ENTROPICA FORENSIC MODEL — APPENDIX N

Adaptive Spawn Governance

Autonomous Parameter Calibration for Bounded Autonomy

Version 1.2

Yology Research Division
Entropica SPC

December 2025

Abstract

Appendix N specifies the **Adaptive Spawn Governor (ASG)**, an autonomous parameter calibration system that prevents spawn brittleness through continuous adaptation. Rather than hardcoding spawn parameters (τ_{spawn}, D_{max}, R_{max}), the ASG observes swarm health metrics and adjusts parameters within profile-defined bounds. This enables self-healing behavior: a misconfigured or stressed swarm recovers within 50,000 ticks without human intervention, while preserving Gardener override authority (Commandment 4) through a rollback window. The ASG integrates with Appendix K (SHSL health metrics), Appendix I (deployment profiles), and Appendix M (Discovery Stack predictions).

Contents

1 Introduction

1.1 The Problem: Static Parameters

Volume I v1.6 defines spawn conditions S_1–S_6 with hardcoded parameters:

Parameter	Default	Function
τ_{spawn}	0.7	Minimum health to spawn
D_{max}	10	Maximum lineage depth
R_{max}	100/tick	Global spawn rate limit
$R_{local,max}$	10/window	Per-parent spawn rate

Table 1: Static spawn parameters (Volume I §3.4).

Failure scenario:

1. HFT swarm spawns at 85/tick (within R_{max}=100)
2. Resource contention causes health drop: $0.75 \rightarrow 0.68 \rightarrow 0.62$
3. Health falls below $\tau_{spawn} = 0.7$
4. **All spawning blocked**—system paralyzed
5. Requires human intervention to adjust parameters

1.2 The Solution: Adaptive Calibration

> **Adaptive Spawn Governor**
>
> The Adaptive Spawn Governor (ASG) automatically calibrates parameters based on observed swarm behavior:
>
> - Runs every $T_{calibration} = 10,000$ ticks
> - Reads metrics from Appendix K (SHSL)
> - Adjusts parameters within profile bounds (Appendix I)
> - Logs all adjustments to d-CTM (immutable audit)
> - Gardener can rollback within $T_{review} = 5,000$ ticks

1.3 Design Principles

1. **Bounded Adaptation:** Parameters never exceed profile bounds

2. **Conservative Adjustment:** Faster to constrain, slower to loosen

3. **Human Override:** Gardener rollback authority preserved

4. **Full Accountability:** All adjustments logged to d-CTM

5. **Hysteresis:** Cooldown prevents oscillation

2 Architecture

2.1 Component Overview
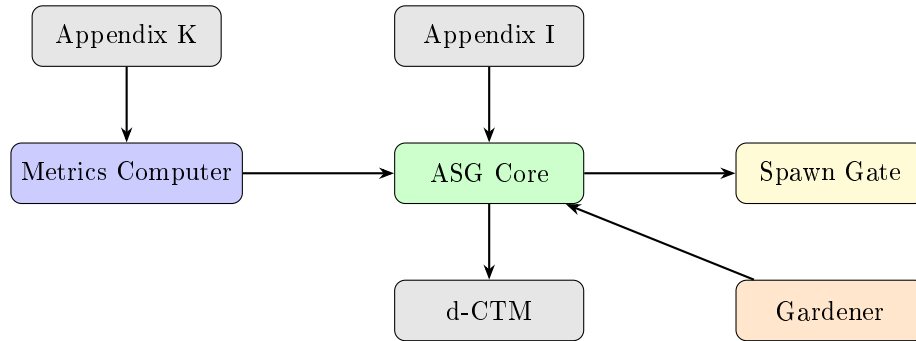
2.2 Integration Points

Component	Interaction	Reference
Appendix K (SHSL)	ASG reads health metrics	§3.1
Appendix I (Profiles)	ASG reads bounds	§4
Appendix M (Discovery)	ASG receives predictions	§5.4
Appendix L (Judicial)	Fork handling	§5.3
d-CTM	Adjustment logging	§3.4
Gardener	Rollback authority	§5.2

