The Architecture of Persistent Observability: Advanced Debugging and Telemetry Strategies in Distributed Screeps Systems

The operational environment of Screeps: World is fundamentally distinct from contemporary real-time strategy games due to its persistence and the requirement for total automation. In a 24/7 simulation where the game engine executes logic once per tick—typically every 2.7 to 5 seconds—the state of a player's colony can undergo catastrophic shifts during the six to eight-hour intervals between active human monitoring sessions.[1] For top-tier players, debugging is not merely a reactive process of fixing syntax errors; it is an engineering discipline centered on the creation of high-fidelity observability pipelines that capture, persist, and visualize state transitions across long timeframes. The challenge is compounded by the game's volatile console and the inherent limitations of the browser-based client, which cannot maintain a historical record of events once a window is closed or refreshed.[3] Consequently, the development of a resilient "black box" logging system and the integration of external telemetry suites like Grafana and Elasticsearch have become prerequisites for survival and optimization in the Screeps MMO environment.[5]

Persistent Storage Mechanisms and the Economics of In-Game Memory

The primary obstacle to effective long-term debugging is the volatility of the default console. While console.log() is the standard instrument for immediate feedback, its output is ephemeral and limited by the browser's buffer.[3] When a player returns after a six-hour absence, approximately 7,200 ticks of history have likely been lost if they relied solely on the web interface. To solve this, expert players utilize the game's internal data structures—Memory and RawMemory—to create persistent logs that survive both browser closures and global resets.[4]

The CPU Costs of Serialization and JSON Parsing

The Memory object in Screeps is a persistent key-value store limited to $2\ \mathrm{MB}$ of data, which is automatically stringified and parsed via JSON.parse() on the first access during each tick.[7] For high-level players, the CPU cost of this parsing is a significant concern. As the memory size approaches the $2\ \mathrm{MB}$ limit, the time taken to deserialize the object can consume a substantial portion of the player's tick-limit, especially on lower-GCL accounts.[7]

Storage Type	Capacity	Parsing Behavior	Optimal Use Case
Global Object	Unlimited (RAM)	None; volatile across resets	Caching cost matrices, class instances [8]
Memory	$2\ \text{MB}$	Automatic JSON.parse every tick	High-frequency flags, persistent state [7]
RawMemory Segments	$10\ \text{MB}$ ($100 \times$)	Explicit access required	Historical logs, base plans, stats [9]
interShardSegment	Variable	Asynchronous inter-shard data	Communicating between shards [10]

The economic trade-off between persistence and CPU efficiency drives top players toward RawMemory segments. By partitioning data into $100\ \text{KB}$ segments, players can avoid the overhead of parsing the entire $2\ \text{MB}$ Memory object every tick.[8] A common strategy involves using specific segments for different categories of diagnostic data, ensuring that only the segments required for a specific operation are requested using RawMemory.setActiveSegments().[10]

Implementation of the In-Memory Circular Buffer

To manage logs over a six-hour window without exceeding storage limits, top players implement circular buffers or "flight recorders" within RawMemory.[11] This pattern involves writing log entries to a segment until it nears its $100\ \text{KB}$ capacity, at which point the script either rotates to the next available segment or begins overwriting the oldest entries. This ensures that the most recent several thousand ticks of data—including the lead-up to any failure—are always available for retrospective analysis.[6]

The structure of such a buffer typically includes a metadata header containing a pointer to the current write head and the total count of logs.[11] By utilizing a structured logging approach where each entry is a JSON object containing a timestamp, a severity level, and a context object, players can later reconstruct the exact state of the colony.[13]

External Telemetry: The Grafana and InfluxDB Pipeline

While in-game memory is suitable for critical error logs, it is ill-equipped to handle the high-volume time-series data required to diagnose long-term economic or defensive trends.[4] Top-tier players almost exclusively offload this data to external databases using the Screeps API.[14] The most prevalent stack for this purpose is the combination of Grafana for visualization and InfluxDB or Graphite for data storage.[5]

The Agent-Based Ingestion Architecture

Because providing a Screeps password to a third-party service is a security risk, the community has standardized on an agent-based approach.[14] A small program, known as a "Screeps Agent," runs on the player's local hardware—such as a Raspberry Pi or a cloud-hosted VPS—and polls the Screeps API using a secure authentication token.[5]

