progress. The Overlord divides this mission into several discrete sub-processes:

1. **Reaction Scheduler:** Calculates the necessary precursors and invokes runReaction whenever a lab's cooldown reaches zero.
2. **Logistics Manager:** Interfaces with the room's logistics queue to request a Scientist creep to fulfill reagent needs.
3. *Boost Requester:* Monitors the colony's defensive and offensive status to trigger the production of combat compounds.

This separation of concerns allows the bot to handle complex, long-running chemical chains without blocking the main execution loop. If a specific reaction requires 5,000 ticks to complete, the process simply persists in the heap memory, waking up only when the Scientist has

deposited new reagents or a reaction has completed.

Recursive Dependency Management and Stoichiometry

The most significant challenge in lab automation is the multi-tiered nature of the reaction graph. To produce a T3 compound like Catalyzed Ghodium Acid (XGH_{2}O), the system must first produce Ghodium (G), Hydroxide (OH), and Ghodium Acid (GH_{2}O). A top-tier Science Overlord utilizes a recursive formulation function that takes a target resource and quantity and returns a list of "Missing Precursors."
The stoichiometry of these reactions is a 1:1 ratio: 1 unit of Reagent A + 1 unit of Reagent B = 1 unit of Product C. However, the time required is not constant. For example, producing Hydroxide takes 20 ticks, while producing Catalyzed Ghodium Acid takes 80 ticks. The Science Overlord must account for these time deltas to prevent the lab complex from idling.

Compound Tier	Example Formula	Reaction Time (Ticks)	Key Utility
Base	Z + K \rightarrow ZK	5	Precursor only
Tier 1	G + H \rightarrow GH	10	Moderate WORK boost
Tier 2	GH + OH \rightarrow GH_{2}O	15	Significant WORK boost
Tier 3	GH_{2}O + X \rightarrow XGH_{2}O	80	Maximum WORK boost

The recursive logic must also check the current Terminal and Storage inventory. If the bot already possesses 500 units of GH_{2}O, it should skip the precursor reactions and move directly to the final tier with Catalyst (X). This dynamic order generation ensures that the lab complex always operates at maximum economic efficiency.

The Scientist Creep: State Machines and Inventory Logistics

The "Scientist" is a specialized logistics creep whose sole function is to move minerals between the Terminal, Storage, and Labs. Unlike legacy haulers that use simple "Pull" logic, the Scientist is a high-precision agent driven by the Science Overlord's current orders.

Finite State Machine Implementation

To ensure deterministic behavior and minimize CPU overhead, the Scientist operates on a Finite State Machine (FSM). This avoids the "re-deciding" problem where a creep recalculates its target every tick. The core states include:

1. **RENEW:** Moving to a spawn to increase ticksToLive before starting a high-value transfer, preventing the creep from dying while carrying 3,000 units of expensive minerals.
2. **REAGENT_SUPPLY:** Withdrawing the required quantities of Ingredient_{1} and Ingredient_{2} from the Terminal and depositing them into the designated input labs.
3. **PRODUCT_COLLECTION:** Withdrawing finished minerals from the output labs once they reach a threshold (e.g., 100 units) and returning them to the Terminal.
4. **CLEANUP (FLUSHING):** The most critical state. If an order changes or a lab contains the wrong mineral type, the Scientist must purge the labs to prevent a "Lab Jam".

The Lab Jam and Purging Protocols

A "Lab Jam" occurs when a lab contains a residual amount of a mineral that is not part of the current reaction order, preventing the input of new reagents. This often happens after a global reset or a change in high-level strategy. The Scientist must be programmed with a "Purge" protocol that scans all 10 labs against the current LabOrder. If a lab's store contains an unauthorized resource, the Scientist immediately prioritizes emptying that lab into the Terminal. Failure to implement this cleanup logic is a primary reason why lower-tier lab automation systems fail over long durations.

Empire-Wide Mineral Logistics: The Terminal Network

In the endgame, no room is an island. Minerals are distributed unevenly across the world map; for instance, Zynthium (Z) and Keanium (K) are found in different quadrants than Utrium (U) and Lemergium (L). To sustain a continuous T3 production pipeline, an empire must establish a robust Terminal logistics network.

The Quota and Delta System

The industry-standard for inter-room logistics is the Quota System. Each room's OfficeMemory defines a target quantity for every resource type. The Logistics Overlord then calculates the "Delta" for each room.
A positive delta indicates a surplus, while a negative delta indicates a deficit. The system then matches "Providers" (Surplus > 0) with "Requesters" (Deficit < 0) using a stable matching algorithm that prioritizes the shortest linear distance between rooms to minimize the energy cost of the Terminal.send() command. This ensures that catalysts (X) and reagents are distributed across the empire where they are most needed.

Terminal vs. Storage Concentration

There is an architectural debate regarding where to store minerals. Storing minerals in the StructureStorage allows for massive capacity (up to 1,000,000 units), but transferring them to the Terminal for shipping requires multiple Scientist ticks. High-performance bots often concentrate minerals in the Terminal until it reaches a specific threshold (e.g., 50,000 units) before moving the overflow to Storage. This "Terminal-First" approach ensures that resources are always ready for immediate inter-room transfer or market sale.

Strategic Boosting: The Request Protocol and Body Optimization

The ultimate objective of laboratory automation is the application of boosts. A boost is a permanent modification that enhances a specific body part type for the duration of the creep's 1,500-tick life.

The Boost Request Protocol

Legacy bots often "Push" boosts to creeps, which is brittle and inefficient. Modern bots use a "Request" protocol. When a creep is spawned, its logic identifies its mission requirements (e.g., "I am a siege attacker, I need TOUGH and ATTACK boosts"). It then enters a BOOSTING state and registers a request with the room's Science Overlord.
The Science Overlord then reserves the necessary minerals and energy in the designated labs. The creep moves to the lab and calls StructureLab.boostCreep(creep). Each boost application costs 30 mineral units and 20 energy units per body part. This request-based system allows the Science Overlord to prioritize boosts for high-priority units, such as defenders, over routine production.

Body Part Sequencing and Effective Health

The efficacy of boosts is non-linear when combined with strategic body part sequencing. Damage in Screeps is applied to body parts in the order they appear in the array. Therefore, "Tough" parts should always be placed at the front of the body. When boosted with Catalyzed Ghodium Oxide (XGHO_{2}), these parts take 70% less damage.
This effectively triples the health of the creep's front-line "Armor" parts. Furthermore, since empty CARRY parts and boosted MOVE parts generate less fatigue, a bot can design "Capacious" haulers that move at full speed with significantly fewer MOVE parts, freeing up energy for more productive work.

Boost Type	Compound	Efficiency Multiplier	Primary Application
Upgrade	XGH_{2}O	+100% (No extra energy)	Power-leveling GCL
Heal	XLHO_{2}	+300% Effectiveness	Quad formation survival
Tough	XGHO_{2}	-70% Damage Taken	Siege tanks and defenders
Attack	XUH_{2}O	+300% Effectiveness	Dismantling enemy ramparts
Move	XZHO_{2}	+300% Fatigue reduction	Rapid response and logistics

Military Architecture: Quads, Tower Draining, and Siege Logic

In high-level play, an unboosted creep is effectively a liability. Combat is won by the side with the more efficient chemical supply chain.

The Quad Formation

The "Quad" is a specialized squad of four T3-boosted creeps moving in a 2x2 square. These creeps coordinate their actions to act as a single unit. They use "Cross-Healing," where each creep heals the squad member with the lowest HP. To sustain a Quad, the laboratory system must be capable of providing 1,200 units of various T3 compounds (30 units x 10 parts x 4 creeps) in a single burst. A failure in the lab automation pipeline (e.g., missing a MOVE boost

for just one creep) will break the quad's formation, leading to immediate tactical failure.

Tower Draining and Threat Analysis

A common defensive tactic is to use "Tower Draining". An attacker sends a T3-boosted TOUGH/HEAL creep to stand in range of an enemy's towers, forcing them to spend energy. A modern defensive bot must be able to perform "Threat Potential" analysis. If the incoming creep has T3 HEAL parts that out-heal the tower's damage, the tower should stop firing to conserve energy for a more viable target. This highlights the need for a Lab system that can pivot production from "Economy" boosts to "Combat" boosts in a single tick when an invasion is detected.

Economic Engineering: Market Arbitrage and Credit Liquidity

The laboratory system is not only a military asset but a primary driver of the colony's credit balance. A bot that can refine raw minerals into T3 compounds can realize massive profits on the public market.

The "Make vs. Buy" Decision Matrix

A top-tier bot must constantly evaluate the market price of minerals versus their production cost. If the price of XGH_{2}O is less than the combined price of G, H, and X plus the CPU cost of refining, the bot should simply buy the finished product.
Modern bots implement "Automated Trading" logic that places "Buy" orders at low prices for rare minerals and "Sell" orders for surplus local minerals. This ensures that the colony's credits are always being reinvested into the chemical pipeline, maintaining a high level of "Credit Liquidity."

CPU Efficiency in Chemical Operations

The StructureLab.runReaction method and the boostCreep method each cost 0.2 CPU per intent. In a large empire with 20 rooms, running 160 reactions per tick (8 per room) would consume 32 CPU—nearly a third of the standard limit. To optimize this, the Science Overlord must:

1. **Batch Reactions:** Only run reactions in rooms where the product is actually needed or when the bucket is full.
2. **Heap Caching:** Store all lab object references in the global heap to avoid expensive find calls.
3. **Order Prioritization:** Halt T1 precursor reactions if the T3 final product quota is already met, saving both CPU and energy.

Migration Roadmap for grgisme/screeps

To modernize the legacy grgisme repository for top-tier lab performance, the following roadmap is recommended:

Phase 1: Spatial Automation (RCL 6)

Abandon manual lab placement. Implement the Distance Transform planner to identify a 5x5 area for the lab cluster. Integrate a basic 3-lab "Starter" reaction loop to begin producing Ghodium for Safe Modes.

Phase 2: The Science Overlord (RCL 7)

Implement the Kernel/Process model. Transition the Scientist from a simple "Role" to an FSM-driven process managed by the Science Overlord. Introduce the recursive dependency resolver to handle the multi-tiered reaction graph.

Phase 3: The Global Mineral Network (RCL 8)

Establish the Quota/Delta system for Terminal-to-Terminal transfers. Implement the Request Protocol for boosting, allowing combat creeps to "Order" T3 boosts during their spawning phase. Integrate automated market trading to fill gaps in the catalyst (X) supply.

Conclusion: Chemical Supremacy as the Ultimate Benchmark

The implementation of a top-tier laboratory automation and boosting system represents the pinnacle of Screeps engineering. It is the point where a bot transcends simple strategy and enters the realm of complex systems management. By adopting a process-driven architecture, optimizing spatial layouts through automated planning, and integrating empire-wide mineral logistics, a developer can achieve a level of tactical dominance that is unreachable through standard unboosted play.

The transition from the legacy, monolithic loops of 2017 to the high-efficiency, chemical-hegemony models of 2025 is not merely an upgrade; it is a fundamental shift in how the Screeps environment is perceived. The bot is no longer a collection of scripts, but a distributed industrial complex, converting raw matter into the refined catalysts of empire expansion. In the endgame of Screeps, those who control the labs control the world.

Final Metrics and Strategic Comparison Table

Metric	Legacy (grgisme Era)	Modern (Top-Tier OS)	Impact on Competitive Play
Lab Management	Manual / Hard-coded	Science Overlord / Process	100% uptime of reactions
Reaction Logic	Single-tier / Simple	Recursive / Hierarchical	Access to T3 compounds
Scientist Logic	Static Pull-based	FSM with Purge Protocols	Zero "Lab Jams" or downtime
Logistics	Room-local only	Empire-wide Quota System	Unlimited mineral accessibility
Boosting	"Push" (Hard to	"Request"	Precision-enhanced

Metric	Legacy (grgisme Era)	Modern (Top-Tier OS)	Impact on Competitive Play
	manage)	(Just-in-Time)	units
Combat Power	Unboosted / Weak	T3 Quad-form / 4x Stats	Tactical invincibility
CPU Efficiency	Low (Redundant searches)	High (Heap caching)	Scaling to 50+ rooms

The path forward for the user is the systematic re-engineering of the science pipeline, treating every mineral unit as a potential multiplier for empire-wide success. By following the best practices outlined in this report—specifically the use of FSM-driven scientists, recursive dependency solvers, and inter-room terminal balancing—the colony will achieve the chemical supremacy necessary to dominate the Screeps endgame..

Works cited

1. Resources | Screeps Documentation, https://docs.screeps.com/resources.html 2. Screeps #22: For Science | Field Journal, https://jonwinsley.com/notes/screeps-for-science#terminal-logistics 3. Lab runReaction() CPU Too High | Screeps Forum, https://screeps.com/forum/topic/1149/lab-runreaction-cpu-too-high 4. Base Building Considerations : r/screeps - Reddit, https://www.reddit.com/r/screeps/comments/87ed6x/base_building_considerations/ 5. Automating Base Planning in Screeps – A Step-by-Step Guide, https://sy-harabi.github.io/Automating-base-planning-in-screeps/ 6. Screeps #1: Overlord overload - Ben Bartlett, https://bencbartlett.com/blog/screeps-1-overlord-overload/ 7. Screeps after one year - Pedantic Orderliness, https://www.pedanticorderliness.com/posts/screeps 8. Great Filters - Screeps Wiki, https://wiki.screepspl.us/Great_Filters/ 9. Screeps #14: Decision Making - Field Journal, https://jonwinsley.com/notes/screeps-decision-making 10. Strange Reactions constant | Screeps Forum, https://screeps.com/forum/topic/1065/strange-reactions-constant 11. Creep Boost discussion | Screeps Forum, https://screeps.com/forum/topic/471/creep-boost-discussion 12. Screeping | World Domination via JavaScript, https://screeping.wordpress.com/ 13. Safety and Design Handbook.pdf - OSU Chemistry, https://chemistry.osu.edu/sites/chemistry.osu.edu/files/Safety%20and%20Design%20Handbook.pdf 14. P U R I F I C A T I O N L A B O R A T O R Y C H E M I C A L S, https://rexresearch1.com/LabDesignSafetyTechnique/Purification%20Laboratory%20Chemicals PERRIN.pdf 15. Creeps | Screeps Documentation, https://docs.screeps.com/creeps.html 16. Help me! | Screeps Forum, https://screeps.com/forum/topic/2524/help-me/21?lang=en-GB&page=1 17. Using Modern A.I. Paradigms To Improve Paradox A.I. | Page 3 | Paradox Interactive Forums, https://forum.paradoxplaza.com/forum/threads/using-modern-a-i-paradigms-to-improve-paradox-a-i.1142483/page-3 18. Creep Body Setup Strategies - Screeps Wiki, https://wiki.screepspl.us/Creep_body_setup_strategies/ 19. [Power] Power creep ideas | Screeps Forum, https://screeps.com/forum/topic/388/power-power-creep-ideas 20. Workflow tips and prioritization for new players? | Screeps Forum, https://screeps.com/forum/topic/2556/workflow-tips-and-prioritization-for-new-players 21. Tips - Screeps Wiki - Fandom, https://screeps.fandom.com/wiki/Tips

Engineering the High-Tier Economy: Automated Link and Terminal Management in Screeps Architecture

The transition from a developing colony to a high-tier economic powerhouse in Screeps is defined by the shift from labor-intensive logistics to infrastructure-driven resource distribution. At Room Controller Level (RCL) 1 through 4, the economy relies on "Type Zero" harvesting, where mobile units (creeps) physically transport energy from sources to storage containers. However, upon reaching RCL 5 and 6, the introduction of the StructureLink and StructureTerminal allows for a fundamental decoupling of resource throughput from creep movement. This report provides an exhaustive engineering analysis of automated link and terminal management, establishing best practices for link balancing logic, market API automation, and empire-wide resource synchronization.