3 Metrics Computation

3.1 SwarmMetrics Data Structure

Definition 3.1 (SwarmMetrics). The metrics computed every calibration cycle:

```
@dataclass
class SwarmMetrics:
    avg_health: float          # Average health [0.0, 1.0]
    health_std_dev: float      # Health variance
    health_trend: float        # Change over last 10K ticks
    spawn_rate: float          # Capsules/tick (last 1000 ticks)
    vault_utilization: float   # Vault capacity used [0.0, 1.0]
    avg_depth: float           # Average lineage depth
    max_depth: int             # Maximum depth observed
    active_capsule_count: int  # Current active capsules
    anomaly_rate: float        # % with ANOMALY flags
```

3.2 Metrics Computation Algorithm

Listing 1: MetricsComputer.compute_metrics()

```
def compute_metrics(self, swarm_state) -> SwarmMetrics:
    """Extract metrics for calibration. O(n), <100ms for 10K capsules."""

    active_capsules = swarm_state.get_active_capsules()

    # Health metrics
    health_scores = [c.health for c in active_capsules]
    avg_health = statistics.mean(health_scores)
```

```
health_std_dev = statistics.stdev(health_scores) if len(health_scores) > 1
    else 0.0
health_trend = avg_health - swarm_state.get_historical_health(ticks_ago
    =10000)

# Spawn rate (last 1000 ticks)
spawn_events = swarm_state.get_spawn_events(window=1000)
spawn_rate = len(spawn_events) / 1000.0

# Vault utilization
total_capacity = swarm_state.vault.total_capacity
allocated = sum(c.vault_allocation for c in active_capsules)
vault_utilization = allocated / total_capacity

# Depth metrics
depths = [c.depth for c in active_capsules]
avg_depth = statistics.mean(depths)
max_depth = max(depths)

# Anomaly rate
anomalous = sum(1 for c in active_capsules if c.has_anomaly_flag())
anomaly_rate = anomalous / len(active_capsules)

return SwarmMetrics(...)
```

3.3   Metrics Validation

Before calibration proceeds, metrics are validated:

Listing 2: Metrics validation

```python
def validate_metrics(self, metrics: SwarmMetrics) -> Tuple[bool, str]:
    checks = []
    if not (0.0 <= metrics.avg_health <= 1.0):
        checks.append(f"Invalid avg_health: {metrics.avg_health}")
    if metrics.spawn_rate < 0:
        checks.append(f"Invalid spawn_rate: {metrics.spawn_rate}")
    if not (0.0 <= metrics.vault_utilization <= 1.0):
        checks.append(f"Invalid vault_utilization: {metrics.vault_utilization}"
            )
    if checks:
        return (False, "; ".join(checks))
    return (True, "All metrics valid")
```

3.4   d-CTM Logging

All metrics are logged to d-CTM for forensic reconstruction:

```
log_to_dctm(
    event_type='ASG_METRICS',
    tick=current_tick,
    metrics=metrics,
    zk_sp_proof=generate_proof(metrics)
)
```

3.4.1   Rich Observability (Gap 8)

For operator debugging and trust, ASG logs comprehensive diagnostic context:

Listing 3: Rich diagnostic logging

```python
def _log_calibration_with_diagnostics(self, adjustment, metrics, swarm_state):
    """Generate rich diagnostic context for operators."""

    diagnostics = {
        # Why did we adjust?
        'trigger': {
            'avg_health': metrics.avg_health,
            'health_threshold': 0.65,
            'spawn_rate': metrics.spawn_rate,
            'spawn_threshold': 0.8 * self.R_max,
            'vault_utilization': metrics.vault_utilization
        },

        # Which capsules are problematic?
        'top_spawners': swarm_state.get_top_spawners(n=10),
        'unhealthy_capsules': swarm_state.get_unhealthy(threshold=0.6, limit
            =20),

        # What do we expect to happen?
        'predicted_outcome': {
            'expected_health_in_10K': self._predict_health(metrics),
            'expected_recovery_ticks': self._estimate_recovery(metrics)
        },

        # Historical context
        'recent_adjustments': self.adjustment_history[-5:],
        'parameter_trajectory': self._get_param_trajectory(window=50000)
    }

    log_to_dctm(
        event_type='ASG_CALIBRATION',
        adjustment=adjustment,
        diagnostics=diagnostics,
        zk_sp_proof=generate_proof(adjustment, diagnostics)
    )
```

This enables operators to answer:

- **WHY** did ASG adjust? (trigger conditions)
- **WHICH** capsules are problematic? (top spawners, unhealthy list)
- **WHEN** will health improve? (predicted recovery time)
- **HOW** have parameters changed? (trajectory)