Agent Component	**Responsibility**	**Performance Impact**
Game Script	Writes stats to Memory.stats or a segment	Minimal (0.1–0.5 CPU) [6]
Screeps API	Exposes endpoints for data retrieval	Rate-limited by official server [19]
Local Agent	Retrieves stats and pushes to InfluxDB	CPU-intensive on host, not in-game [14]
InfluxDB	Stores time-stamped metrics for years	Zero in-game impact [16]
Grafana	Queries InfluxDB to render dashboards	Zero in-game impact [16]

High-level players often utilize services like ScreepsPlus, which provides a hosted Grafana instance and a standardized agent configuration.[14] This system allows players to monitor variables such as GCL growth rates, CPU bucket levels, and market balances over periods of days or weeks.[1] If a player checks their dashboard after six hours and sees a sharp drop in energy levels, they can correlate that timestamp with in-game error logs to identify the exact moment a harvester failed or an invader appeared.[4]

External API Rate Limits and Scoping

Developing a robust external logging system requires adherence to the official Screeps API rate limits.[19] When using authentication tokens, top players must optimize their polling frequency to avoid being throttled.[15]

Endpoint	Max Frequency (Authenticated)	Typical Usage in Logging
GET /api/user/memory	1440 requests / day	High-level status updates [19]
POST /api/user/memory	240 requests / day	Sending commands to the bot [19]
GET /api/user/memory-segment	360 requests / hour	Pulling detailed historical logs [19]
POST /api/user/memory-segment	60 requests / hour	Updating base plans or persistent state [19]

By utilizing RawMemory segments for stats, players can pull $100\ \text{KB}$ of diagnostic data every 10 seconds without affecting their primary memory rate limit, allowing for near-real-time observability outside the game client.[9]

Advanced Error Handling and Stack Trace Management

In a long-term autonomous system, unhandled exceptions are catastrophic.[19] If a script crashes early in the tick, it may fail to register intents (like attacks or repairs), leaving the colony defenseless for the duration of the player's absence.[19] Top players mitigate this by wrapping their entire execution loop in a try...catch block and utilizing the Error.stack property.[4]

Capture and Serialization of the Stack

A common mistake in beginner debugging is logging only the error message, which often yields vague strings like TypeError: Cannot read property 'pos' of undefined.[22] Expert systems instead capture the full stack trace, which provides the file name and line number where the failure occurred.[19] This information is then serialized and pushed to a persistent log in Memory or RawMemory.[4]

Because the Error object itself is not directly serializable to JSON, players must manually extract the stack string.[19] A typical high-level implementation might look as follows:

```
JavaScript

try {
  mainLoop();
} catch (error) {
  let errorLog = {
    tick: Game.time,
    message: error.message,
    stack: error.stack,
    context: "Global Loop"
  };
  Memory.errors.push(errorLog);
  console.log('<font color="red">' + error.stack + '</font>');
}
```

This ensures that even if the player's browser is closed, the error is preserved in the server-side Memory.[4] Upon returning to the game, the player can inspect Memory.errors to see exactly what went wrong during their absence.[4]

Managing the Memory Overhead of Logs

Writing logs to Memory introduces the risk of exceeding the $2\ \mathrm{MB}$ limit.[8] If a recurring error generates thousands of log entries, the script will eventually fail to save its state, potentially causing a crash-loop.[4] Top-tier players prevent this by implementing log rotation and pruning.[4] They often limit the Memory.errors array to 20 or 50 entries, shifting out the oldest data as new errors arrive.[4] For more extensive logging, they offload the data to RawMemory segments where the $10\ \mathrm{MB}$ capacity allows for much larger volumes of diagnostic information.[6]

Global Reset Tracking and State Reconstitution

A unique challenge in the Screeps environment is the "Global Reset," which occurs when the server migrates a player's script to a new Virtual Machine (VM).[4] This event wipes all variables stored in the global scope, though Memory and RawMemory persist.[8] For bots that rely on complex, in-memory caches to save CPU—such as pathfinding cost matrices or room analysis data—a global reset can cause a massive CPU spike as the bot attempts to rebuild its state.[8]

Detecting and Logging Resets

Top players monitor for global resets to determine if an overnight failure was caused by a script crash or a server-side migration.[4] By initializing a global.resetTick variable, the script can detect when it is running on a fresh VM:

JavaScript

```javascript
if (typeof global.wasReset === 'undefined') {
  global.wasReset = true;
  global.resetTick = Game.time;
  console.log("Global Reset detected at tick: " + Game.time);
}
```

