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