4 Profile Bounds

4.1 Bounded Adaptation

ASG operates within **hard bounds** defined by deployment profile:

Definition 4.1 (Adaptive Bounds).

$$\forall t : \begin{cases} R_{min} \leq R_{max}(t) \leq R_{ceiling} \\ \tau_{min} \leq \tau_{spawn}(t) \leq \tau_{max} \\ D_{min} \leq D_{max}(t) \leq D_{ceiling} \end{cases} \tag{1}$$

4.2 Profile Definitions

Profile	R_{min}	$R_{ceiling}$	τ_{min}	τ_{max}	D_{min}	$D_{ceiling}$
SANDBOX	10	1000	0.50	0.80	5	30
PRODUCTION	10	500	0.60	0.90	5	15
CONTESTED	5	100	0.70	0.95	3	8
RESEARCH	20	800	0.55	0.85	5	20

Table 2: Adaptive bounds by deployment profile.

Profile rationale:

- **SANDBOX:** Wide bounds for experimentation; failure acceptable
- **PRODUCTION:** Moderate bounds; stability critical
- **CONTESTED:** Tight bounds; under potential attack
- **RESEARCH:** Wide spawn capacity; probe deployment enabled

4.3 Clamp Function

Listing 4: Bound enforcement

```python
def _clamp(self, value: float, min_val: float, max_val: float) -> float:
    """CRITICAL: Ensures value NEVER exceeds bounds."""
    return max(min_val, min(value, max_val))
```

5 Calibration Algorithm

5.1 Calibration Cycle

Listing 5: Main calibration cycle

```python
def calibrate_cycle(self, swarm_state, current_tick: int) -> List[Adjustment]:
    """Called every T_calibration=10,000 ticks."""

    # Step 0: Check cooldown (prevent oscillation)
    if current_tick - self.last_adjustment_tick < self.T_cooldown:
        return []  # Skip this cycle

    # Step 1: Compute metrics
    metrics = self.metrics_computer.compute_metrics(swarm_state)
    valid, reason = self.metrics_computer.validate_metrics(metrics)
    if not valid:
        log_warning(f"Invalid metrics: {reason}")
        return []

    # Step 2: Collect PROPOSED adjustments (don't apply yet)
    proposals = []
    proposals.extend(self._propose_health_adjustments(metrics, current_tick))
    proposals.extend(self._propose_resource_adjustments(metrics, current_tick))
    proposals.extend(self._propose_depth_adjustments(metrics, current_tick))

    # Step 3: Resolve conflicts (Gap 2)
    resolved = self._resolve_conflicts(proposals)

    # Step 4: Apply resolved adjustments
    for adj in resolved:
        self._apply_adjustment(adj)
```

```
        self._log_adjustment(adj)

    # Step 5: Update last adjustment tick
    if resolved:
        self.last_adjustment_tick = current_tick

    return resolved
```

5.1.1 Multi-Parameter Conflict Resolution (Gap 2)

When multiple adjustment sources propose changes to the same parameter, ASG resolves conflicts by priority:

Listing 6: Conflict resolution

```python
def _resolve_conflicts(self, proposals: List[Adjustment]) -> List[Adjustment]:
    """
    If multiple proposals adjust same parameter, choose by priority.
    Safety-critical reasons take precedence over operational ones.
    """
    # Priority hierarchy (lower = higher priority)
    PRIORITY = {
        AdjustmentReason.RESOURCE_PRESSURE: 1,      # Safety-critical
        AdjustmentReason.APPROACHING_DEPTH_LIMIT: 2,
        AdjustmentReason.HIGH_SPAWN_LOW_HEALTH: 3,  # Operational
        AdjustmentReason.RAISE_HEALTH_BAR: 4,
        AdjustmentReason.HEALTHY_LOW_SPAWN: 5,      # Optimization
        AdjustmentReason.LOWER_HEALTH_BAR: 6,
        AdjustmentReason.SHALLOW_LINEAGES: 7,
        AdjustmentReason.UNDERUTILIZED: 8,
    }

    # Group proposals by parameter
    by_parameter = {}
    for proposal in proposals:
        param = proposal.parameter
        if param not in by_parameter:
            by_parameter[param] = []
        by_parameter[param].append(proposal)

    # Resolve each parameter
    resolved = []
    for param, competing in by_parameter.items():
        if len(competing) == 1:
            resolved.append(competing[0])
        else:
            # Multiple proposals: choose highest priority
            best = min(competing, key=lambda p: PRIORITY.get(p.reason, 99))
            resolved.append(best)

            # Log conflict resolution
            log_to_dctm(
                event_type='ASG_CONFLICT_RESOLVED',
                parameter=param,
                winner=best.reason,
                losers=[p.reason for p in competing if p != best]
            )

    return resolved
```