Logging these resets to an external dashboard allows players to visualize the frequency of VM migrations.[4] If a bot consistently fails immediately following a global reset, it indicates that the initialization logic—such as rebuilding class hierarchies or re-loading data from segments—is flawed or too CPU-intensive.[8]

The Build and Refresh Pattern

The "Overmind" codebase, a prominent community bot, utilizes a "build and refresh" architecture to handle these resets gracefully.[1] Instead of re-instantiating every object every tick, the bot keeps class instances in the global scope and "refreshes" their references to game objects (like creeps or structures) which are updated by the engine every tick.[23] Every 20 ticks, or upon a global reset, the bot performs a full "build" to ensure no stale data remains in the cache.[23] This architecture includes internal validation and checksums to ensure that the persistent state remains consistent across resets.[23]

Visual Debugging: RoomVisuals and MapVisuals

While text logs provide historical context, they are often insufficient for diagnosing spatial or geometric failures, such as poor base planning or inefficient pathing.[19] To bridge this gap, high-level players utilize the RoomVisuals and MapVisuals APIs to draw diagnostic overlays directly onto the game world.[19]

Dynamic Visual Overlays

Expert scripts often include a "debug mode" that can be toggled via Memory.[27] When active, the bot draws lines to represent pathing intent, circles to highlight intended construction sites, and labels to display the internal state machines of individual creeps.[19]

Visual Element	**Diagnostic Purpose**	**Performance Consideration**
Path Lines	Identifying bottlenecks or inefficient routing	High CPU; toggle via memory flag [3]
State Labels	Viewing creep role and current task (e.g., STATE_MINING)	Low CPU; useful for real-time monitoring [28]
CostMatrix Heatmaps	Visualizing pathfinding weights and obstacles	High CPU; draw only on request [8]
Territory Overlays	Monitoring expansion planning and influence zones	Map-level visuals; low per-room cost [5]

Because these visuals are only visible when the player is actively looking at a room, they do not need to be stored long-term.[19] However, some players take "state snapshots" of these visuals and store them as serialized strings in RawMemory segments.[6] This allows them to effectively "rewind" the visual state of a room to the moment an error occurred, even if they were offline at the time.[4]

External Automation and CI/CD for Robust Deployments

Top-tier players recognize that many overnight failures are the result of deploying "half-done" code that works in simulation but fails under real-world conditions like CPU limits or global resets.[2] To mitigate this, they implement sophisticated build pipelines using tools like Grunt, Gulp, or Rollup.[1]

Build-Time Validation and Checksums

Codebases like Overmind use Rollup to bundle TypeScript into a single main.js file, incorporating checksums for internal validation.[1] This "Assimilator" module ensures that if the code is tampered with or if a deployment fails partially, the bot can detect the integrity violation and alert the player via external notifications.[23]

Tooling Component	**Role in Debugging and Reliability**	**Deployment Impact**
TypeScript / Kotlin	Provides static typing and compile-time error checking	Prevents common runtime TypeErrors [5]
Grunt / Gulp	Automates the upload of code to multiple shards	Ensures consistency across the MMO world [5]
Rollup	Bundles modules and performs tree-shaking	Reduces code size and improves load times [1]
Jest / Mocha	Unit tests logic for base planning and pathing	Catches regressions before deployment [5]

By utilizing screeps-server-mockup or other private server packages for unit testing, players can simulate edge cases—such as all creeps dying or the CPU bucket being empty—before pushing code to the public server.[4] This "pre-emptive debugging" reduces the likelihood of encountering an unhandled error during a long absence.[4]

Community Services and Multi-Player Intelligence

In the high-level Screeps ecosystem, debugging is often a collaborative or competitive endeavor.[26] Players use the #client-abuse and #world-help channels on Discord to share snippets for advanced console formatting, such as adding buttons or links to the in-game console using HTML.[19]

Public Memory Segments and Diplomacy

RawMemory also facilitates communication between players.[9] By setting specific segments as "public," players can expose their bot's status, diplomatic stance, or even a shared "master ledger" of directives for an alliance.[10] This allows an ally to "debug" a teammate's failure by checking their public segment for reports of an attack or an economic crisis.[24]

The RawMemory.setActiveForeignSegment() method allows a script to read another player's public data, enabling cross-player observability.[10] This is particularly useful for debugging inter-player interactions, such as terminal trades or power-bank raiding coordination, where one player's code must respond to the state of another's.[10]

Metrics for the Long Haul: What to Track

When a player is away for six hours, they need their telemetry to tell a story.[13] Simply tracking "total energy" is insufficient.[4] Top-tier Grafana dashboards are designed with "Information Density" in mind, grouping metrics to highlight causal relationships.[13]

