Shard 3 "Provident" Architecture: Implementation Guide

Core Philosophy

Target: 20 CPU Hard Limit. **Constraint:** Logic must be cheaper than the `JSON.parse` cost of storing it. **Method:** Functional Programming > Object Oriented Programming.

Phase 1: The Foundation (Survival)

Goal: Reach RCL 4 on < 5 CPU using the "Big Creep" theory.

1.1 TypeScript Memory Typing (The Backbone)

Before writing logic, define the shape of your data. This prevents "hidden" memory bloat.

Challenge: Do not extend the `Creep` class. Define an interface for the memory instead.

```
// src/types.d.ts
interface CreepMemory {
  role: 'miner' | 'hauler' | 'upgrader' | 'fastfiller' | 'roadwarrior'; // Added roadwc
  room: string;
  working: boolean;
  // The "Brain" of the creep is just this task object
  task?: {
    id: Id<Structure | Resource>;
    type: 'transfer' | 'withdraw' | 'harvest' | 'upgrade' | 'repair';
    targetPos?: RoomPosition; // For caching movement
  };
}

interface RoomMemory {
  // Cache IDs, not objects, to save CPU
  sourceIds: Id<Source>[];
  containerIds: Id<StructureContainer>[];
  // Timestamps for cache invalidation
  cacheValidUntil: number;
  // Remote Intel
  roadHealth?: number; // Average road hits (0-1)
}
```

1.2 The Simple Loop (No Kernel Yet)

Don't over-engineer an Operating System yet. Use a functional loop.

1. **Garbage Collection:** Delete memory of dead creeps every 100 ticks (not every tick, save CPU).
2. **Init Phase:** Run `Room.init()` to scan sources if `cacheValidUntil` is expired.

3. **Role Execution:** Iterate through `Game.creeps`. Switch on `creep.memory.role` and pass the creep to a handler function (e.g., `runMiner(creep)`).

1.3 The "Big Miner" Logic

Math: 1 Creep with 6 WORK parts costs 0.2 CPU to harvest.

- **Why 6?** A source requires 5 WORK parts to harvest fully (10 energy/tick). The 6th part acts as a **Repair Buffer**.
- **The CPU Arbitrage:** This eliminates the **Dedicated Container Repairer**. The miner repairs the container (which decays fast) with zero movement cost.
- **What about Roads?** We let **Towers** repair roads. Spawning a creep to walk to a road costs move-CPU. A tower shooting a road costs 0.2 CPU flat.

Implementation:

- Static Mining: The miner moves *once* to the container.
- Check: `if (creep.pos.isEqualTo(containerPos))`
- Logic:
  - If container health < 90%: `creep.repair(container)`
  - Else: `creep.harvest()`
- If not at pos: `creep.moveTo(containerPos)`

Phase 2: The Efficiency Engine (The Dispatcher)

Goal: Solve the Logistics $O(N^2)$ problem. Reduce CPU usage by 60%.

2.1 The Dispatcher Pattern

Instead of haulers asking "Where do I go?", the Room Manager calculates needs once and assigns them.

Architecture:

1. **Scan:** Iterate room structures *once*.
   - Push empty extensions/spawn to `demandQueue`.
   - Push full containers/dropped resources to `supplyQueue`.
2. **Match:**
   - Sort `demandQueue` by priority (Spawn > Extension > Tower).
   - Find idle haulers (`!creep.memory.task`).
   - Assign the highest priority demand to the closest idle hauler.
3. **Write:** Directly modify the hauler's memory:

```
    creep.memory.task = {
      id: target.id,
      type: 'transfer'
    };
```

2.2 Hauler "Dumb" Execution

The hauler code becomes incredibly simple (and fast):

```
function runHauler(creep: Creep) {
  if (!creep.memory.task) return; // Idle, wait for Dispatcher

  const target = Game.getObjectById(creep.memory.task.id);
  if (!target) {
     delete creep.memory.task; // Task invalid, request new one next tick
     return;
  }

  // Execute without thinking
  if (creep.transfer(target, RESOURCE_ENERGY) === ERR_NOT_IN_RANGE) {
     creep.moveTo(target);
  }
}
```

Phase 3: Autonomy & caching

Goal: Flag-based control and resilience against Global Resets.

3.1 Global vs. Memory

The Trap: Storing everything in `Memory` (serialized to JSON) costs CPU. **The Fix:** Store heavy data in the Heap (`global`).

Implementation: Create a `Cache` object in the global scope.

```
// global.d.ts
declare var Cache: {
  costMatrices: { [roomName: string]: CostMatrix };
  paths: { [pathKey: string]: string }; // Serialized paths
}
```

Rehydration: At the start of the tick, check if `Cache` exists. If undefined (Global Reset happened), rebuild it using minimal data from `Memory` .

3.2 The Flag Directive System

Use colored flags to trigger logic without writing code.

- **Purple Flag:** `runRemoteMining(flag)`
- **Red Flag:** `runOffense(flag)`

Logic: The main loop iterates `Game.flags`. `if (flag.color === COLOR_PURPLE)` -> Trigger the Remote Mining module for that room. `if (flag.color === COLOR_RED)` -> Spawn a duo and send to `flag.pos.roomName`.

3.3 Tower Logic (Hysteresis)

Don't repair ramparts every tick (jittering).

Algorithm:

1. Find ramparts < 5,000 hits.
2. Repair them until 25,000 hits.
3. **Roads:** Repair roads if hits < 50%. (Towers handle this, not creeps).
4. **Crucial:** Store the `targetId` in the tower/repairer memory so they don't re-search for a target every tick. Stick to one target until it meets the threshold.

3.4 Remote Infrastructure (The "Road Warrior")

Problem: Remote rooms have no towers. **Trap:** Giving Haulers `WORK` parts adds CPU overhead to every hauler tick. **Solution:** Periodic Maintenance.

1. **Swamps Only:** Do not pave plains in remote rooms. Only pave swamps (5:1 movement ratio).
2. **The Road Warrior:**
   - Check remote room road health every ~1000 ticks.
   - If health < 60%: Spawn **one** `roadwarrior` creep (Work/Carry/Move).
   - It repairs the path once and recycles itself.
   - **Benefit:** Concentrates CPU cost into a short burst rather than a constant tax.

Development Checklist for Taylor

1. [ ] **Setup:** Initialize a TS project with `rollup` (efficient bundling).
2. [ ] **Typing:** Define the `CreepMemory` interface to forbid random property additions.
3. [ ] **The Loop:** Write a loop that spawns 1 miner and logs CPU usage.