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