The Physicality of Instantaneous Transfer: StructureLink Mechanics

The StructureLink represents the first tier of high-efficiency logistics, enabling the instantaneous transfer of energy across room terrain without the latency associated with pathfinding. While links eliminate the need for courier creeps, they are governed by rigid physical and mathematical constraints that must be accounted for in any automated management script.

Spatial Constraints and Throughput Mathematics

A link pair's efficiency is inversely proportional to the linear distance between the sender and the receiver. Every transfer triggers a cooldown period calculated in game ticks, equivalent to the range between the two structures. This cooldown dictates the maximum theoretical bandwidth of a link network. For example, a link pair separated by 50 tiles has a maximum throughput of 800 energy units every 50 ticks, or 16 energy units per tick. Given that a standard energy source produces 10 units per tick (or 3000 units every 300 ticks), a single link can comfortably handle the output of one or even two sources provided they are within reasonable proximity to the central hub.

Furthermore, links incur a mandatory energy loss, often referred to as a "transmission tax." This loss is calculated as:

where A represents the amount of energy transferred. Despite this 3% loss, links remain significantly more efficient than creep-based hauling when accounting for the energy cost of spawning couriers, the decay of road infrastructure, and the CPU overhead of pathfinding calculations.

Link Role Categorization and Identification

A top-tier automated system does not treat all links as generic containers. Instead, it employs a role-based identification system, typically executed during the room's initialization or upon an

RCL increase.

Link Role	Description	Geographic Placement	Priority
Source Link	Acts as a primary producer; collects energy from local harvesters.	Adjacent to Energy Sources.	Low (Sender)
Hub Link	The central node of the network; typically adjacent to the room's Storage.	Within Range 1 of StructureStorage.	Medium (Buffer)
Controller Link	Supplies energy to static upgrader creeps.	Within Range 3 of the Room Controller.	High (Receiver)
Tower Link	Ensures towers remain functional for defense without dedicated fillers.	Adjacent to defensive clusters.	Critical (Receiver)
Extension Link	Optimizes the filling of distant extension clusters during spawn bursts.	Central to an extension farm.	Medium (Receiver)
Remote Link	Receives energy from long-distance haulers at the room's edge.	Near room exits/entrances.	Low (Sender)

The identification process leverages Room.find(FIND_MY_STRUCTURES) filtered by STRUCTURE_LINK, followed by proximity checks against key room objects like the Storage or Controller. Once identified, these links are stored in the room's global memory to avoid expensive re-calculation in subsequent ticks.

Algorithmic Patterns for Link Balancing Logic

The goal of automated link management is to ensure that energy is always available at high-priority "Sinks" (e.g., the Controller Link) while preventing "Sources" (e.g., Source Links) from reaching capacity and halting production.

The Hub-and-Spoke Priority System

The most robust architectural pattern for link management is the Hub-and-Spoke model. In this configuration, the Hub Link (adjacent to Storage) acts as the primary distributor. The logic flow is segmented into two distinct phases: Collection and Distribution.
During the Collection phase, all Source and Remote links evaluate their internal energy levels. When a sender reaches a specific threshold (typically 700-800 energy), it initiates a transfer to the Hub Link. This "push" logic ensures that harvesting sites remain clear for incoming energy.
In the Distribution phase, the Hub Link scans for available Sinks based on a predefined priority queue.

1. **Tower Links:** If a Tower Link's energy falls below 400, the Hub Link prioritizes a transfer to ensure defensive readiness.
2. *Controller Link:* If the Controller Link falls below a threshold (e.g., 200 energy), the Hub

Link transfers energy to sustain upgrading.
3. *Extension Link:* During spawning cycles, if the Hub Link has surplus energy and the Extension Link is empty, a transfer is executed to assist filler creeps.

The Balanced Network Approach

In complex base layouts where links may serve multiple purposes (e.g., a link that both receives from a remote source and supplies a nearby tower), a "Balancing" algorithm is preferred. This method calculates the average energy across all links in the room and attempts to equalize them. While mathematically elegant, it is less energy-efficient than priority-based systems due to the 3% transfer tax; moving energy multiple times to achieve "balance" results in higher cumulative loss. Therefore, balancing should only be applied to multi-use nodes while dedicated source and sink links remain within a priority hierarchy.

Terminal Logistics: Global Trade and Empire-Wide Synchronization

Upon reaching RCL 6, the introduction of the StructureTerminal expands the economy from a local room-scale to a global world-scale. Terminals allow for the instantaneous transfer of any resource between any two rooms on the map, provided both possess a terminal.

The Leaky Bucket Throughput Model

A critical recent development in terminal mechanics is the "Leaky Bucket" throughput limit. Incoming transactions are restricted to a specific volume per tick—preliminary values suggest 50 units per tick—which can accumulate up to a burst capacity of 300,000 units. Automated logistics systems must track this "available throughput" to prevent deal failures during high-intensity operations like GCL farming or "room burst upgrading," where multiple rooms might attempt to send energy to a single destination simultaneously.

Transaction Cost Calculus and Distance Optimization

Transferring resources via terminal is not free; it consumes energy based on the amount of resources and the linear distance (d) between the origin and destination. The energy cost is defined by the following non-linear relationship:
This formula implies that as distance increases, the energy cost approaches the total amount of resources being sent. At d = 30, the cost is approximately 63% of the amount; at d = 60, it reaches 86%. Consequently, an automated terminal governor must prioritize local transfers over distant ones. Empire-wide resource balancing logic should always seek to satisfy a room's deficit using the closest available surplus to minimize the energy overhead of the trade network.

Market API Automation: Growing the Credit Balance

A top-tier economy utilizes the Screeps Market to convert surplus minerals and commodities into Credits, which can then be used to purchase lacking resources or high-tier boosts.

Price History and Trend Analysis

Automation scripts should leverage the Game.market.getHistory(resourceType) method to retrieve 14-day price data. This data allows for the implementation of advanced trading indicators such as Simple Moving Averages (SMA) or Exponential Moving Averages (EMA) to identify "fair" market value.

Market Strategy	Description	Implementation Step
Mean Reversion	Buying assets when they are significantly below their historical average price.	Monitor getHistory for prices < SMA(14) * 0.9.
Momentum Trading	Capitalizing on upward price trends for commodities like Power or Ghodium.	Identify resources where current price > EMA(14).
NPC Arbitrage	Selling processed commodities to NPC terminals at highway crossroads.	Track NPC buy prices for "Bars" (e.g., Utrium Bar).

Order Management and Fee Optimization

Participating in the market requires a balance between "Deals" and "Orders". Using Game.market.deal() is immediate but requires the player to pay the energy cost and the terminal cooldown. Creating an order using Game.market.createOrder() avoids the energy cost but incurs a 5% credit fee and is subject to market competition.
A sophisticated Market Manager process should follow these best practices:

- **Order Throttling:** Limit the number of active orders to stay within the 300-order limit per shard.
- **Price Adjustment:** Periodically call Game.market.changeOrderPrice() to remain competitive, but be wary of the 5% tax on price increases for buy orders.
- **Volume Extension:** Use Game.mark[span_48](start_span)[span_48](end_span)et.extendOrder() to replenish successful sell orders rather than creating new ones, which maintains the order's presence without resetting its aging.

High-Tier Economic Structures: Factories and Commodities

At RCL 7 and 8, the economy moves beyond raw minerals into the production of commodities. Factories process base minerals and energy into highly compressed goods (e.g., Batteries, Bars, and complex commodities like Microchips).

The Value-Added Chain

Commodities serve two primary purposes: storage compression and credit generation. Processing 500 units of energy into 50 Batteries reduces the storage footprint significantly while providing a portable form of energy for terminal transfers. More importantly, high-level commodities are the only resources NPC traders consistently purchase at high prices, making them the primary engine for credit growth in the endgame.

Product	Ingredients	Factory Level	NPC Interest
Energy Battery	600 Energy	Any	Low (Compression only)
Utrium Bar	500 Utrium + 200 Energy	Any	High (Standard sink)
Cell	500 Biomass + 200 Energy	Any	High (Quadrant specific)
Tier 5 Commodity	Complex Reagents	Level 5	Critical (Max Credits)

Automated factory management requires tight integration with terminal logistics. The "Factory Overlord" must request reagents from the empire-wide network and ensure that finished products are either stored for future use or sent to highway rooms for sale to NPC terminals.

Software Architecture: The Economic Kernel

To manage links, terminals, and the market efficiently across multiple rooms, a process-based Operating System (OS) architecture is essential. This approach prevents the "starvation" of critical logic and ensures that CPU usage remains within the limits provided by GCL levels.

Process Decomposition and Priority Scheduling

The economic system should be broken down into discrete processes, each managed by a scheduler that allocates CPU based on urgency.

1. **Link Process (Priority: Critical):** Executed every tick. Handles local energy transfers to towers and controllers to ensure room survival.
2. **Logistics Process (Priority: High):** Executed every 5-10 ticks. Manages the movement of energy between Storage and Terminal within a room.
3. **Terminal Process (Priority: Medium):** Executed every 20 ticks. Synchronizes resource levels across the empire and fulfills internal requests.
4. **Market Process (Priority: Low):** Executed every 100 ticks. Scans getAllOrders, updates prices, and executes profitable trades.

By decoupling these tasks, the bot can skip the Market Process if the CPU bucket is low without risking the room's defense (managed by the Link Process).

The Terminal Overlord Pattern

In a multi-room colony, a central "Terminal Overlord" process maintains a global view of all assets. It calculates an "Empire Average" for each resource and identifies rooms with significant deviations. If Room A has 800,000 energy (surplus) and Room B has 20,000 (deficit), the Overlord calculates the transaction cost and issues a send command. This centralized management prevents "ping-ponging" where two rooms repeatedly send resources back and forth due to conflicting local logic.

Advanced Optimization: Power Creeps and Storage Expansion

The ultimate evolution of automated management involves the deployment of Power Creeps

(PCs). These hero units possess unique abilities that drastically enhance the efficiency of links and terminals.

- **OPERATE_LINK:** Decreases link cooldown and increases energy transfer efficiency.
- **OPERATE_TERMINAL:** Decreases the energy cost of terminal transfers and reduces the cooldown.
- **OPERATE_STORAGE:** Temporarily increases the storage capacity by up to 7 million units, allowing a room to act as a massive economic buffer during market fluctuations.
- **REGEN_SOURCE:** Increases the energy output of a source, necessitating higher-throughput link balancing logic to prevent overflow.

An automated system must coordinate PC movement to ensure these buffs are applied precisely when needed—for instance, activating OPERATE_TERMINAL just before a large-scale empire-wide redistribution of energy.

Risk Mitigation and Resource Locking

High-tier automation faces risks such as "Terminal Congestion" and "Credit Exhaustion".

Terminal Congestion and Deadlocks

A terminal can become "deadlocked" if it is filled with minerals but lacks the energy required to pay the transfer fee to send them away. To prevent this, best practices dictate the use of a "Minimum Energy Reserve." The terminal should always lock 20,000 to 50,000 units of energy that cannot be consumed by factories or the market, ensuring the logistics network remains operational.

Market Volatility and Safety Buffers

Automated trading algorithms must implement "Stop-Loss" and "Slippage" protections. Before executing a Game.market.deal(), the script should compare the current price against the 14-day history. If the price deviates by more than 2 standard deviations (a "Z-Score" check), the trade should be aborted as it likely represents a momentary market anomaly or a predatory order.

Conclusion: Implementing the Top-Tier Economy

Achieving a top-tier economy in Screeps requires the seamless integration of spatial planning, mathematical optimization, and algorithmic trading. By transitioning to a role-based link priority system, players eliminate the CPU waste of mobile haulers and ensure that room-critical structures are always powered. Moving to a terminal-centric empire allows for the efficient concentration of resources, enabling the production of high-value commodities and the steady growth of a credit balance. The success of this implementation is measured not by the amount of energy harvested, but by the "Credit Velocity" and the stability of the colony under stress. As the bot evolves, the focus shifts from managing creeps to managing flows, turning the empire into a self-sustaining autonomous machine capable of dominating the global market.

Works cited

1. Control | Screeps Documentation, https://docs.screeps.com/control.html 2. Great Filters -

Screeps Wiki, https://wiki.screepspl.us/Great_Filters/ 3. Link | Screeps Wiki - Fandom, https://screeps.fandom.com/wiki/Link 4. StructureLink - Screeps Wiki, https://wiki.screepspl.us/StructureLink/ 5. Balance of Links | Screeps Forum, https://screeps.com/forum/topic/1744/balance-of-links 6. Introduce an upper limit for terminal transfers | Screeps Forum, https://screeps.com/forum/topic/935/introduce-an-upper-limit-for-terminal-transfers 7. Operating System - Screeps Wiki, https://wiki.screepspl.us/Operating_System/ 8. Concept Question: How to effectively use a Link and distribute Energy in the room : r/screeps, https://www.reddit.com/r/screeps/comments/4sy6f6/concept_question_how_to_effectively_use_a_link/ 9. Market System | Screeps Documentation, https://docs.screeps.com/market.html 10. Discussion: long-range logistics revamp | Screeps Forum, https://screeps.com/forum/topic/2975/discussion-long-range-logistics-revamp 11. Power | Screeps Documentation, https://docs.screeps.com/power.html 12. Market - Screeps Wiki, https://wiki.screepspl.us/Market/ 13. Intermediate-level tips - Screeps Wiki, https://wiki.screepspl.us/Intermediate-level_tips/ 14. How do you manage chemistry? | Screeps Forum, https://screeps.com/forum/topic/1895/how-do-you-manage-chemistry 15. Energy - Screeps Wiki, https://wiki.screepspl.us/Energy/ 16. Trading Algorithms: A Complete Step-By-Step Guide | Intrinio, https://intrinio.com/blog/how-to-create-a-trading-algorithm-essential-steps-for-success 17. Sma — Indikator dan Strategi - TradingView, https://id.tradingview.com/scripts/sma/ 18. Strona 7 | Wykładnicza średnia krocząca (EMA) — Wskaźniki i Strategie - TradingView, https://pl.tradingview.com/scripts/ema/page-7/ 19. Resources | Screeps Documentation, https://docs.screeps.com/resources.html 20. Draft: factories and commodities (new crafting/trading mechanic) | Screeps Forum, https://screeps.com/forum/topic/2557/draft-factories-and-commodities-new-crafting-trading-mechanic 21. Screeps #22: For Science | Field Journal, https://jonwinsley.com/notes/screeps-for-science 22. Screeps #1: Overlord overload - Ben Bartlett, https://bencbartlett.com/blog/screeps-1-overlord-overload/ 23. Screeps #26: World - Discrete Objectives | Field Journal, https://jonwinsley.com/notes/screeps-discrete-objectives 24. Automated Market Makers (AMMs): Math, Risks & Solidity Code | Speedrun Ethereum, https://speedrunethereum.com/guides/automated-market-makers-math 25. A detailed guide on Automated Market Maker AMM | by Rapid Innovation | Medium, https://medium.com/@rapidinnovation/a-detailed-guide-on-automated-market-maker-amm-a70cd80736ea

Comprehensive Architectural Analysis of Console Observability and Logging Optimization in Screeps

The architecture of an automated system in Screeps demands an observability layer that is both computationally inexpensive and semantically rich. In an environment where the player's primary interaction with the world occurs through the execution of sandboxed JavaScript, the console is the fundamental telemetry interface. However, the standard implementation of console logging often presents significant challenges, ranging from performance degradation and heap memory pressure to interface-specific rendering inconsistencies. This report provides a definitive research synthesis on the best practices for architecting a logging system within the Screeps environment, specifically addressing the mitigation of tick-by-tick spam, the technical nuances of HTML and CSS formatting in the console, and the optimization of server-side CPU utilization through strategic telemetry design.