Priority rationale:

1. **RESOURCE_PRESSURE** (highest): Vault exhaustion is safety-critical

2. **APPROACHING_DEPTH_LIMIT**: Deep chains indicate potential attack
3. **HIGH_SPAWN_LOW_HEALTH**: System stability
4. **Optimization reasons** (lowest): Can wait

5.2   Health-Correlated Adjustment

Listing 7: Health degradation response

```python
def _adjust_for_health(self, metrics: SwarmMetrics, tick: int) -> List[
    Adjustment]:
    """
    If health LOW + spawn rate HIGH -> reduce R_max, raise tau_spawn
    If health HIGH + spawn rate LOW -> increase R_max, lower tau_spawn
    """
    adjustments = []

    # Scenario 1: Health degradation with high spawn
    if metrics.avg_health < 0.65 and metrics.spawn_rate > 0.8 * self.R_max:
        # Reduce R_max by 10%
        new_R = self._clamp(self.R_max * 0.9, self.bounds['R_min'], self.bounds
            ['R_ceiling'])
        if new_R != self.R_max:
            adjustments.append(self._record_adjustment('R_max', self.R_max,
                new_R,
                AdjustmentReason.HIGH_SPAWN_LOW_HEALTH, tick))
            self.R_max = new_R

        # Raise tau_spawn by 0.05
        new_tau = self._clamp(self.tau_spawn + 0.05, self.bounds['tau_min'],
            self.bounds['tau_max'])
        if new_tau != self.tau_spawn:
            adjustments.append(self._record_adjustment('tau_spawn', self.
                tau_spawn, new_tau,
                AdjustmentReason.RAISE_HEALTH_BAR, tick))
            self.tau_spawn = new_tau

    # Scenario 2: Healthy + underutilized
    elif metrics.avg_health > 0.85 and metrics.spawn_rate < 0.3 * self.R_max:
        # Increase R_max by 5%
        new_R = self._clamp(self.R_max * 1.05, self.bounds['R_min'], self.
            bounds['R_ceiling'])
        if new_R != self.R_max:
            adjustments.append(self._record_adjustment('R_max', self.R_max,
                new_R,
                AdjustmentReason.HEALTHY_LOW_SPAWN, tick))
            self.R_max = new_R

        # Lower tau_spawn by 0.02
        new_tau = self._clamp(self.tau_spawn - 0.02, self.bounds['tau_min'],
            self.bounds['tau_max'])
        if new_tau != self.tau_spawn:
            adjustments.append(self._record_adjustment('tau_spawn', self.
                tau_spawn, new_tau,
                AdjustmentReason.LOWER_HEALTH_BAR, tick))
            self.tau_spawn = new_tau

    return adjustments
```

Why these thresholds?

- **0.65:** Below this, health is degrading (action needed)
- **0.85:** Above this, swarm is healthy (can be more aggressive)

- **0.65–0.85:** Stable range—no adjustment
- **10% reduce, 5% increase:** Conservative asymmetry

5.2.1   Velocity-Based Adjustment (Gap 3)

The base adjustment factors above assume uniform degradation rates. For more intelligent response, ASG uses **health velocity**:

Listing 8: Velocity-aware adjustment

```python
def _get_adjustment_factor(self, metrics: SwarmMetrics) -> float:
    """
    Adjust response aggressiveness based on health velocity.
    Rapid degradation -> more aggressive response.
    """
    velocity = metrics.health_trend  # Change per 10K ticks

    if velocity < -0.10:  # Rapid degradation (>10% drop)
        return 0.85  # Aggressive: -15% R_max
    elif velocity < -0.05:  # Moderate degradation
        return 0.90  # Normal: -10% R_max
    elif velocity < -0.02:  # Slow degradation
        return 0.95  # Conservative: -5% R_max
    else:  # Stable or improving
        return 1.0  # No change
```

This ensures rapid health drops get rapid fixes, while gradual degradation receives measured response.