Metric Category	**Specific Data Points**	**Diagnostic Value**
CPU Health	bucket, used, limit, getUsedCPU()	Identifying infinite loops or script timeouts [4]
Economic Flow	energy/tick, terminal.store, market.price	Finding inefficiencies in the supply chain [1]
Population	count(role), ticksToLive(min), spawn.queue	Detecting "colony death" before it happens [4]
GCL Progress	progress/tick, level, ticksToNextLevel	Monitoring long-term bot performance [2]
Combat Intel	room.defense, damage/tick, invader.count	Reconstructing battle events post-mortem [4]

By using "fingers_crossed" logging—where the system logs everything but only emails or alerts the player if a specific threshold is crossed—players can maintain a high-level view without being overwhelmed by data.[35] For example, a script might only call Game.notify() if the CPU bucket drops below 1000 or if an RCL 8 room's controller is downgraded, providing a timestamped record of the crisis.[4]

Performance Profiling and Bottleneck Identification

Long-term debugging also involves identifying code that is "slowly" failing.[5] A script might run fine for hours, but a gradual memory leak or an $O(N^2)$ algorithm that scales with the number of rooms can eventually lead to a CPU exhaustion event.[2]

Community Profilers

Top players use community-built profilers like screeps-profiler (for JavaScript) or specialized tools for TypeScript and Rust to find these bottlenecks.[19] These tools wrap game functions and track the average CPU cost per call over thousands of ticks.[19]

Profiler	Language Support	Key Feature
gdborton/screeps-profiler	JavaScript	Function wrapping and average CPU reporting [19]
screepers/screeps-typescript-profiler	TypeScript	Type-safe performance monitoring [19]
banan	Multiple	Generates flame graphs for visual analysis [19]
slothsoft/screeps-script	Multiple	Built-in "fancy" overview and per-role CPU logs [28]

By running a profiler for a one-hour window and reviewing the results in the console (often formatted as a table using HTML), a player can identify which specific creep role or room-management module is consuming the most resources.[19] This allows for targeted optimization that prevents "overnight" failures caused by the server's hard CPU limits.[8]

Conclusion: The Expert Debugging Workflow

The ability of top-tier Screeps players to debug issues over a six-hour absence is the result of a multi-layered observability strategy that transforms the game from a black box into a transparent, data-driven system.[5] This workflow is characterized by:

1. **Immediate Persistence:** Errors are never just printed; they are caught, enriched with stack traces, and stored in RawMemory segments to ensure they survive global resets and browser closures.[4]
2. **External Telemetry:** High-volume statistics are offloaded to Grafana via agents, providing a historical time-series view that allows for the correlation of economic trends with discrete failure events.[1]
3. **Visual Context:** RoomVisuals and state snapshots provide a geometric understanding of the bot's logic, allowing players to "see" why a pathfinder failed or a base layout was blocked.[19]
4. **Proactive Reliability:** Build pipelines and unit testing suites catch errors before they reach the production server, while global reset tracking ensures that the bot's caches remain consistent over long intervals.[5]

By synthesizing these tools and techniques, top-tier players ensure that even when they are asleep, their code is meticulously recording its own history, ready to be audited and optimized at the next check-in.[1] In the competitive environment of Screeps, observability is not just a debugging tool; it is a fundamental component of the AI's evolutionary process.[23]