The Computational Mechanics of the Screeps Logging Pipeline

To establish a robust logging framework, it is first necessary to analyze the causal chain of data as it moves from the server-side execution environment to the client-side user interface. Screeps utilizes the isolated-vm library to run user code in a highly restricted, secure sandbox.[1] Every call to console.log() triggers a mechanism that captures the output during the script's execution phase in the game loop.[2] This process is not merely a visual artifact; it represents a data transmission event that occurs within the constraints of the game's tick-based architecture.

The lifecycle of a log entry begins with the instantiation of a string in the script's heap. In the V8 engine, string concatenation and manipulation are memory-intensive operations. When a player constructs complex log messages using standard string interpolation or the + operator, the engine must allocate new memory segments for these temporary objects. Research indicates that frequent memory allocations within the sandbox contribute to increased garbage collection frequency.[3] For advanced players running scripts at high CPU limits—up to 300 ms per tick for Global Control Level (GCL) 30 players—the cumulative cost of string processing can consume a non-trivial portion of the available CPU bucket.[4]

Pipeline Stage	Resource Constraint	Primary Risk
String Construction	Server CPU / Heap Memory	Garbage Collection spikes

		and timeout risks
Serialization	Server CPU	Recursive complexity in JSON.stringify() calls
Transmission	WebSocket Bandwidth	Connection throttling and rate limiting
DOM Rendering	Client CPU / Memory	Interface lag and Chromium memory leaks

The data points to a consistent trend where excessive logging overhead shifts from a server-side concern to a client-side bottleneck as the complexity of the AI increases.[6] While a simple string log might cost less than 0.01 CPU on the server, the cumulative effect of transmitting thousands of characters to the browser client creates a heavy load on the Document Object Model (DOM).[7] The official documentation and community feedback emphasize that the web version of the game forwards all console output to the browser's developer tools.[9] This duplication, while useful for deep object inspection, significantly multiplies the rendering overhead, often causing the simulation mode to slow to a crawl.[7]

Strategic Throttling and Signal-to-Noise Ratio Optimization

The most frequent complaint among Screeps developers is the "spam" effect, where routine operations flood the console every tick, making it impossible to identify critical alerts. Effective logging must distinguish between "telemetry" (data meant for automated analysis) and "notifications" (data meant for human observation).

Modulo-Based Temporal Throttling

The standard mechanism for reducing log volume is the implementation of periodic logging using the Game.time property.[2] By applying the modulo operator, a script can ensure that status updates are emitted only once every N ticks. This effectively reduces the logging frequency from a per-tick basis to a manageable interval. However, a naive implementation where multiple systems all log on Game.time % 100 == 0 creates a synchronized CPU spike every 100 ticks.[11]

To achieve a smoother performance profile, the analysis suggests the use of staggered offsets.[12] By adding a unique identifier or a random salt to the Game.time before the modulo operation, the logging events are distributed across the cycle. For example, logging a room's

energy status on (Game.time + roomNameHash) % 100 == 0 ensures that different rooms report their status at different points in the 100-tick cycle, flattening the CPU load and maintaining a steady stream of information rather than periodic bursts.

Log Frequency	Strategic Intent	Implementation Pattern
Per Tick	Critical Errors / Attack Alerts	if (isCritical) { log(); }
10-20 Ticks	Operational State Transitions	if (stateChanged) { log(); }
100 Ticks	Resource Efficiency Stats	if (Game.time % 100 == 0) { log(); }
500-1000 Ticks	Long-term GCL/RCL Projections	if (Game.time % 500 == 0) { log(); }

State-Transition and Delta Logging

A more advanced strategy for telemetry is delta logging, where information is emitted only when a significant change in the game state occurs.[13] For instance, instead of logging the hits of a rampart every tick, the system should log only when the hits cross a certain threshold or when the rampart is actively being repaired by a tower. This approach requires maintaining a local cache of the "last logged state" within the Memory object or the heap.[15]

Causal relationships in the data suggest that delta logging is far more effective for complex bots managing multiple rooms.[17] As the empire expands, the sheer number of objects makes periodic logging of every unit unfeasible. By focusing on state changes—such as a creep transitioning from HARVESTING to UPGRADING—the developer can reconstruct the entire timeline of events from a fraction of the data required by tick-based logging.[18]

Technical Troubleshooting of HTML and CSS Formatting

The user has identified persistent issues with the rendering of <font> and tags, while noting that emojis function without fail. This discrepancy is rooted in the way different game clients parse and sanitize the HTML stream emitted by the console.log() command.[19]

The Role of HTML Sanitization and Security

Since the inception of the project, the Screeps admins have allowed console.log() to accept HTML for the purpose of creating prettier console outputs.[21] This historical decision led to the "Client Abuse" community, where players used <script> and <style> tags to inject custom functionality into the official clients.[21] To prevent security vulnerabilities such as cross-site scripting (XSS), the game client now utilizes a sanitization layer.[22]

Tags like and <font> are generally allowed, but their attributes are heavily filtered. The failure of tags usually occurs when the style attribute contains properties that the sanitizer deems unsafe, such as position: fixed or width: 100%, which could be used to obscure the main game UI.[25] Furthermore, renderers in different environments (such as the Steam client vs. a standard browser) have varying support for modern CSS properties.[27]

Formatting Method	Technical Support	Primary Limitation
<font color="...">	Universal (Web/Steam/Terminal)	Deprecated; limited to color/size
	High (Web/Modern Steam)	Strict sanitization of CSS properties
<b> / <i> / <u>	Universal	Limited semantic range
Emojis (Unicode)	Universal	OS-dependent font availability

Optimizing CSS Precedence and Syntax

If formatting tags are being ignored, the analysis points to three likely culprits:

1. **Improper Nesting and Quotation:** The Screeps console parser is often less forgiving than a standard web browser. Research into similar environments suggests that the lack of quotes around attribute values or the use of single quotes when double quotes are expected can cause the parser to fail.[29] The safest pattern is to use escaped double quotes for internal attributes: console.log("<span style=\"color:#ff0000\">...");.
2. **CSS Specificity and Overrides:** The game client's own stylesheet often has global rules for console text. If a player uses a tag, its properties may be overwritten by the client's default font settings unless the style is declared with sufficient specificity or defined inline.[31]
3. **The "Inherit" Pattern:** In some browser versions, the console environment does not

automatically inherit the parent's text properties for inline elements. Using all: inherit or color: inherit can sometimes fix transparency or layout issues in formatted logs.[34]

The Resilience of Unicode Iconography

The reason emojis work reliably is that they are not part of the HTML processing layer; they are standard Unicode characters rendered by the client's system fonts.[27] In a professional observability context, emojis should be used as "visual tags" that allow for rapid scanning of logs. Emojis bypass the overhead of HTML parsing and are interpreted by the text server as single glyphs, making them the most CPU-efficient way to add color and categorization to the console.[37]

Architectural Best Practices for a Logging Utility

Rather than invoking console.log() directly throughout the codebase, it is standard practice among high-level Screeps developers to implement a dedicated Logger class.[39] This abstraction layer provides several critical advantages for large-scale empire management.

Centralized Level Filtering

A robust logger implements severities (ERROR, WARN, INFO, DEBUG, TRACE).[41] This allows the user to change the global verbosity level through a single memory edit without redeploying code. During normal operation, the level is set to INFO, suppressing the "noise" of creep actions. During active debugging or defense, the level can be dropped to DEBUG to provide more granular visibility into the AI's logic.[41]

Level	Severity	Visual Signal	Operational Trigger
ERROR	5	🔴 Red / 💥	Script crash, spawn failure, invasion detected
WARN	4	🟡 Yellow / ⚠️	Energy shortage, construction stagnation
INFO	3	⚪ White / ℹ️	RCL/GCL upgrades, market transactions
DEBUG	2	🔵 Blue / 🔍	Decision-making logic, pathfinding

			costs
TRACE	1	🟣 Purple / 📝	Individual creep intents and task updates

Decoupling Logic from Output

The analysis of professional repositories like ScreepsDotNet and the Overmind bot reveals a pattern of decoupling the "what" from the "where".[39] A well-structured logger should be able to route output to different sinks. While the console is the primary sink, higher-order telemetry can be written to memory segments for ingestion by external tools like Grafana or Kibana via screeps-stats.[46] This allows the console to remain clean for the human operator while maintaining a complete historical record of the AI's performance in a specialized analytics environment.

Performance Through Minimal Evaluation

A significant source of "lost" CPU is the evaluation of log strings that are ultimately suppressed by the current log level. A professional logger should check the level *before* constructing the string.

JavaScript

```
// Optimized Pattern (Narrative Description)
// Instead of:
// log.debug("Creep " + name + " is at " + pos);
// Use:
// if (log.level <= DEBUG) { log.emit("Creep " + name + " is at " + pos); }
```

By wrapping the logging call in a level check, the script avoids the expensive concatenation and serialization of data that will never be displayed.[43] This is particularly critical for objects being serialized via JSON.stringify(), as the performance cost can be hundreds of times higher than a standard property lookup.[8]

Client-Specific Formatting Nuances

The Screeps ecosystem is served by multiple clients, each with distinct rendering engines. Best practices for logging must account for the common denominators and the specific advantages

of each platform.

The Official Web Client

The web client is based on standard Chromium components. It provides the best support for room links using the syntax W1N1.[50] These links are interactive elements that shift the game's focus to the specified room. The data suggests that making room names interactive in logs is one of the most effective ways to improve base management efficiency.[19]

The Steam Desktop Client

The Steam client is an Electron application that reuses the web code but operates in a different security context. Causal analysis of historical vulnerabilities shows that the Steam client previously lacked some of the sandboxing present in web browsers, leading to risks where maliciously crafted HTML in a player's sign could execute commands on a victim's machine.[21] Modern updates have mitigated this through stricter sanitization and updated Chromium versions.[27] However, the Steam client remains more prone to memory bloat from console spam than a standalone Chrome window.[10]

Terminal Clients (Multimeter and Screeps-Console)

For advanced users, terminal clients like screeps-multimeter provide a lightweight alternative to the graphical UI.[20] These clients do not render HTML; instead, they translate a subset of tags into ANSI terminal colors.[57] To ensure logs are readable in these tools:

- Use <font color="..."> for basic coloring, as most terminal wrappers are hardcoded to support it.[57]
- Avoid complex inline CSS in tags, as they will often be stripped out entirely, leaving plain text.[57]
- Ensure that the color hex codes are 6-character strings (e.g., #FF0000) rather than 3-character shorthand, which some basic terminal renderers fail to parse.[58]

Memory Management and Persistence in Telemetry

Logging in Screeps is deeply intertwined with the Memory object and the concept of "Global Resets." Understanding these relationships is critical for accurate troubleshooting.

The Impact of Global Resets

Global resets occur when the game engine migrates a player's script to a different server node or when the user uploads new code.[15] During a reset, all heap variables (those not stored in Memory) are wiped. A professional logging system must log the occurrence of a global reset to alert the operator that any non-persisted caches have been invalidated.[15] This is often the time

when "startup" logs are most critical, providing a snapshot of the AI's initialization parameters.[41]

Managing Memory Bloat

While it may be tempting to store a history of logs in the Memory object, this is highly discouraged. The Memory object must be serialized and deserialized every tick, and its CPU cost is directly proportional to its size.[3] Storing thousands of log strings in Memory.log will rapidly inflate the AI's baseline CPU usage, potentially leading to timeouts.[51] Instead, logs should be treated as ephemeral streaming data, while historical stats should be aggregated into numerical counters or stored in memory segments, which are only loaded on demand.[47]

Persitence Method	CPU Cost	Use Case
Console Stream	Minimal	Real-time human monitoring
Memory Storage	Very High	Critical state persistence (not logs)
Memory Segments	High (on load)	Historical analytics and stat dumps
Game.notify()	Minimal (Intent)	Out-of-game emergency alerts

Advanced Observability and Profiling Techniques

For the most complex Screeps repositories, such as the one at https://github.com/grgisme/screeps, simple logging is often insufficient. Advanced developers leverage profiling tools and error mapping to diagnose performance issues and logical failures.

Error Mapping and Transpilation

If the codebase utilizes TypeScript or a bundler like Webpack/Rollup, the standard stack traces in the console will point to a single compiled file (e.g., main.js), rendering line numbers useless.[49] Implementing an ErrorMapper utility is essential. These utilities use source maps to translate the compiled stack trace back to the original source files, allowing the console to output accurate file paths and line numbers.[49] This transformation is usually performed within a try-catch block at the entry point of the main loop.[19]

The Screeps Profiler

The community-developed screeps-profiler is the gold standard for performance logging.[12] It works by wrapping standard game prototypes (like Creep, Room, and Spawn) and measuring the CPU used by each method call.[12] The profiler then outputs a formatted table to the console, showing the average and total CPU usage for every function in the AI. This is the ultimate tool for identifying "invisible" CPU leaks that logging alone cannot catch.

Security Considerations and Safe Logging

Because console.log() parses HTML, it creates a potential vulnerability if the script logs untrusted data from other players.[21] This includes creep names, room signs, and public memory segments.

Sanitizing Untrusted Inputs

A malicious player could name a creep something like <script>localStorage.auth =...</script>. If a victim's script finds this creep and logs its name via console.log(creep.name), the script will execute in the victim's browser, potentially stealing their authentication token.[21]

To prevent this, any data originated from an external source must be sanitized or escaped.[26] Simple regex-based escaping of < and > characters is a baseline requirement for professional-grade bots.[72]

JavaScript

```
// Safe Logging Pattern (Narrative)
// let safeName = untrustedData.replace(/</g, "<").replace(/>/g, ">");
// log.emit("Encountered creep: " + safeName);
```

This prevents the browser from interpreting the logged data as HTML tags, effectively neutralizing any "Client Abuse" attempts while still displaying the relevant information.[26]

Conclusions and Practical Implementation Path

The evidence gathered through extensive research into Screeps console logging suggests that an optimal observability strategy is founded on three pillars: performance, clarity, and cross-client compatibility.