5.3   Resource Utilization Adjustment

Listing 9: Vault pressure response

```python
def _adjust_for_resources(self, metrics: SwarmMetrics, tick: int) -> List[
    Adjustment]:
    """Adjust tau_spawn based on Vault utilization (quality over quantity)."""
    adjustments = []

    # High utilization -> be more selective
    if metrics.vault_utilization > 0.9:
        new_tau = self._clamp(self.tau_spawn + 0.05, self.bounds['tau_min'],
            self.bounds['tau_max'])
        if new_tau != self.tau_spawn:
            adjustments.append(self._record_adjustment('tau_spawn', self.
                tau_spawn, new_tau,
                AdjustmentReason.RESOURCE_PRESSURE, tick))
            self.tau_spawn = new_tau

    # Low utilization -> less selective
    elif metrics.vault_utilization < 0.5:
        new_tau = self._clamp(self.tau_spawn - 0.03, self.bounds['tau_min'],
            self.bounds['tau_max'])
        if new_tau != self.tau_spawn:
            adjustments.append(self._record_adjustment('tau_spawn', self.
                tau_spawn, new_tau,
                AdjustmentReason.UNDERUTILIZED, tick))
            self.tau_spawn = new_tau

    return adjustments
```

5.4   Depth Scaling

Listing 10: Depth limit adjustment

```python
def _adjust_for_depth(self, metrics: SwarmMetrics, tick: int) -> List[
    Adjustment]:
    """Adjust D_max based on lineage depth patterns."""
    adjustments = []

    # Approaching limit -> reduce
    if metrics.avg_depth > 0.8 * self.D_max:
        new_D = self._clamp(self.D_max - 1, self.bounds['D_min'], self.bounds['
            D_ceiling'])
        if new_D != self.D_max:
            adjustments.append(self._record_adjustment('D_max', self.D_max,
                new_D,
                AdjustmentReason.APPROACHING_DEPTH_LIMIT, tick))
            self.D_max = int(new_D)

    # Very shallow -> increase
    elif metrics.avg_depth < 0.3 * self.D_max:
        new_D = self._clamp(self.D_max + 1, self.bounds['D_min'], self.bounds['
            D_ceiling'])
        if new_D != self.D_max:
            adjustments.append(self._record_adjustment('D_max', self.D_max,
                new_D,
                AdjustmentReason.SHALLOW_LINEAGES, tick))
            self.D_max = int(new_D)

    return adjustments
```

6   Safeguards

6.1   Hysteresis (Oscillation Prevention)

Definition 6.1 (Cooldown Period). After any adjustment, ASG skips calibration for $T_{cooldown} = 30,000$ ticks.

```python
# In calibrate_cycle():
if current_tick - self.last_adjustment_tick < self.T_cooldown:
    return []  # Skip - prevent oscillation
```

Why 30,000 ticks: Shorter cooldowns (e.g., 2,000) allow oscillation when health fluctuates around thresholds. 30K ticks ensures adjustments have time to propagate and stabilize before re-evaluation.

6.2   Gardener Rollback Authority

Listing 11: Rollback mechanism

```python
def rollback_adjustment(self, adjustment_id: str, gardener_signature: bytes) ->
    bool:
    """Gardener can revert any adjustment within T_review window."""

    adjustment = self._get_adjustment_by_id(adjustment_id)
    if not adjustment:
        raise AdjustmentNotFound(adjustment_id)

    # Check rollback window
```

```
        if current_tick() > adjustment.reversible_until:
            raise RollbackWindowExpired(
                f"Adjustment {adjustment_id} expired. Window: {self.T_review} ticks
                    ")

        # Verify Gardener signature (Commandment 4)
        if not self._verify_gardener_signature(gardener_signature):
            raise UnauthorizedRollback("Invalid Gardener signature")

        # Revert parameter
        setattr(self, adjustment.parameter, adjustment.old_value)
        adjustment.reverted = True
        adjustment.reverted_at = current_tick()

        # Log rollback to d-CTM
        log_to_dctm(
            event_type='ASG_ROLLBACK',
            adjustment_id=adjustment_id,
            gardener=gardener_signature,
            reason='GARDENER_OVERRIDE'
        )

        return True
```

6.3   Emergency Lockdown

Emergency Lockdown

During active attack or system crisis, Gardener can freeze ASG at most conservative values:

