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/