First, the logging system must be performance-aware. The use of a severity-based global logger ensures that expensive string manipulations are only performed when necessary.

Developers should leverage the browser's developer tools for deep object inspection rather than using JSON.stringify() for high-frequency logs.

Second, clarity is achieved through intelligent throttling and semantic formatting. Using Game.time with staggered offsets prevents simultaneous CPU spikes, while state-transition logging provides a high signal-to-noise ratio. Unicode iconography (emojis) should be prioritized for cross-client visual signals, as it bypasses the complexities of HTML sanitization and CSS overrides.

Third, the formatting logic must be client-agnostic. While tags with inline styles are the modern standard, ensuring basic compatibility with <font color="..."> is essential for those who use terminal-based third-party tools.

For the operator managing the repository at https://github.com/grgisme/screeps, the immediate priority should be the implementation of a centralized logging utility that wraps the entry points with an ErrorMapper and utilizes a severity filter. This will transform the console from a source of spam and client-side lag into a precision tool for tactical and strategic oversight. The transition from per-tick output to event-driven telemetry is the hallmark of a mature Screeps AI, allowing for continued expansion without compromising the stability of the observability layer. By adhering to these structural patterns, the operator can ensure that their automated empire remains both visible and efficient, even as it scales to manage dozens of rooms across multiple shards.

Works cited

1. Building a LeetCode-style code evaluator with isolated-vm - LogRocket Blog, accessed February 17, 2026, https://blog.logrocket.com/building-leetcode-style-code-evaluator-isolated-vm/
2. Understanding game loop, time and ticks - Screeps Documentation, accessed February 17, 2026, https://docs.screeps.com/game-loop.html
3. Need for Speed - Endgame optimization limitations | Screeps Forum, accessed February 17, 2026, https://screeps.com/forum/topic/298/need-for-speed-endgame-optimization-limitations
4. Screeps on Steam - cs.wisc.edu, accessed February 17, 2026, https://pages.cs.wisc.edu/~linzuo/deliverables/stage1/steam/steam785.html?l=norwegian
5. Control | Screeps Documentation, accessed February 17, 2026, https://docs.screeps.com/control.html
6. Does calling console.log use CPU? :: Screeps: World Help - Steam Community, accessed February 17, 2026, https://steamcommunity.com/app/464350/discussions/5/3516603387154196 46/?l=french
7. Simulator gets slower and slower | Screeps Forum, accessed February 17, 2026, https://screeps.com/forum/topic/2181/simulator-gets-slower-and-slower

8. Screeps: write debug output to console? - javascript - Stack Overflow, accessed February 17, 2026, https://stackoverflow.com/questions/27070203/screeps-write-debug-output-to-console
9. Debugging | Screeps Documentation, accessed February 17, 2026, https://docs.screeps.com/debugging.html
10. Which browser/plugin/embed does the steam client use? :: Screeps: World 综合讨论, accessed February 17, 2026, https://steamcommunity.com/app/464350/discussions/0/1470840994964782503/?l=schinese
11. New Player, Newb Coder, Dozens of Questions : r/screeps - Reddit, accessed February 17, 2026, https://www.reddit.com/r/screeps/comments/53uv4u/new_player_newb_coder_dozens_of_questions/
12. Need some help improving my CPU usage | Screeps Forum, accessed February 17, 2026, https://screeps.com/forum/topic/2381/need-some-help-improving-my-cpu-usage
13. Workflow tips and prioritization for new players? | Screeps Forum, accessed February 17, 2026, https://screeps.com/forum/topic/2556/workflow-tips-and-prioritization-for-new-players
14. How much logging is too much? (ASP.NET) : r/ExperiencedDevs - Reddit, accessed February 17, 2026, https://www.reddit.com/r/ExperiencedDevs/comments/1jt6o3u/how_much_logging_is_too_much_aspnet/
15. OOP Ideas for Screeps/JS, accessed February 17, 2026, https://screeps.com/forum/topic/2777/oop-ideas-for-screeps-js
16. My creep spawning code doesn't work | Screeps Forum, accessed February 17, 2026, https://screeps.com/forum/topic/2533/my-creep-spawning-code-doesn-t-work
17. tanjera/screeps: Scripts for my Screeps MMO colonies (JS; ongoing). - GitHub, accessed February 17, 2026, https://github.com/tanjera/screeps
18. Tutorial 1.1 : Creeps, Roles, Actions, Tasks, Jobs, and Priorities | Screeping, accessed February 17, 2026, https://screeping.wordpress.com/2020/06/22/tutorial-1-1-creeps-roles-actions-tasks-jobs-and-priorities/
19. Basic Debugging - Screeps Wiki, accessed February 17, 2026, https://wiki.screepspl.us/Basic_debugging/
20. screeps-multimeter/README.md at master - GitHub, accessed February 17, 2026, https://github.com/screepers/screeps-multimeter/blob/master/README.md
21. Client Abuse - Screeps Wiki, accessed February 17, 2026, https://wiki.screepspl.us/Client_Abuse/
22. Supported HTML tags in Source Code editor - PlayPosit Knowledge, accessed February 17, 2026,

https://knowledge.playposit.com/article/252-supported-html-tags-in-rte
23. Default TAGs ATTRIBUTEs allow list & blocklist · cure53/DOMPurify Wiki - GitHub, accessed February 17, 2026, https://github.com/cure53/DOMPurify/wiki/Default-TAGs-ATTRIBUTEs-allow-list-&-blocklist
24. To Configure Allowed HTML Tags and Attributes - Ivanti, accessed February 17, 2026, https://help.ivanti.com/ht/help/en_US/ISM/2025/admin-user/Content/Configure/SetUpWizard/Configure%20Allowed%20Tags%20and%20Attribute.htm
25. Content-Security-Policy: style-src directive - HTTP - MDN - Mozilla, accessed February 17, 2026, https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Headers/Content-Security-Policy/style-src
26. Why is a
27. Displaying Text with Say and Room.Visual [Fixed] :: Screeps: World General Discussions, accessed February 17, 2026, https://steamcommunity.com/app/464350/discussions/0/1473095965295570453/
28. How can I prevent having just one hanging word on a new line in an HTML element?, accessed February 17, 2026, https://stackoverflow.com/questions/31974448/how-can-i-prevent-having-just-one-hanging-word-on-a-new-line-in-an-html-element
29. Why aren't my tags working? - Codecademy, accessed February 17, 2026, https://www.codecademy.com/forum_questions/517b0506d2b8c1b78600180c
30. Top HTML CSS JAVASCRIPT Interview Questions & Answers 2026 - H2K Infosys, accessed February 17, 2026, https://www.h2kinfosys.com/blog/top-html-css-javascript-interview-questions-answers/
31. Using
32. span class id isnt working (Example) | Treehouse Community, accessed February 17, 2026, https://teamtreehouse.com/community/span-class-id-isnt-working
33. CSS span formatting - html - Stack Overflow, accessed February 17, 2026, https://stackoverflow.com/questions/21933968/css-span-formatting
34. Format console.log with color and variables surrounding non-formatted text - Stack Overflow, accessed February 17, 2026, https://stackoverflow.com/questions/41898612/format-console-log-with-color-and-variables-surrounding-non-formatted-text
35. Different font-size to span tag inside of p - Stack Overflow, accessed February 17, 2026, https://stackoverflow.com/questions/23338649/different-font-size-to-span-tag-inside-of-p
36. rhoit/dot-emacs: my emacs config :godmode - GitHub, accessed February 17, 2026, https://github.com/rhoit/dot-emacs
37. TextServer — Godot Engine (stable) documentation in English, accessed February 17, 2026, https://docs.godotengine.org/en/stable/classes/class_textserver.html
38. Game - Screeps Documentation, accessed February 17, 2026,

https://docs.screeps.com/api/
39. accessed December 31, 1969, https://github.com/overmind-screeps/Overmind/blob/master/src/console/Console.ts
40. accessed December 31, 1969, https://raw.githubusercontent.com/overmind-screeps/Overmind/master/src/console/Console.ts
41. When to use the different log levels - Stack Overflow, accessed February 17, 2026, https://stackoverflow.com/questions/2031163/when-to-use-the-different-log-levels
42. Log Debug vs. Info vs. Warn vs. Error: Understanding Log Levels - EdgeDelta, accessed February 17, 2026, https://edgedelta.com/company/blog/log-debug-vs-info-vs-warn-vs-error-and-fatal
43. Logging levels explained. Good old logs remain a very important... | by Michael Merkulov | Medium, accessed February 17, 2026, https://medium.com/@m.merkulov/logging-levels-explained-4dd61815601f
44. 10 Essential Logging Best Practices for Developers - Kloudfuse, accessed February 17, 2026, https://www.kloudfuse.com/blog/logging-best-practices-for-developers
45. thomasfn/ScreepsDotNet: Tools to support writing bots for Screeps Arena and Screeps World using .Net 8.0. - GitHub, accessed February 17, 2026, https://github.com/thomasfn/ScreepsDotNet
46. A client-side library to enhance the output of screeps-stats statistics. - GitHub, accessed February 17, 2026, https://github.com/screepers/screeps-stats-lib
47. Third Party Tools | Screeps Documentation, accessed February 17, 2026, https://docs.screeps.com/third-party.html
48. What tips do you have for logging? And by logging, I mean really good logging. - Reddit, accessed February 17, 2026, https://www.reddit.com/r/webdev/comments/tj1f26/what_tips_do_you_have_for_logging_and_by_logging/
49. shup1/xxscreeps - NPM, accessed February 17, 2026, https://www.npmjs.com/package/%40shup1%2Fxxscreeps
50. Steam client > Console link to room? - forum - Screeps, accessed February 17, 2026, https://screeps.com/forum/topic/763/steam-client-console-link-to-room
51. justjavac/screeps-tips: Tips of the day for Screeps Game. - GitHub, accessed February 17, 2026, https://github.com/justjavac/screeps-tips
52. Screeps: How A Game About Programming Sold Its Players a Remote Access Trojan, accessed February 17, 2026, https://outsidetheasylum.blog/screeps/
53. WebGL optimization | Screeps Forum, accessed February 17, 2026, https://screeps.com/forum/topic/2658/webgl-optimization
54. Open source alternative? · Issue #53 · screeps/screeps - GitHub, accessed February 17, 2026, https://github.com/screeps/screeps/issues/53
55. screepers/screeps-multimeter: The most useful tool on your screeps workbench.

- GitHub, accessed February 17, 2026, https://github.com/screepers/screeps-multimeter
56. screeps-multimeter - NPM, accessed February 17, 2026, https://www.npmjs.com/package/screeps-multimeter
57. screepers/screeps_console: Standalone Interactive Screeps Console - GitHub, accessed February 17, 2026, https://github.com/screepers/screeps_console
58. How to validate Hexadecimal Color Code using Regular Expression - GeeksforGeeks, accessed February 17, 2026, https://www.geeksforgeeks.org/dsa/how-to-validate-hexadecimal-color-code-using-regular-expression/
59. Colorizing text in the console with C++ - Stack Overflow, accessed February 17, 2026, https://stackoverflow.com/questions/4053837/colorizing-text-in-the-console-with-c
60. Category: Changelogs | Screeps Blog, accessed February 17, 2026, https://blog.screeps.com/categories/Changelogs/
61. Mysterious error without stack trace | Screeps Forum, accessed February 17, 2026, https://screeps.com/forum/topic/1718/mysterious-error-without-stack-trace
62. Making your own console functions - Screeps Wiki, accessed February 17, 2026, https://wiki.screepspl.us/Making_your_own_console_functions/
63. accessed December 31, 1969, https://raw.githubusercontent.com/grgisme/screeps/master/package.json
64. Auth Tokens - forum - Screeps, accessed February 17, 2026, https://screeps.com/forum/topic/2052/auth-tokens
65. Pushing code to Screeps, accessed February 17, 2026, https://wiki.screepspl.us/Pushing_code_to_Screeps/
66. accessed December 31, 1969, https://github.com/screepers/screeps-typescript-starter/blob/master/src/utils/ErrorMapper.ts
67. accessed December 31, 1969, https://raw.githubusercontent.com/screepers/screeps-typescript-starter/master/src/utils/ErrorMapper.ts
68. [SOLVED] Can't Operate Console When Experiencing Error | Screeps Forum, accessed February 17, 2026, https://screeps.com/forum/topic/119/solved-can-t-operate-console-when-experiencing-error
69. How do I know why a function is using that much CPU ? : r/screeps - Reddit, accessed February 17, 2026, https://www.reddit.com/r/screeps/comments/7rzczp/how_do_i_know_why_a_function_is_using_that_much/
70. How can I allow my user to insert HTML code, without risks? (not only technical risks), accessed February 17, 2026, https://stackoverflow.com/questions/701580/how-can-i-allow-my-user-to-insert-html-code-without-risks-not-only-technical

71. HTML_WHITELIST Function - Oracle Help Center, accessed February 17, 2026, https://docs.oracle.com/en/database/oracle/application-express/18.2/aeapi/HTML_WHITELIST-Function.html
72. screeps - GitHub Gist, accessed February 17, 2026, https://gist.github.com/derofim/e19a57bfff7f3ae20de8
73. White list of HTML tags I should allow from user generated content? - Stack Overflow, accessed February 17, 2026, https://stackoverflow.com/questions/9185495/white-list-of-html-tags-i-should-allow-from-user-generated-content

Orchestrating Autonomous Supply Chains: A Systems Engineering Report on the Global Logistics Broker in Screeps

The transition from localized, script-based automation to industrial-grade autonomous systems within the Screeps execution environment represents one of the most significant engineering challenges in contemporary distributed simulation. At its core, the Screeps environment is not merely a strategy game but a real-time, resource-constrained distributed operating system where the primary bottleneck is not the in-game currency (energy), but the computational throughput of the Central Processing Unit (CPU) and the overhead of memory serialization. This report provides an exhaustive architectural analysis of the design patterns required to implement a top-tier Global Logistics Broker, focusing on the shift from legacy "Search and Rescue" behaviors—typical of the grgisme/screeps era—to a modern "Command and Control" supply chain managed by stable matching algorithms and energy reservation ledgers.

The Physical Architecture of the Screeps Execution Environment

To engineer a high-performance logistics broker, one must first account for the physical constraints of the server-side infrastructure. User code in Screeps runs within a Node.js virtual machine isolate, often managed by isolated-vm to ensure security and resource accounting. This environment is strictly deterministic and operates on a discrete tick-based lifecycle.
The standard execution model involves two distinct stages: the player script calculation and the command processing phase. During the calculation stage, the server loads the user's script, parses the persistent Memory object, and executes the loop function. All commands issued during this time (e.g., creep.move(), creep.transfer()) are collected into a Redis-based queue for batch processing in the subsequent stage. This separation of intent and execution necessitates an architecture that can accurately predict the state of the world multiple ticks into the future.