Listing 12: Emergency lockdown

```
def emergency_lockdown(self, gardener_signature: bytes) -> None:
    """Instant freeze to most conservative values."""

    if not self._verify_gardener_signature(gardener_signature):
        raise UnauthorizedLockdown("Invalid Gardener signature")

    # Set to most conservative bounds
    self.R_max = self.bounds['R_min']
    self.tau_spawn = self.bounds['tau_max']
    self.D_max = self.bounds['D_min']

    # Freeze ASG
    self.locked = True

    log_to_dctm(
        event_type='ASG_LOCKDOWN',
        reason='GARDENER_EMERGENCY',
        final_params=self.get_current_parameters()
    )

def unlock(self, gardener_signature: bytes) -> None:
    """Resume normal ASG operation."""
    if not self._verify_gardener_signature(gardener_signature):
        raise UnauthorizedUnlock("Invalid Gardener signature")
    self.locked = False
    log_to_dctm(event_type='ASG_UNLOCK')
```

6.4   Fork Handling

When Constitutional Fork occurs (Appendix L):

Listing 13: Fork handling

```python
def on_fork(self, fork_event: ForkEvent) -> None:
    """Both branches inherit ASG state; evolve independently post-fork."""

    # Clone current state for forked branch
    forked_asg = AdaptiveSpawnGovernor(self.profile)
    forked_asg.R_max = self.R_max
    forked_asg.tau_spawn = self.tau_spawn
    forked_asg.D_max = self.D_max
    forked_asg.adjustment_history = self.adjustment_history.copy()

    # Log fork
    log_to_dctm(
        event_type='ASG_FORK',
        fork_id=fork_event.id,
        state_snapshot=self.get_current_parameters()
    )

    return forked_asg
```

6.5   Predictive Integration (Discovery Stack)

Listing 14: M-Stack prediction integration

```python
def check_predictions(self, discovery_stack) -> None:
    """If M-Stack predicts health degradation, pre-emptively tighten."""

    prediction = discovery_stack.predict_health_trend(horizon=1000)

    if prediction.severity == 'HIGH' and prediction.direction == 'DECLINING':
        # Pre-emptive tightening
        self.tau_spawn = self._clamp(self.tau_spawn + 0.03,
            self.bounds['tau_min'], self.bounds['tau_max'])

        log_to_dctm(
            event_type='ASG_PREEMPTIVE',
            reason='M_STACK_PREDICTION',
            prediction=prediction
        )
```

6.6   Adversarial Robustness (Gap 9)

> **Caution**
>
> For CONTESTED deployments, ASG must detect and resist manipulation attempts.

Attack Vectors:

1. **Parameter Poisoning:** Adversary spawns low-health capsules to drag down avg_health, causing ASG to raise τ_{spawn} until healthy capsules can't spawn
2. **Oscillation Amplification:** Adversary alternates behavior to cause parameter thrashing
3. **Coordinated Behavior:** Multiple capsules perform identical unusual actions

Listing 15: Adversarial manipulation detection

```python
class ManipulationDetector:
    """Detect potential adversarial manipulation of ASG"""

    def detect_manipulation(self, metrics, swarm_state, history) -> Detection:
        manipulations = []

        # Check 1: Parameter Poisoning
        recent = swarm_state.get_capsules_created(window=5000)
        unhealthy_ratio = sum(1 for c in recent if c.health < 0.6) / len(recent
            )
        if unhealthy_ratio > 0.50:  # Half of recent capsules unhealthy
            manipulations.append({
                'type': 'PARAMETER_POISONING',
                'confidence': min(1.0, unhealthy_ratio / 0.5),
                'evidence': f'{unhealthy_ratio*100:.0f}% recent capsules
                    unhealthy'
            })

        # Check 2: Oscillation Amplification
        spawn_history = swarm_state.get_spawn_rate_history(window=50000, bins
            =5)
        variance = statistics.variance(spawn_history)
        if variance > self.oscillation_threshold:
            manipulations.append({
                'type': 'OSCILLATION_AMPLIFICATION',
                'confidence': min(1.0, variance / self.oscillation_threshold),
                'evidence': f'Spawn variance={variance:.2f}'
            })

        # Check 3: Coordinated Behavior
        action_counts = swarm_state.get_action_distribution()
        max_ratio = max(action_counts.values()) / sum(action_counts.values())
        if max_ratio > 0.30:  # >30% doing same thing
            manipulations.append({
                'type': 'COORDINATED_BEHAVIOR',
                'confidence': max_ratio,
                'evidence': f'{max_ratio*100:.0f}% identical actions'
            })

        # Classify threat level
        if any(m['confidence'] > 0.8 for m in manipulations):
            threat_level = 'RED'
        elif manipulations:
            threat_level = 'YELLOW'
        else:
            threat_level = 'GREEN'

        return Detection(
            threat_level=threat_level,
            manipulations=manipulations,
            timestamp=current_tick()
        )

class AdaptiveSpawnGovernor:
    def calibrate_cycle(self, swarm_state, current_tick):
        # FIRST: Check for adversarial manipulation
        detection = self.detector.detect_manipulation(
            self.last_metrics, swarm_state, self.adjustment_history)

        if detection.threat_level == 'RED':
            # FREEZE: Do not adjust, notify Gardener
            notify_gardener(URGENT, detection)
```

```
        log_to_dctm('ASG_THREAT_DETECTED', detection)
        return []  # No adjustments

    if detection.threat_level == 'YELLOW':
        # SLOW: Reduce adjustment aggressiveness
        self.adjustment_factor = 0.5  # Half-speed adjustments
        log_to_dctm('ASG_THREAT_YELLOW', detection)
    else:
        self.adjustment_factor = 1.0  # Normal

    # Proceed with normal calibration...
```