Works cited

1. Overmind/README.md at master · bencbartlett/Overmind · GitHub, accessed February 21, 2026, https://github.com/bencbartlett/Overmind/blob/master/README.md
2. Everything takes a long time : r/screeps - Reddit, accessed February 21, 2026, https://www.reddit.com/r/screeps/comments/6dz3xn/everything_takes_a_long_time/
3. Debugging | Screeps Documentation, accessed February 21, 2026, https://docs.screeps.com/debugging.html
4. How do you debug problems that happen overnight? : r/screeps, accessed February 21, 2026, https://www.reddit.com/r/screeps/comments/8liega/how_do_you_debug_problems_that_happen_overnight/
5. Third Party Tools | Screeps Documentation, accessed February 21, 2026, https://docs.screeps.com/third-party.html
6. screepers/screeps-stats: Access Screeps Console, Performance, and Statistics Data via Kibana and ElasticSearch - GitHub, accessed February 21, 2026, https://github.com/screepers/screeps-stats
7. Global Objects | Screeps Documentation, accessed February 21, 2026, https://docs.screeps.com/global-objects.html
8. Caching Overview | Screeps Documentation, accessed February 21, 2026, https://docs.screeps.com/contributed/caching-overview.html
9. Memory - Screeps Wiki, accessed February 21, 2026, https://wiki.screepspl.us/Memory/
10. docs/api/source/RawMemory.md at master · screeps/docs - GitHub, accessed February 21, 2026, https://github.com/screeps/docs/blob/master/api/source/RawMemory.md
11. Lightweight In-Memory Logging - Preshing on Programming, accessed February 21, 2026, https://preshing.com/20120522/lightweight-in-memory-logging/
12. Saving and accessing simple object to Memory - code help : r/screeps - Reddit, accessed February 21, 2026, https://www.reddit.com/r/screeps/comments/1fy2fu2/saving_and_accessing_simple_object_to_memory_code/
13. What tips do you have for logging? And by logging, I mean really good logging. - Reddit, accessed February 21, 2026, https://www.reddit.com/r/webdev/comments/tj1f26/what_tips_do_you_have_for_logging_and_by_logging/
14. Grafana · ScreepsPlus, accessed February 21, 2026,

https://screepspl.us/services/grafana/
15. screepers/node-screeps-api - GitHub, accessed February 21, 2026, https://github.com/screepers/node-screeps-api
16. Grafana Guide - InfluxDB, accessed February 21, 2026, https://www.influxdata.com/grafana/
17. Get started with Grafana and InfluxDB, accessed February 21, 2026, https://grafana.com/docs/grafana/latest/fundamentals/getting-started/first-dashboards/get-started-grafana-influxdb/
18. ScreepsPlus/node-agent - GitHub, accessed February 21, 2026, https://github.com/ScreepsPlus/node-agent
19. Basic Debugging - Screeps Wiki, accessed February 21, 2026, https://wiki.screepspl.us/Basic_debugging/
20. screeps-api 0.6.0 - Docs.rs, accessed February 21, 2026, https://docs.rs/crate/screeps-api/latest
21. Stack Trace, Error, and Try/Catch: How to Handle Errors Intelligently in JavaScript - Medium, accessed February 21, 2026, https://medium.com/@aninhabort/stack-trace-error-and-try-catch-how-to-handle-errors-intelligently-in-javascript-563d659b98af
22. Mysterious error without stack trace | Screeps Forum, accessed February 21, 2026, https://screeps.com/forum/topic/1718/mysterious-error-without-stack-trace
23. Releases · bencbartlett/Overmind - GitHub, accessed February 21, 2026, https://github.com/bencbartlett/Overmind/releases
24. Screeps #6: Verifiably refreshed | Ben Bartlett, accessed February 21, 2026, https://bencbartlett.com/blog/screeps-6-verifiably-refreshed/
25. Automating Base Planning in Screeps – A Step-by-Step Guide, accessed February 21, 2026, https://sy-harabi.github.io/Automating-base-planning-in-screeps/
26. Community Communication - Screeps Wiki, accessed February 21, 2026, https://wiki.screepspl.us/Community_Communication/
27. sscholl/screeps-debug: This is a debug module for screeps.com. - GitHub, accessed February 21, 2026, https://github.com/sscholl/screeps-debug
28. My scripts for the game Screeps. - GitHub, accessed February 21, 2026, https://github.com/slothsoft/screeps-script
29. Current action of a creep... : r/screeps - Reddit, accessed February 21, 2026, https://www.reddit.com/r/screeps/comments/8ha6wr/current_action_of_a_creep/
30. What causes scripts to stop working... with no errors? :: Screeps: World Help, accessed February 21, 2026, https://steamcommunity.com/app/464350/discussions/5/3223871682614026540/
31. bencbartlett/Overmind: AI for Screeps, a multiplayer programming strategy game - GitHub, accessed February 21, 2026, https://github.com/bencbartlett/Overmind
32. Useful Tools - ScreepsPlus, accessed February 21, 2026, https://screepspl.us/useful-tools/
33. Best way to get started? : r/screeps - Reddit, accessed February 21, 2026, https://www.reddit.com/r/screeps/comments/1knb7yw/best_way_to_get_started/
34. Public memory segments | Screeps Forum, accessed February 21, 2026,

https://screeps.com/forum/topic/122/public-memory-segments
35. Logging best practices : r/PHP - Reddit, accessed February 21, 2026, https://www.reddit.com/r/PHP/comments/kxf0jg/logging_best_practices/