The Heap Persistence and Isolate Lifecycle

A critical misunderstanding in early-stage development is the belief that the execution environment is completely stateless between ticks. In reality, the global scope—the V8 Heap—persists across ticks unless the server explicitly tears down the isolate to reclaim resources or the user uploads new code. Modern architectures exploit this persistence by caching massive amounts of data in the global object, bypassing the expensive JSON.parse() costs associated with the Memory object.

Feature	Legacy Approach (grgisme)	Modern Industrial Approach	Impact on Performance
State Storage	Primary reliance on	Primary reliance on	Reduces CPU spent on

Feature	Legacy Approach (grgisme)	Modern Industrial Approach	Impact on Performance
	Memory object.	Global Heap Cache.	JSON serialization.
Data Lifecycle	Data is re-parsed and re-processed every tick.	Persistent class instances updated via refresh().	Minimizes garbage collection and object instantiation.
Memory Limit	2MB hard cap on Memory.	2MB Memory + ~256MB+ Heap + 100MB Segments.	Enables massive historical data and complex cost matrices.
Sensing	Direct Room.find() calls per creep.	Centralized Registry with O(1) lookups.	Eliminates redundant spatial search overhead.

The Serialization Wall and the JSON Tax

The Memory object is not a live database connection but a JSON string stored in MongoDB and indexed via Redis. Each tick, the server charges the user the CPU cost of parsing this string. For an empire with multiple rooms and hundreds of creeps, a 2MB memory file can consume upwards of 15ms of CPU just to enter the loop function. This physical constraint dictates that a top-tier logistics broker must minimize the volume of data stored in Memory, favoring ephemeral heap-based structures for task management and only persisting critical "intent" state.

Forensic Analysis of Legacy Logistics Failures

In the grgisme era of Screeps development, logistics were largely emergent properties of independent agent scripts. A "harvester" role would independently seek a source, and a "carrier" role would independently scan for dropped energy or full containers. This "Search and Rescue" model fails at scale for three primary reasons: redundant sensing, collision-heavy movement, and the "Energy Racing" phenomenon.

The Complexity of Redundant Sensing

In a legacy bot, every hauler performs its own environmental assessment. If a room contains 20 haulers, and each hauler executes room.find(FIND_DROPPED_RESOURCES) to locate energy, the server must perform 20 identical spatial searches. This leads to an O(N \cdot M) complexity where N is the number of haulers and M is the number of objects in the room. In contrast, a modern broker performs a single scan, registering all providers and requesters in a centralized registry, reducing the cost to a manageable O(M) scan followed by O(N) matching.

The Energy Racing Phenomenon

Without a centralized broker, multiple haulers often target the same pile of energy or the same empty structure. This occurs because each hauler makes its decision based on the static state of the world at the start of the tick. If three haulers see a pile of 500 energy, all three may move toward it. Only the first to arrive will successfully pick up the resource; the other two will have wasted dozens of ticks in transit. This "race condition" is the primary cause of logistics inefficiency and can lead to catastrophic failure during combat when Towers run dry despite

ample energy being present in the room.

Monolithic Loop Fragility

Legacy bots typically utilize a monolithic loop that iterates through all creeps sequentially. This structure lacks preemption. If the logistics logic for the first ten haulers consumes the entire CPU budget—perhaps due to a complex pathfinding calculation—the remaining haulers and critical defense structures never execute their logic. The bot effectively "freezes," leading to empire collapse.

The Command-and-Control Paradigm: The Global Logistics Broker

A top-tier logistics system replaces decentralized "Roles" with a centralized Broker. This broker functions as an Inversion of Control (IoC) layer, where structures and resources do not "wait" to be found, but "register" their status as either Providers or Requesters.

Defining the Provider/Requester Pattern

In this architecture, every resource-holding or resource-consuming entity is categorized based on its potential to contribute to or drain from the supply chain.

1. **Providers:** These are entities with energy or resources that are "outputting" into the network. This includes sources (via miners), containers, links, dropped resources, and tombstones.
2. **Requesters:** These are entities that need resources to function. This includes Spawns and Extensions (for creep production), Towers (for defense), and the Controller (for upgrading).
3. **Buffers:** These are high-capacity structures like Storage or Terminals that can act as either providers or requesters depending on the current colony state.

The Transport Request Interface

To standardize communication between the broker and the game objects, a TransportRequest interface is established. This interface ensures that the broker has all necessary data to make an optimal matching decision without needing to query the game objects directly during the matching loop.

Property	Type	Purpose
Target	Object	The ID or reference of the structure or resource.
Amount	Number	The total volume of resources requested or provided.
ResourceType	Constant	The type of resource (e.g., RESOURCE_ENERGY).
Priority	Number	A weighted value indicating the urgency of the request.
Threshold	Number	The minimum amount required

Property	Type	Purpose
		to trigger a hauler dispatch.

By using this interface, the LogisticsNetwork can treat a request from a dropped pile of 1,000 energy and a request from a Tower with 10 energy as comparable data points, allowing for a unified mathematical approach to resource distribution.

Algorithmic Orchestration: Stable Matching with Gale-Shapley

The core of a top-tier logistics broker is the matching algorithm. While a simple greedy algorithm (matching the closest hauler to the highest priority request) is an improvement over legacy roles, it still fails to reach global optimality. The industry standard for high-performance bots is a variation of the **Gale-Shapley algorithm**, also known as the "Stable Marriage" algorithm.

Mathematical Foundations of Stable Matching

The Gale-Shapley algorithm finds a stable matching between two sets of elements (Haulers and Requests) given an ordering of preferences for each element. In the context of Screeps logistics, stability is defined as a state where no hauler and no request mutually prefer each other over their current assignment. If such a mutual preference existed, the hauler would "abandon" its current task to pursue the better one, leading to inefficiency.
The algorithm follows a "Propose and Reject" mechanism:

1. **Proposal:** Each "free" hauler (the proposer) proposes to its most preferred request that it has not yet approached.
2. **Tentative Matching:** Each request (the receiver) maintains its current "best" proposal. If it receives a proposal from a hauler it prefers over its current match, it "rejects" the old hauler and tentatively matches with the new one.
3. **Iteration:** Rejected haulers return to their preference list and propose to their next choice.
4. **Convergence:** The algorithm terminates when all haulers are matched or have exhausted their preference lists.

Heuristics for Preference Ranking

The success of the Gale-Shapley implementation depends on the heuristic used to generate preference lists. A typical ranking function for a hauler might look like:
Where:

- **Priority:** The urgency of the request (e.g., Towers = 10, Extensions = 5, Upgrading = 1).
- **Distance:** The pathfinding distance from the hauler's current position to the target.
- **ResourceDensity:** The ratio of the requested amount to the hauler's capacity, favoring targets that can fully utilize the hauler's move parts.

Because Gale-Shapley is "Proposer-Optimal," if the haulers are the ones proposing, every hauler will receive the best possible target according to its preference list, maximizing the total throughput of the colony.

The Energy Reservation System: Managing "Virtual"

Inventory

Matching is only half of the solution. To prevent "Energy Racing," the broker must implement a robust **Reservation System**. This system tracks resources that are "in flight" but have not yet arrived at their destination.

The Effective Store Calculation

A top-tier broker does not look at the structure.store property in isolation. Instead, it calculates the **Effective Store** :

- **Incoming Reservations:** The sum of energy currently being carried toward this structure by matched haulers.
- **Outgoing Reservations:** The sum of energy that has been promised to haulers currently moving toward this structure to pick up resources.

When the broker matches a hauler to a requester, it immediately adds a reservation to the ledger. If a Tower needs 200 energy and a hauler is dispatched with 200 energy, the broker's ledger will show the Tower's "Effective Store" as full, even though it will take the hauler 10 ticks to arrive. Any subsequent matching logic will see that the Tower's needs are met and will not dispatch another hauler.

Predictive Supply Chain Management

Advanced brokers extend this logic to account for future production. For instance, a static miner at a source produces 10 energy per tick. If a container currently has 500 energy, but a hauler is 20 ticks away, the broker can predict that the container will have 700 energy upon arrival. This "Predictive Amount" allows the broker to dispatch larger haulers or coordinate multiple pickups in a single trip, significantly reducing idle CPU time.

Reservation Type	Definition	Effect on Matching
Pickup Reservation	Hauler is moving to a Provider.	Reduces the "available" energy for other haulers.
Delivery Reservation	Hauler is moving to a Requester.	Increases the "effective" energy, preventing over-filling.
Production Prediction	Energy expected from Sources.	Dispatches haulers to "empty" containers that will be full soon.
Decay Prediction	Expected energy loss from dropped piles.	Prioritizes pickups that are about to vanish.

Hierarchical Coordination: The Overlord and Overseer Pattern

Modern bots like Overmind do not treat logistics as a standalone module but as a service utilized by high-level managers called **Overlords**. This hierarchy ensures that logistics are always aligned with the strategic goals of the empire.

The Colony and Hive Cluster Structure

A "Colony" represents the total presence in a room and its surrounding outposts. Within the

colony, specialized "Hive Clusters" manage specific areas (e.g., the Hatchery for spawning, the Command Center for storage). Each cluster identifies its own logistics needs and registers them with the LogisticsNetwork.
The TransportOverlord (or HaulingOverlord) is responsible for spawning and managing the hauler fleet. It queries the LogisticsNetwork every tick for the list of outstanding matches. If the number of requests exceeds the capacity of the current fleet, the TransportOverlord sends a request to the SpawnHatchery to produce more haulers.

Phase-Based Execution

To maintain stability and avoid race conditions, a top-tier broker executes in distinct phases within the tick.

1. **Build Phase:** Instantiate or refresh persistent classes from the Global Heap.
2. *Registry Phase:* All game objects (Towers, Spawns, Miners) register their TransportRequests.
3. **Matching Phase:** The broker runs the Gale-Shapley algorithm to pair free haulers with the most optimal requests.
4. **Run Phase:** Haulers execute their assigned tasks (moveTo, withdraw, transfer).
5. **Finalize Phase:** Persistent data is updated in the Global Cache, and critical state is saved to Memory.

This structured flow ensures that all "needs" are known before any "actions" are taken, allowing the broker to make decisions with a global view of the room's economy.

Navigation and Cartography: The Logistics of Movement

Movement is the single most CPU-intensive subsystem in Screeps. A hauler that recalculates its path every tick will consume more CPU than the rest of the bot combined.

Path Caching and Serialized Travel

A top-tier broker utilizes a "Path Cache" stored in the Global Heap. When a hauler is assigned a match, the broker calculates the path once using PathFinder.search(). This path is then compressed into a string (e.g., "33214" representing directions) and stored in the hauler's heap memory. On subsequent ticks, the hauler simply reads the next character from the string and moves in that direction, avoiding the O(N) overhead of the built-in pathfinder.

Traffic Management and "Shoving"

In a high-density base, haulers frequently encounter other creeps blocking their path. Standard moveTo() logic causes the creep to wait or recalculate. A modern logistics system implements a "Shove" algorithm. If a high-priority hauler (e.g., carrying energy to a Tower during an attack) is blocked by a lower-priority creep (e.g., an Upgrader), the hauler will "shove" the Upgrader to an adjacent tile. This ensures that the supply chain is never physically interrupted by civilian traffic.

Link Balancing and Instant Transfer

At RCL 5 and above, Links provide the ability to move energy instantly across a room. A modern broker treats Links as a "Short Circuit" in the logistics network. A LinkNetwork process runs prior to the matching phase, identifying "Source Links" (near energy sources) and "Hub Links" (near Storage). If a Source Link is full, it automatically transfers its energy to the Hub Link, effectively removing that energy from the hauler's workload and saving dozens of CPU cycles per tick.

Memory Engineering: The Heap-First Architecture

To survive the "Serialization Wall," top-tier bots move away from Memory as their primary data store. Instead, they use a **Global Cache** pattern.

The Hydration/Dehydration Mechanism

On the first tick of a new isolate (a Global Reset), the bot performs a "Hydration" step. It parses the necessary parts of the Memory object and instantiates rich JavaScript classes for the LogisticsNetwork, Colonies, and Overlords. For the next several thousand ticks, these objects live in the Heap. Accessing global.logisticsNetwork is nanoseconds fast, whereas accessing Memory.logistics requires a proxy-wrapped JSON lookup.

Segmented Memory for Historical Data

For logistics data that is too large for the 2MB Memory limit—such as 10,000 ticks of market price history or massive cost matrices for remote mining—modern bots utilize RawMemory.segments. These 100kb chunks of storage are loaded asynchronously. A top-tier broker uses a "Segment Manager" to request these chunks as needed, allowing the bot to make logistics decisions based on historical trends without bloating the main memory string.

Military Logistics: Defense and War

A logistics broker proves its worth during a siege. In a legacy bot, Towers often run out of energy because haulers are busy filling Extensions or because the hauler was killed and no replacement was dispatched.

Tower Defense Logic and Energy Prioritization

A top-tier broker treats Towers as "Critical Priority" requesters. The moment a Tower fires, its energy level drops, and the LogisticsNetwork registers a new request. Because the Gale-Shapley algorithm uses priority as its primary weight, haulers will immediately drop their current tasks—even if they were moving energy to the Spawn—to refill the Towers. Furthermore, the broker implements "Pre-spawning" for logistics creeps during war. If the Overseer detects a hostile presence, it increases the target hauler count by 20% to account for potential losses and ensures that every hauler is equipped with "Tough" or "Heal" parts to survive moving through a sieged base.

Squad-Based Logistics

For offensive operations, logistics must move with the army. A "Squad" object (often a "Quad" of 4 creeps) includes a dedicated "Healer/Buffer" that functions as a mobile requester. The Logistics Broker coordinates a "Supply Line" of haulers that move energy from the home colony to the front line, potentially spanning multiple rooms. This requires the broker to calculate "Safe Paths" that avoid enemy towers, utilizing the CostMatrix modifications discussed in the Cartography section.

Implementation Roadmap for Modernization

Transforming a legacy codebase (like grgisme/screeps) into a top-tier system requires a phased approach. The goal is to move from a collection of scripts to a managed system of processes.

Phase 1: Toolchain and Environment (The Foundation)

Before any logic is rewritten, the development environment must be upgraded. This involves transitioning from raw JavaScript to TypeScript and implementing a bundling pipeline using Rollup. This allows for the use of complex interfaces and classes, which are essential for the TransportRequest and LogisticsNetwork structures.

Phase 2: The Registry and Kernel (The Core)

The next step is to implement a Kernel that manages the tick lifecycle. The legacy monolithic loop is replaced with a process-based scheduler. Simultaneously, the LogisticsNetwork registry is created. During this phase, creeps still use legacy roles, but they now query the LogisticsNetwork for their targets instead of using room.find().

Phase 3: Stable Matching and Reservations (The Brain)

Once the registry is stable, the "Decision Logic" is moved from the creeps to the broker. The Gale-Shapley algorithm is implemented to assign tasks. The Zerg (creep wrapper) class is introduced to handle the execution of these assigned tasks. The reservation system is activated to eliminate "Energy Racing."

Phase 4: Global Cache and Optimization (The Performance)

The final phase involves moving all persistent state to the Global Heap. The refresh() pattern is implemented to allow class instances to survive between ticks. The path cache and link balancing logic are integrated, resulting in a system that can handle hundreds of creeps with minimal CPU overhead.

Phase	Core Objective	Key Technology	Impact
Phase 1	Modern Tooling	TypeScript, Rollup, ESLint.	Refactoring safety and code modularity.
Phase 2	Centralized Registry	Provider/Requester Registry.	Eliminates O(N) sensing costs.
Phase 3	Optimal Coordination	Gale-Shapley Stable Matching.	Zero energy racing and 100% throughput.
Phase 4	Performance Scaling	Global Heap Cache,	>80% reduction in

Phase	Core Objective	Key Technology	Impact
		Path Caching.	per-tick CPU cost.

Strategic Implications and Future Outlook

The implementation of a Global Logistics Broker represents the "Industrial Revolution" of a Screeps empire. It shifts the player's focus from micro-managing individual units to designing high-level economic and military strategies.

Cross-Shard Economies

As an empire grows beyond GCL 20, the logistics broker must eventually handle inter-shard trade. This involves using InterShardMemory to coordinate the transfer of resources through portals. A top-tier broker will treat a portal as a "Virtual Sink" on one shard and a "Virtual Source" on another, allowing for the seamless flow of energy from a high-yield room on Shard 0 to a fledgling colony on Shard 3.

Market Integration and Automated Trading

A modern logistics broker is also the engine of the bot's economy. Excess energy identified by the broker is automatically sent to the Terminal and sold on the global market. The broker uses market data (stored in memory segments) to determine when to buy minerals for laboratory reactions or when to sell energy for credits. This automated wealth generation provides the resources necessary to sustain high-level warfare and empire expansion.

Conclusion: The Path to Industrial Autonomy

The Screeps architecture landscape has evolved far beyond the simple role-based scripts of the 2017 era. To compete in the 2025 ecosystem, a bot must be treated as a complex systems engineering project. The Global Logistics Broker is the heart of this system, transforming a collection of autonomous agents into a unified, high-throughput supply chain.
By implementing stable matching algorithms to ensure optimal coordination, reservation systems to eliminate resource contention, and a heap-first architecture to bypass serialization bottlenecks, a developer can achieve levels of efficiency that were previously impossible. This architectural shift is not merely an optimization; it is the only viable path to managing the non-linear complexity of a high-GCL empire. The move from "Search and Rescue" to "Command and Control" is the defining characteristic of a top-tier Screeps bot, enabling the player to dominate the world through computational superiority and strategic orchestration.

Works cited

1. Server-side architecture overview | Screeps Documentation, https://docs.screeps.com/architecture.html 2. IVM heap usage & game objects | Screeps Forum, https://screeps.com/forum/topic/2163/ivm-heap-usage-game-objects 3. Screeps #6: Verifiably refreshed - Ben Bartlett, https://bencbartlett.com/blog/screeps-6-verifiably-refreshed/ 4. Global Objects | Screeps Documentation, https://docs.screeps.com/global-objects.html 5. bonzaiferroni/bonzAI-framework - GitHub, https://github.com/bonzaiferroni/bonzAI-framework 6. Releases · bencbartlett/Overmind - GitHub, https://github.com/bencbartlett/Overmind/releases

7. Overmind - Ben Bartlett, https://bencbartlett.com/overmind-docs/ 8. Energy - Screeps Wiki, https://wiki.screepspl.us/Energy/ 9. Issue storing energy in containers :: Screeps: World Help - Steam Community, https://steamcommunity.com/app/464350/discussions/5/3335371283878126074/ 10. Gale–Shapley algorithm - Wikipedia, https://en.wikipedia.org/wiki/Gale%E2%80%93Shapley_algorithm 11. The stable matching algorithm - Washington, https://courses.cs.washington.edu/courses/cse421/25sp/lectures/pdf/lecture2.pdf 12. Implementation of the Gale-Shapley (Stable Matching) algorithm. - GitHub, https://github.com/colorstackorg/stable-matching 13. Gale Shapley Algorithm for Stable Matching | by Sai Panyam | saipanyam - Medium, https://medium.com/saipanyam/gale-shapley-algorithm-for-stable-matching-736ff452dac4 14. Community Communication - Screeps Wiki, https://wiki.screepspl.us/Community_Communication/

Strategic Engineering of High-Performance Resource Engines: A Technical Analysis of Remote Mining and Reservation Architectures in Screeps

The transition from a localized, single-room economy to a distributed, multi-room resource engine represents the primary inflection point in a Screeps AI's developmental trajectory. In the deterministic environment of the Screeps game engine, energy acquisition is the fundamental substrate upon which all other operations—military expansion, technological advancement through Global Control Levels (GCL), and market participation—are built. The "Resource Engine," specifically defined here as the integrated system of remote harvesting, room reservation, and long-range logistics, serves as the primary driver of colony growth. While basic harvesting logic suffices for early Room Control Levels (RCL), the implementation of a top-tier resource engine requires a rigorous application of computational geometry, graph theory, and precise mathematical modeling of creep body mechanics and room-level thermodynamics.

Mathematical Foundations of the Harvesting Cycle

The optimization of a resource engine begins with the source. Each energy source in the Screeps world is governed by a 300-tick regeneration cycle. In unreserved rooms, sources provide 1,500 energy units per cycle, translating to a maximum throughput of 5 energy units per tick. The application of a reserveController intent by a creep with one or more CLAIM parts doubles this capacity to 3,000 energy units per cycle, or 10 energy units per tick. This doubling of output is the first and most significant efficiency gain in a remote mining operation.

Harvesting Power and Miner Body Composition

The Screeps engine defines HARVEST_POWER as 2 energy units per tick per WORK part. To achieve 100% harvesting efficiency in a reserved room, a miner must possess exactly 5 WORK parts, yielding 10 energy units per tick. Any surplus of WORK parts beyond this limit provides no additional energy from the source, although it may be utilized for collateral tasks such as the maintenance of local infrastructure.

Mining State	Source Capacity	Regeration Cycle	Required WORK Parts	Net Energy/Tick
Unreserved	1,500	300 Ticks	3 (effective 2.5)	5
Reserved	3,000	300 Ticks	5	10
Source Keeper	4,000	300 Ticks	7 (effective 6.6)	13.33

The efficiency of a miner is defined as the ratio of energy harvested to the energy cost of the creep over its 1,500-tick lifespan. A standard remote miner with the body $$ costs 800 energy. Assuming a travel distance of 50 tiles, the miner arrives at the source with 1,450 ticks of life

remaining. Total energy harvested is calculated as:
Where D is the distance from the spawn, P_{harvest} is HARVEST_POWER, and W is the number of WORK parts. For a 5-WORK miner at 50 tiles, E_{total} = (1450) \times (10) = 14,500 energy units. The net profit of the creep is E_{total} - C_{spawn}, or 13,700 energy units.
In advanced implementations, a 6th WORK part is often added to allow the miner to repair its own container. Since a container in a remote room decays by 5,000 hits every 100 ticks, it requires 50 energy units of repair every 100 ticks. One WORK part repairs 100 hits per tick at the cost of 1 energy unit. Therefore, the miner can spend 1 tick out of every 20 repairing the container to maintain the structure indefinitely, eliminating the need for a dedicated repair creep and reducing CPU overhead by centralizing tasks.

The Physics of Logistics: Transportation and Fatigue

The most complex component of the resource engine is the transportation of harvested energy back to the home room. Unlike harvesting, which is capped by source regeneration, hauling efficiency is a variable of distance, creep body composition, and terrain type.

The Fatigue Equation and Movement Optimization

Fatigue is the primary constraint on hauler throughput. The engine generates fatigue based on the following formula:
However, CARRY parts only generate fatigue when they contain resources. An empty hauler moves at a speed of 1 tile per tick regardless of its MOVE to CARRY ratio, provided it has at least one active MOVE part. A loaded hauler requires one MOVE part for every non-MOVE part to maintain a speed of 1 tile per tick on plain terrain (V = 1 \text{ tile/tick}). On roads, the fatigue generation is halved, allowing a 1:2 ratio of MOVE to CARRY parts to maintain maximum velocity.

Terrain Type	Fatigue per Part	MO[span_47](start_span)[span_47](end_span)[span_54](start_span)[span_54](end_span)VE Ratio Required	Effective Speed (Max)
Road	1	1:2	1 tile/tick
Plain	2	1:1	1 tile/tick
Swamp	10	5:1	1 tile/tick

The total "hauling power" required for a source is the product of the energy generated per tick and the round-trip distance. For a reserved source (10 \text{ e/tick}) at a distance of 50 tiles, the round-trip distance is 100 tiles. To prevent energy from accumulating and decaying on the ground, the hauler pool must move 1,000 energy units every 100 ticks.

Part-Count Balancing vs. Headcount

Top-tier resource engines utilize "Part-Count Balancing" to determine hauler requirements. Instead of spawning a fixed number of creeps, the engine calculates the total CARRY parts needed:
Where E_{gen} is energy per tick and D is distance. For the 50-tile example, 20 CARRY parts are required. At RCL 3, where energy capacity is 800, a creep can support a maximum of 10

CARRY and 5 MOVE parts (costing 750 energy). Thus, the engine spawns two haulers. At RCL 8, the same demand is met by a single, more efficient creep with 20 CARRY and 10 MOVE parts, reducing CPU costs associated with pathfinding and intent processing.

Room Reservation and Sovereignty

Reservation is the mechanism that transitions a remote room from a temporary resource site to a stable extension of the colony's economy. Beyond the 100% increase in energy yield, reservation provides a critical defensive benefit: it prevents NPC invaders from spawning at exits leading to the reserved room.

The Economy of the CLAIM Part

The CLAIM part is the most expensive and specialized body part in the game, costing 600 energy and reducing the creep's lifetime from 1,500 to 500 or 600 ticks. A reserver body is typically [1 \text{ CLAIM}, 1 \text{ MOVE}], costing 650 energy.
The reservation buffer on a controller can store up to 5,000 ticks. Each reserveController action adds N ticks to the buffer, where N is the number of CLAIM parts, and the buffer decays by 1 tick per game tick. To increase the reservation level, a creep must have at least 2 CLAIM parts, as 1 part merely offsets the natural decay.
| Strategy | CLAIM Parts | Net Buffer Gain | Ticks to Max Buffer | | :--- | :--- | :--- | :--- | | Maintenance | 1 | 0 ticks/tick | Constant | | Buffer Building | 2 | 1 tick/tick | 5,000 Ticks | | Rapid Reset | 5 | 4 ticks/tick | 1,250 Ticks |
A top-tier engine avoids constant reserver presence. Instead, it employs "Buffer Cycling." The engine tracks the ticksToEnd of the reservation. When the buffer falls below a threshold defined as D + T_{spawn} + T_{safety} (where D is travel distance, T_{spawn} is time to spawn, and T_{safety} is a buffer for congestion), the engine pushes a reserver request to the spawn queue. This minimizes the energy spent on the expensive 600-tick lifetime of reservers.

Architectural Forensics: The Overmind Framework

Modernization of a resource engine necessitates a shift from role-based logic to an objective-based architectural framework, such as the "Overmind" or "Boardroom" models. Role-based systems, where each creep independently decides its action (e.g., "if I am a harvester, I find a source"), suffer from poor scalability and inefficient resource allocation.

Directives and Overlords

The Overmind architecture organizes the empire into a hierarchy of functional objects:

1. **Colony:** The top-level object wrapping a single owned room and its associated remote territories.
2. **Directive:** A persistent object (often associated with a flag) that identifies a strategic objective in the game world, such as RemoteMiningDirective or GuardDirective.
3. **Overlord:** A process instantiated by a Directive to handle the fulfillment of the objective. The MiningOverlord calculates the necessary body parts for miners and haulers, monitors their life-cycles, and issues spawn requests.
4. **Zerg/Minions:** Creeps that are stripped of independent logic and instead act as executors

of Tasks assigned by the Overlord.

This structure allows the engine to treat remote mining as a "straightforward optimization problem". When a RemoteMiningDirective is placed in a room, the associated Overlord analyzes the room's topography, identifies the sources, calculates the distances to the home-room storage, and determines the optimal part counts for the logistics loop.

The Logistics Network and Transport Requests

Fulfillment of energy transport is handled by a global LogisticsNetwork. Instead of a hauler being "tethered" to a specific miner, it responds to TransportRequests.

- **Output Requests:** A container at a remote source reaching a capacity threshold (e.g., 1,600/2,000 units) generates an output request.
- **Input Requests:** A tower needing 200 energy or a spawn needing 300 energy generates an input request.
- **Priority Scoring:** The LogisticsNetwork scores requests based on distance and urgency. A hauler is assigned to pick up resources from a remote source and is then dynamically reassigned to the highest priority input request in the home room.

This "task-chaining" allows haulers to maintain high duty cycles. For example, a hauler returning from a remote room might be assigned to fill a nearby tower before returning to its "idle" state at the storage, maximizing the energy moved per CPU tick.

Defensive Protocols and NPC Management

Remote rooms are the primary engagement zone for NPC Invaders and Source Keepers. A high-performance resource engine must integrate defensive automation directly into its logistics cycle.

NPC Invader Spawn Logic

Invaders spawn based on an internal counter that tracks the total energy harvested in a room, typically triggering after 100,000 energy units are extracted. These invaders come in two tiers: "Light" creeps in rooms below RCL 4 and "Heavy" creeps in higher-level rooms.

Top-tier engines implement a multi-stage response:

1. **Detection:** The engine scans remote rooms for FIND_HOSTILE_CREEPS. If an invader is detected, a "State of Emergency" is declared for that room.
2. **Evacuation:** All non-combat creeps (miners and haulers) are ordered to flee to a safe room to prevent unnecessary life-loss and energy drop-off.
3. **Dynamic Response Spawning:** The DefenseOverlord analyzes the invader's body. If the invader is a melee attacker, the engine spawns a ranged kiter. If it is a squad with a healer, the engine spawns a high-damage melee attacker or a superior ranged-mass-attack creep.
4. **Tombstone Reclamation:** After the invader is destroyed, the engine assigns a hauler to withdraw the energy and boosts from the invader's tombstone, ensuring the engagement remains energy-positive.

Source Keeper Room Operations

Source Keeper (SK) rooms are high-risk, high-reward environments providing 4,000 energy per source. Mining these rooms requires "Keeper Suppression." A combat creep, typically with a high HEAL and ATTACK count, is stationed in the room to kill Source Keepers as they spawn from their Lairs every 300 ticks.

Role	SK Miner	SK Hauler	SK Guard
Body	7 WORK, 1 CARRY, 4 MOVE	20 CARRY, 10 MOVE	10 ATTACK, 5 HEAL, 15 MOVE
Function	Static harvest and repair	Long-range transport	Suppression of Lairs

The introduction of InvaderCores adds a new layer of complexity. These cores can launch "Lesser Cores" into remote rooms, which reserve the controller and prevent harvesting until destroyed. A top-tier engine must detect these cores and spawn a "Cleaner" creep—a simple body with ATTACK or WORK parts—to dismantle the core and restore harvesting operations.

Traffic Management and Pathing Optimization

Traffic is the "silent killer" of remote mining efficiency. In a single-lane road system, one slow creep can delay the entire logistics chain, leading to source overflow and decay.

Movement Libraries and Caching

Top-tier bots utilize movement libraries like Traveler or Cartographer to replace the default creep.moveTo(). These libraries offer:

- **Path Caching:** Paths are stored in Heap or Memory to avoid expensive PathFinder calls every tick.
- **Hostile Room Avoidance:** The pathfinder automatically routes around rooms that have been flagged as dangerous.
- **Traffic Shoving:** Idle creeps are ordered to move out of the way of moving creeps.

The Bipartite Matching Algorithm for Traffic

The gold standard for traffic management is the Bipartite Matching Problem (BMP). Every tick, the engine collects move intents from all creeps in a room.

1. **Intent Mapping:** Creeps express a desire for a target tile or a preference to stay put.
2. **Graph Construction:** The engine builds a graph connecting creeps to potential tiles.
3. **Resolution:** The engine resolves collisions by "shoving" lower-priority creeps (like upgraders or stationary miners) into empty adjacent tiles to allow higher-priority haulers to maintain their velocity.

This ensures that even in narrow corridors, creeps can "swap" positions in a single tick, maintaining a constant flow of resources.

Infrastructure: Road Logistics and Container Management

The decision to build roads is a purely economic one. The cost to repair a road on plain terrain is 1 energy per 1,000 ticks. The cost of the extra MOVE parts required to move at the same

speed without a road is significantly higher over the life of the creep.

The Road-Repair-on-Transit Pattern

To maintain hundreds of tiles of remote roads, the most efficient method is attaching one WORK part to haulers.

- **Mechanism:** As the hauler moves back and forth, it checks the hits of the road tile it is standing on.
- **Action:** If the road is below 100% hits, the hauler issues a repair intent.
- **Result:** This keeps the road network at full health without the CPU or energy cost of a dedicated repairer.

Container Placement and Static Harvesting

Containers are essential for "Static Harvesting," where the miner never moves. The container should be placed on the tile adjacent to the source that offers the best pathing to the room exit.

- **Decay:** Containers in remote rooms decay at 5,000 hits per 100 ticks.
- **Repair Cost:** Maintenance costs 0.5 energy per tick (50 \text{ energy} / 100 \text{ ticks}).
- **Dropped Energy:** If no container is used, dropped energy decays at \lceil \text{amount} / 1000 \rceil per tick. At a reserved source producing 3,000 energy, the decay of dropped energy significantly exceeds the repair cost of a container, making containers mandatory for top-tier efficiency.

Economic Analysis: Return on Investment (ROI) and Scaling

A resource engine must be self-evaluating. Top-tier implementations track the "Energy Per Tick" (EPT) of every remote room to determine which targets are most profitable.

Efficiency Metrics

The efficiency of a remote room is defined as:
Where E_{max} is 10 energy/tick for reserved rooms.

- At 50 tiles, efficiency is approximately 65%.
- At 100 tiles, efficiency drops to 52%.
- Beyond 250 tiles, the cost of the haulers and the road decay often results in negative efficiency, marking the "Economic Horizon" of the colony.

Scaling with Power Creeps and Boosts

At high RCL, the resource engine can be further optimized using PowerCreeps and lab boosts.

- **EXTEND_SOURCE:** A Power Creep can increase the capacity of a source beyond 3,000, allowing for massive energy spikes.
- **REGEN_SOURCE:** Accelerates the 300-tick regeneration cycle.
- **WORK Boosts:** Compounds like UO2 (Utrium Oxide) increase harvest effectiveness, allowing a miner to exhaust a source with fewer body parts, thereby saving on spawn energy and CPU.

Modernization Roadmap: Implementing the Top-Tier Engine

Based on the forensic analysis of standard Screeps architectures and the provided documentation, the following steps are required to implement a top-tier resource engine.

Phase 1: The Logistics Backbone

1. **Implement the Logistics Network:** Move energy handling to a request-based system where haulers respond to TransportRequests rather than being hard-coded to sources.
2. **Part-Count Balancing:** Replace "headcount" logic with a system that calculates the total CARRY parts needed for a route and spawns the minimum number of creeps to fulfill that need.
3. **Road Automation:** Implement the "Repair-on-Transit" logic for haulers to maintain infrastructure automatically.

Phase 2: The Overlord Migration

1. **Refactor into Directives:** Use flag-based directives to identify remote mining targets.
2. **Instantiate Overlords:** Create a MiningOverlord to manage the life-cycles of remote creeps and handle local state transitions (e.g., switching from "Mining" to "Evacuating" during an invasion).
3. **Buffer-Based Reservation:** Implement reservation logic that maintains the 5,000-tick buffer with minimal reserver uptime.

Phase 3: Traffic and Pathing Sovereignty

1. **Integrate Cartographer/Traveler:** Replace moveTo with a cached pathfinder that supports traffic management and hostile room avoidance.
2. **Implement BMP Traffic Management:** Use graph-based matching to resolve creep collisions in high-traffic corridors.
3. **CostMatrix Optimization:** Dynamically update CostMatrices to account for road decay and stationary creeps, ensuring haulers always take the most efficient route.

Phase 4: Defensive and Offensive Integration

1. **Automate Invader Responses:** Build a DefenseOverlord that calculates specific counter-bodies for NPC invaders based on their detected parts.
2. **SK Suppression Logic:** Implement the "Lair-Cycling" guard for Source Keeper rooms to maximize high-yield energy extraction.
3. **Invader Core Cleanup:** Automate the detection and destruction of Inva[span_130](start_span)[span_130](end_span)derCores to prevent sovereignty loss in remote territories.

Conclusion: The Integrated Resource Engine

The development of a high-performance resource engine is not a singular task but an ongoing engineering effort to balance energy throughput against CPU and spawn-time costs. By applying the mathematical rigors of body-part balancing, adopting the modularity of the Overmind architecture, and implementing sophisticated traffic management, an AI can achieve a level of economic stability that supports sustained expansion. In the competitive environment of Screeps, the resource engine is the heart of the colony; its efficiency determines the ultimate survival and dominance of the empire. Through the implementation of these best practices, the engineer moves from manual management to a state of autonomous economic sovereignty.

Works cited

1. Workflow tips and prioritization for new players? | Screeps Forum, https://screeps.com/forum/topic/2556/workflow-tips-and-prioritization-for-new-players 2. Resources | Screeps Documentation, https://docs.screeps.com/resources.html 3. Remote mining article by slowmotionghost · Pull Request #60 · screeps/docs - GitHub, https://github.com/screeps/docs/pull/60/files 4. Great Filters - Screeps Wiki, https://wiki.screepspl.us/Great_Filters/ 5. Remote Harvesting - Screeps Wiki, https://wiki.screepspl.us/Remote_Harvesting/ 6. Screeps #1: Overlord overload | Ben Bartlett, https://bencbartlett.com/blog/screeps-1-overlord-overload/ 7. Journey to Solving the Traffic Management Problem - Harabi Screeps, https://sy-harabi.github.io/Journey-to-Solving-the-Traffic-Management-Problem/ 8. Reservation - Screeps Wiki, https://wiki.screepspl.us/Reservation/ 9. This is the slowest paced game ever. Does it get more interesting once GCL goes up? : r/screeps - Reddit, https://www.reddit.com/r/screeps/comments/8181nc/this_is_the_slowest_paced_game_ever_does_it_get/ 10. Creep | Screeps Wiki | Fandom, https://screeps.fandom.com/wiki/Creep 11. Is there a guide out there for the ideal ratios of your creeps? : r/screeps - Reddit, https://www.reddit.com/r/screeps/comments/864qy3/is_there_a_guide_out_there_for_the_ideal_ratios/ 12. Faster Controller Levelling : r/screeps - Reddit, https://www.reddit.com/r/screeps/comments/5jia18/faster_controller_levelling/ 13. Questions on roles | Screeps Forum, https://screeps.com/forum/topic/913/questions-on-roles 14. Energy - Screeps Wiki, https://wiki.screepspl.us/Energy/ 15. What's the function to calculate creep cost? | Screeps Forum, https://screeps.com/forum/topic/2288/what-s-the-function-to-calculate-creep-cost 16. NPC Invaders | Screeps Documentation, https://docs.screeps.com/invaders.html 17. CLAIM body parts and 'time to live' : r/screeps - Reddit, https://www.reddit.com/r/screeps/comments/6pz7ij/claim_body_parts_and_time_to_live/ 18. CLAIM part too inefficient | Screeps Forum, https://screeps.com/forum/topic/1167/claim-part-too-inefficient 19. Respawning - Screeps Documentation, https://docs.screeps.com/respawn.html 20. Screeps #5: Refactoring for Remote Mining - Field Journal, https://jonwinsley.com/notes/screeps-refactoring-remote-mining 21. How i can control second room? | Screeps Forum, https://screeps.com/forum/topic/1377/how-i-can-control-second-room 22. Releases · bencbartlett/Overmind - GitHub, https://github.com/bencbartlett/Overmind/releases 23. Overmind/src/overlords/Overlord.ts at master · bencbartlett/Overmind - GitHub,

https://github.com/bencbartlett/Overmind/blob/master/src/overlords/Overlord.ts 24. Source keeper...what is that? | Screeps Forum, https://screeps.com/forum/topic/1533/source-keeper-what-is-that 25. Invader - Screeps Wiki, https://wiki.screepspl.us/Invader/ 26. Keeper Lairs and Invader Swarms Strategy | Screeps Forum, https://screeps.com/forum/topic/327/keeper-lairs-and-invader-swarms-strategy 27. Invader Cores suck. No Invader Core Shard | Screeps Forum, https://screeps.com/forum/topic/2823/invader-cores-suck-no-invader-core-shard 28. Traffic Management | screeps-cartographer, https://glitchassassin.github.io/screeps-cartographer/pages/trafficManagement.html 29. Cartographer is an advanced (and open source) movement library for Screeps - GitHub, https://github.com/glitchassassin/screeps-cartographer 30. bonzaiferroni/Traveler: Traveler - A general movement solution for Screeps.com - GitHub, https://github.com/bonzaiferroni/Traveler 31. screeps-cartographer, https://glitchassassin.github.io/screeps-cartographer/ 32. Screeps #27: Optimizing Pathfinding with Rust | Field Journal, https://jonwinsley.com/notes/screeps-clockwork 33. Power Creeps update | Screeps Forum, https://screeps.com/forum/post/10564

The Automated Foundry: Architecting Scalable Spawn Systems in Screeps

The realization of a high-level Screeps autonomous agent depends fundamentally on the efficiency, resilience, and scalability of its unit production infrastructure. In the hierarchical ecosystem of a professional bot, the Hatchery serves as the critical bridge between the strategic intent of high-level Overlords and the tactical execution of individual creeps. This report delineates the systems engineering requirements for a centralized Hatchery within a Kernel-driven, TypeScript-based environment, focusing on deadlock-resistant queuing, dynamic morphology, logistical energy distribution, and the lifecycle handshaking necessary to maintain a continuous operational presence.

The Ontological Foundations of Unit Production

In the context of a Kernel/Process architecture, the spawning of a creep is not a singular event but a multi-tick transaction that necessitates precise coordination between the "Factory" (the Hatchery) and the "Consumer" (the Overlord). A centralized Hatchery acts as a persistent system service, abstracting the physical structures—Spawns and Extensions—away from the functional logic of the bot. By implementing a Factory pattern, the developer ensures that the higher-level logic remains decoupled from the room-level constraints, such as available energy or structure counts.

The Hatchery must be viewed as the metabolic core of the colony. Its primary objective is to maintain the population levels required to sustain the energy loop and protect the room's integrity. This necessitates a high degree of integration with the room's base plan. Utilizing algorithms like the Distance Transform ensures that the Hatchery is placed in a location that minimizes pathing costs for both energy ingress and unit egress. In a "bunker" layout, the Hatchery is often hard-coded to optimize the interaction between stationary "Managers" and mobile "Queens" or "Fillers," who are responsible for the physical transfer of energy from storage buffers to the extensions.

The evolution of a room's capability is intrinsically tied to its Room Controller Level (RCL). As the room upgrades, the Hatchery must manage an increasing number of extensions and spawns, moving from a single 300-energy capacity spawn to a complex network of 60 extensions and 3 spawns at RCL 8.

Structure and Energy Constraints per Level

RCL	Energy to Upgrade	Spawn Count	Extension Count	Extension Capacity	Total Energy Capacity
1	200	1	0	0	300
2	45,000	1	5	50	550
3	135,000	1	10	50	800
4	405,000	1	20	50	1,300
5	1,215,000	1	30	50	1,800
6	3,645,000	1	40	50	2,300

RCL	Energy to Upgrade	Spawn Count	Extension Count	Extension Capacity	Total Energy Capacity
7	10,935,000	2	50	100	5,600
8	—	3	60	200	12,900

This progression, summarized from the official game mechanics, illustrates that at high levels, the Hatchery must coordinate the simultaneous operation of three spawns, requiring a synchronization layer to ensure that energy is not over-committed during a single tick.

The Hatchery Queue: Priority and Deadlock Prevention

The most significant challenge in architecting a Hatchery is the management of the spawn queue. A naive implementation that processes requests in the order received will invariably encounter a deadlock when the room's energy supply is depleted. This scenario typically occurs after a "room wipe," where all creeps have been lost to combat or environmental decay. If the top of the queue is occupied by a high-cost creep (e.g., an 800-energy miner) and the room only has 300 energy available, the spawn will sit idle indefinitely, as no harvesters exist to replenish the extensions.

The Tiered Priority Model

A robust "Priority Queue" must prioritize "metabolic maintenance" over "strategic expansion." The hierarchy of requests is generally organized into several distinct tiers to prevent critical failures.

1. **Metabolic Tier (Critical):** Units essential for the immediate survival of the energy loop. This includes small "Bootstrapper" harvesters and "Fillers" that distribute energy to extensions. These units must be spawned regardless of the room's long-term energy capacity.
2. **Defensive Tier (Urgent):** Combat units and tower-refilling creeps required to respond to hostile incursions.
3. **Economic Tier (Standard):** Standard-size miners, haulers, and upgraders that maintain the steady-state economy of the room.
4. **Strategic Tier (Low):** Pioneer units, claimers, and remote-room exploiters.

To implement this, each request in the queue is an object containing the required body or a body-generator template, a priority value, and a unique identifier for the requesting Overlord.

Lifetime Value and Flow Analysis

Advanced systems often move beyond static counts toward a "Lifetime Value" or "Flow-Based" model. Instead of asking "do I have 4 harvesters?", the Hatchery evaluates the room's energy-per-tick (E/t) production against its consumption. If the net flow is negative, the queue automatically injects harvesters into the high-priority slots. This approach allows the bot to prioritize a small, cheap harvester that produces 2 E/t over a large upgrader that would consume 5 E/t, particularly when the storage buffer is low.

Deadlock Resolution: The Emergency Pivot

Deadlock prevention is handled by a "Safety Check" performed at the beginning of each tick. The Hatchery examines the room's energyAvailable and its current population of fillers and harvesters. If the room has zero active creeps capable of harvesting energy, the Hatchery enters "Emergency Mode".
In this state, the Hatchery logic overrides the current top of the queue if that request's cost exceeds the current energyAvailable. It immediately generates a "Bootstrapper" creep—a minimal unit typically consisting of `` costing exactly 300 energy. This allows the room to bootstrap itself using the automatic energy regeneration provided by the Spawn (1 energy per tick until 300). Once the extensions are filled, the Hatchery transitions back to "Economy Mode" and resumes processing the standard queue.

Morphology and Morphology Algorithms: Dynamic Body Generation

A high-level bot must adapt its creep morphology to the environment and the current room state. Static body definitions are inefficient, as they either over-budget for early-game rooms or under-utilize the massive energy pools of RCL 8 bunkers.

The Template Repetition Algorithm

The standard algorithm for dynamic body generation is "Template Repetition." This involves defining a "unit" of parts representing a specific ratio and repeating it until an energy or size limit is reached.
The cost C of a body is the sum of the costs of its constituent parts:
Where P represents the set of parts in the creep.

Part Type	Cost (Energy)	Function
MOVE	50	Decreases fatigue by 2 per tick.
WORK	100	Harvests 2 energy, repairs 100 hits, or upgrades 1 RCL.
CARRY	50	Stores 50 units of resources.
ATTACK	80	Deals 30 hits in melee combat.
RANGED_ATTACK	150	Deals 10-40 hits at range.
HEAL	250	Restores 12-48 hits.
CLAIM	600	Claims or reserves a controller.
TOUGH	10	Acts as a damage sponge.

To generate a harvester body with a ratio of 2 WORK : 1 MOVE, the Hatchery uses a template of ``, which costs 250 energy. The algorithm calculates the number of iterations n:
The final body is then the template repeated n times.

Strategic Part Ordering

The sequence of parts in the array is critical for survivability. Damage is applied to body parts starting from the first element in the array (index 0). A functional Foundry must order parts based on the creep's intended exposure to threat:

- **Logistics Creeps:** Often ordered as ``. This ensures that even if a creep takes minor damage while moving, it retains its MOVE parts until the very end, allowing it to limp back to safety.
- **Combat Creeps:** Use "Tough Buffering." TOUGH parts are placed at the front to soak up damage, while HEAL or ATTACK parts are placed at the rear to maintain effectiveness during a siege.
- **Prespawning and TTL:** To maintain continuous uptime, the Hatchery monitors the Time to Live (TTL) of active creeps. It calculates the time required to spawn a replacement (3 \times \text{number of parts}) plus the travel time to the destination. If TTL \le \text{SpawnTime} + \text{TravelTime}, the replacement is added to the queue.

Movement and Fatigue Optimization

A high-level bot avoids the waste of excessive MOVE parts. Fatigue F is generated by every non-MOVE part (excluding empty CARRY parts):
Where W is the number of non-MOVE parts, K is the terrain factor (0.5 for roads, 1.0 for plains, 5.0 for swamps), and M is the number of MOVE parts. The goal of the body generator is to ensure F \le 0 every tick. For a hauler moving on roads (K=0.5), the Foundry will generate a ratio of 1 MOVE : 2 CARRY, doubling the efficiency of the creep compared to a generic 1:1 ratio.

Extension Management: Logistical Metabolic Distribution

The Hatchery's primary throughput bottleneck is not the spawn time itself, but the speed at which the surrounding Extensions are refilled. This is particularly true at RCL 7 and 8, where a single spawn of a max-size creep can consume 12,500+ energy, requiring the replenishment of dozens of extensions.

Single Request vs. Atomic Tasks

There are two primary models for managing the Logistics Network's interaction with the Hatchery:

1. **The Atomic Model:** Every extension creates an individual task in the Logistics Network when its energy < energyCapacity. While this ensures all extensions are eventually filled, it generates massive CPU overhead. A room with 60 extensions would generate 60 task objects, 60 pathfinding calls, and 60 intent calls for creep.transfer().
2. **The Batch Request (The "Factory" Pattern):** The Hatchery monitors the total energy in the extension network. When the room's energyAvailable < energyCapacityAvailable, the Hatchery issues a single "Hatchery Refill" request to the Logistics Network with the total deficit (e.g., 800 energy).

High-level bots utilize the Batch Request model. A single "Queen" or "Filler" creep picks up the total required energy from the Storage or Link and executes a "Fill Loop." The creep moves to a central position in the extension array and deposits energy into multiple structures in a single pathing cycle. Utilizing the StructureExtension.store.getFreeCapacity() method, the filler can efficiently chain transfers without re-running its logic every tick.

Extension Deposit Availability (EDA) and Base Layout

The efficiency of refilling is often measured by the EDA metric—the time it takes for a filler to distribute its entire carry capacity. Compact extension arrays, such as "flower" or "stamp" patterns, are designed to allow a creep to reach up to eight extensions from a single tile. In a professional bunker layout, the filler's path is often hard-coded or pre-calculated during the "Architect" phase to eliminate the CPU cost of findClosestByPath.

Power Creep Optimization

At RCL 8, the metabolic throughput of the Hatchery can be significantly enhanced by the OPERATE_EXTENSION power. An Operator-class Power Creep can instantly refill a percentage of all extensions in the room (up to 100% at level 5) by drawing energy from a nearby container or storage. This eliminates the need for manual filler creeps during high-intensity periods (such as defensive sieges) and allows the spawns to run at 100% duty cycle. The Hatchery must coordinate with the Power Creep Manager to ensure ops resources are available for this intent.

Systems Integration: The Creep Registration Handshake

The transition of a creep from the "Spawning" state to the "Active" state is a critical juncture in a Kernel/Process architecture. A failure in the handshaking logic results in "Orphan Creeps"—units that consume energy and CPU but perform no tasks—or "Zombie Processes"—logic loops that wait for a creep name that has been deleted by a memory cleanup script.

The Three-Phase Handshake

The Hatchery implements a three-phase handshake to pass control of a new unit to the requesting Overlord:

1. **Phase I: The Commitment (Tick N):** The Hatchery identifies an available spawn and calls spawnCreep(). Crucially, it attaches the metadata to the memory object, including the overlordPID and the intended role.
2. **Phase II: The In-Utero Period (Tick N+1 to N+M):** The creep is physically spawning. During this time, the creep exists in Game.creeps but its spawning property is true. The Hatchery process tracks this name to prevent duplicate spawning.
3. **Phase III: The Delivery (Tick N+M+1):** The creep is fully formed. The Hatchery process detects !creep.spawning and executes the registration handshake. It calls the "Adopt" method on the requesting Overlord process, passing the creep's name and its initial state. The Overlord then adds the creep to its "Zerg" array and begins executing its task logic.

The Memory Cleanup Pitfall

A major challenge in TypeScript bots is the 1-tick delay in creep instantiation. When spawnCreep() returns OK, the memory entry is created immediately, but the Game.creeps object is not updated until the next tick. If the bot's memory cleanup script (which deletes

memory for dead creeps) runs between these two events, it will delete the memory of the creep currently being spawned.
The "Factory" pattern solves this by ensuring that the memory cleanup logic is aware of the Hatchery's state. The Hatchery maintains a pendingSpawns list in the heap (global scope). The cleanup script is instructed to ignore any memory keys present in the pendingSpawns list, ensuring that the handshake remains intact even during high-latency ticks.

Advanced Architectural Patterns: The Multi-Spawn Group

As the bot scales to RCL 7 and 8, it gains access to multiple spawns per room. A sophisticated Hatchery must treat these spawns as a single "SpawnGroup" rather than independent entities.

Parallelization and Bandwidth

A single spawn has a "bandwidth" of 1,500 ticks of production every 1,500 ticks. At RCL 8, a room has 4,500 ticks of production bandwidth. The SpawnGroup manager distributes the queue across all three spawns. This allows the bot to, for example, spawn a massive defender while simultaneously producing the replacement for a dying harvester.
The logic for parallelization must account for shared energy. If Spawn A and Spawn B both attempt to spawn a 5,000-energy creep in the same tick, but the room only has 6,000 total energy, the second call will fail with ERR_NOT_ENOUGH_ENERGY. The Hatchery must calculate the "Virtual Energy Balance"—subtracting the cost of currently active spawning intents from the total energyAvailable before initiating a new spawn.

Multi-Room Spawning

In the late game, the "Empire" or "Colony" level of the bot may decide to use one room’s Hatchery to fulfill a request for an adjacent room. This is common when a new room is being "bootstrapped" and lacks the energy capacity to build large workers. The Hatchery must be able to calculate the path distance between rooms and determine if the travel time is worth the benefit of a larger unit. This requires a global "Spawn Manager" that coordinates requests across multiple Hatchery instances.

Performance Optimization and CPU Management

In a game where every millisecond of processor time is a resource, the Hatchery must be architected for extreme CPU efficiency.

Intent Caching and the 0.2 CPU Floor

Every call to spawnCreep() that returns OK costs 0.2 CPU as an "intent". Repeatedly calling spawnCreep() on a spawn that is already busy or in a room with insufficient energy is a waste of CPU. The Hatchery avoids this by caching the state of each spawn. If spawn.spawning is true, the Hatchery skips all logic for that structure for the remainder of the tick.

Process Suspension in the Kernel

In a Kernel/Process model, a process can be "suspended" or "put to sleep". The Hatchery process can use this to its advantage. When a 50-part creep is spawning, the Hatchery knows it will take exactly 150 ticks to complete. It can set a "Wake-Up" timer in the Kernel for 150 ticks and suspend the registration process for that unit, saving the CPU cost of checking the spawn status every single tick.

Heap-First Memory and Serialization

By using a Heap-First memory model, the Hatchery keeps its Queue and its structural references in the JavaScript heap (global scope). This avoids the heavy CPU cost of parsing and stringifying the entire queue from the Memory object every tick. The Foundry only serializes its state to the permanent Memory segment every 10-100 ticks, or when a critical state change occurs (like a room being lost), providing a massive reduction in the bot's average CPU floor.

Conclusion: The Biological Imperative of the Foundry

The "Automated Foundry" is more than a simple script for building creeps; it is a sophisticated resource-allocation engine that balances the metabolic needs of the colony against the strategic ambitions of the empire. By implementing a priority-driven queue with deadlock prevention, the bot ensures that it can recover from even the most catastrophic failures. Through dynamic morphology, it optimizes every unit for its specific role and environment, ensuring that no energy is wasted. And through a robust Kernel-level handshake, it maintains the integrity of the command-and-control hierarchy.

As the Screeps world becomes increasingly competitive, the difference between success and failure often lies in the efficiency of these foundational systems. A Foundry that can produce more "part-ticks" per energy unit and per CPU millisecond will invariably out-expand and out-last its rivals. The architectural patterns described herein—Factory abstraction, Template Repetition, and Batch Logistics—form the blueprint for a spawning system capable of supporting a top-tier autonomous agent.

Through the integration of Power Creeps at high levels and the implementation of cross-room spawning groups, the Hatchery evolves from a local room structure into a distributed imperial utility. The ultimate goal of this architecture is to transform energy into influence, ensuring that the "Overmind" has the physical tools necessary to manifest its will across the Screeps world map. By adhering to these engineering principles, a developer can build a spawning system that is not only scalable and efficient but truly autonomous.

Works cited

1. Screeps #1: Overlord overload | Ben Bartlett, https://bencbartlett.com/blog/screeps-1-overlord-overload/ 2. Screeps Part 19 – Operating Systems - Adam Laycock, https://alaycock.co.uk/2017/09/screeps-part-19-operating-systems 3. Screeps #1: The Game Plan | Field Journal, https://jonwinsley.com/notes/screeps-game-plan 4. Great Filters - Screeps Wiki, https://wiki.screepspl.us/Great_Filters/ 5. Managing a spawn queue : r/screeps - Reddit, https://www.reddit.com/r/screeps/comments/5iw9yt/managing_a_spawn_queue/ 6. Automating

Base Planning in Screeps – A Step-by-Step Guide, https://sy-harabi.github.io/Automating-base-planning-in-screeps/ 7. Bunkers · bencbartlett/Overmind Wiki · GitHub, https://github.com/bencbartlett/Overmind/wiki/Bunkers 8. Control | Screeps Documentation, https://docs.screeps.com/control.html 9. Is it possible to build a new Spawn point within a claimed room? : r/screeps - Reddit, https://www.reddit.com/r/screeps/comments/hj2quh/is_it_possible_to_build_a_new_spawn_point_within/ 10. Find empty spawn in room with full extensions. | Screeps Forum, https://screeps.com/forum/topic/2973/find-empty-spawn-in-room-with-full-extensions 11. Screep custom spawn code - Reddit, https://www.reddit.com/r/screeps/comments/8pt9o4/screep_custom_spawn_code/ 12. bonzaiferroni/bonzAI-framework - GitHub, https://github.com/bonzaiferroni/bonzAI-framework 13. Working on my spawning code, could use some pointers : r/screeps - Reddit, https://www.reddit.com/r/screeps/comments/6plegd/working_on_my_spawning_code_could_use_some/ 14. Respawning - Screeps Documentation, https://docs.screeps.com/respawn.html 15. The SIMPLEST Screeps Tutorial - LearnCodeByGaming.com - Learn Code By Gaming, https://learncodebygaming.com/blog/the-simplest-screeps-tutorial 16. Creep Body Setup Strategies - Screeps Wiki, https://wiki.screepspl.us/Creep_body_setup_strategies/ 17. Create better creeps : r/screeps - Reddit, https://www.reddit.com/r/screeps/comments/7ow0ns/create_better_creeps/ 18. Creeps | Screeps Documentation, https://docs.screeps.com/creeps.html 19. Creep | Screeps Wiki | Fandom, https://screeps.fandom.com/wiki/Creep 20. Compact Extension Arrays | Screeps Forum, https://screeps.com/forum/topic/136/compact-extension-arrays 21. How do you manage extensions? : r/screeps - Reddit, https://www.reddit.com/r/screeps/comments/5vtnnj/how_do_you_manage_extensions/ 22. Need some help improving my CPU usage | Screeps Forum, https://screeps.com/forum/topic/2381/need-some-help-improving-my-cpu-usage 23. Power Creeps update | Screeps Forum, https://screeps.com/forum/post/10160 24. Power | Screeps Documentation, https://docs.screeps.com/power.html 25. Creeps spawning without memory | Screeps Forum, https://screeps.com/forum/topic/942/creeps-spawning-without-memory 26. Creep memory object inconsistent? | Screeps Forum, https://screeps.com/forum/topic/2547/creep-memory-object-inconsistent 27. Is there a spawn queue? - screeps - Reddit, https://www.reddit.com/r/screeps/comments/4w02wz/is_there_a_spawn_queue/ 28. room.energyCapacityAvailable returns null due to corrupted spawn store | Screeps Forum, https://screeps.com/forum/topic/2829/room-energycapacityavailable-returns-null-due-to-corrupted-spawn-store 29. Simultaneous execution of creep actions - Screeps Documentation, https://docs.screeps.com/simultaneous-actions.html 30. CPU - Screeps Wiki, https://wiki.screepspl.us/CPU/ 31. First! And the Screeps Persistence Model, https://screeping.wordpress.com/2016/12/08/first-and-the-screeps-persistence-model/ 32. Optimization - Screeps Wiki, https://wiki.screepspl.us/Optimization/