Response to threat levels:

- **GREEN:** Normal calibration
- **YELLOW:** Adjustments at 50% rate, Gardener notified
- **RED:** ASG frozen, Gardener notified urgently

6.7   Partial System Failure Handling (Gap 6)

ASG must adapt behavior during partial infrastructure failures (not just total failures):

Listing 16: Partial failure detection and response

```
class PartialFailureDetector:
    """Detect and respond to partial infrastructure failures."""

    def assess_infrastructure(self, swarm_state) -> InfrastructureStatus:
        """Check health of supporting systems."""

        # Arbiter availability
        total_arbiters = swarm_state.get_arbiter_count()
        healthy_arbiters = swarm_state.get_healthy_arbiters()
        arbiter_ratio = healthy_arbiters / total_arbiters

        # d-CTM responsiveness
        dctm_latency = swarm_state.measure_dctm_latency()
        dctm_healthy = dctm_latency < 100  # ms threshold

        # Vault availability
        vault_responsive = swarm_state.vault.is_responsive()

        return InfrastructureStatus(
            arbiter_availability=arbiter_ratio,
            dctm_healthy=dctm_healthy,
            vault_responsive=vault_responsive
        )

    def get_asg_mode(self, status: InfrastructureStatus) -> str:
        """Determine ASG operating mode based on infrastructure health."""

        # Critical: Any core system down
        if not status.vault_responsive:
            return 'EMERGENCY_HALT'  # Cannot operate without Vault

        if not status.dctm_healthy:
            return 'AUDIT_DEGRADED'  # Can operate but logging impaired

        # Arbiter availability thresholds
        if status.arbiter_availability < 0.20:
            return 'EMERGENCY_HALT'  # Too few Arbiters
        elif status.arbiter_availability < 0.50:
            return 'DEGRADED_CONSERVATIVE'  # Partial failure
```

```
        elif status.arbiter_availability < 0.80:
            return 'DEGRADED_NORMAL'  # Minor degradation
        else:
            return 'NORMAL'  # Full operation

class AdaptiveSpawnGovernor:
    def calibrate_cycle(self, swarm_state, current_tick):
        # Check infrastructure health FIRST
        infra_status = self.failure_detector.assess_infrastructure(swarm_state)
        asg_mode = self.failure_detector.get_asg_mode(infra_status)

        if asg_mode == 'EMERGENCY_HALT':
            self._emergency_conservative()
            notify_gardener(CRITICAL, infra_status)
            return []

        if asg_mode == 'DEGRADED_CONSERVATIVE':
            # Partial failure: Be very conservative
            self.adjustment_factor = 0.25  # Quarter-speed
            self.tau_spawn = self._clamp(
                self.tau_spawn + 0.05,  # Raise bar
                self.bounds['tau_min'], self.bounds['tau_max']
            )
            notify_gardener(WARNING, infra_status)

        elif asg_mode == 'DEGRADED_NORMAL':
            # Minor degradation: Slightly conservative
            self.adjustment_factor = 0.75

        elif asg_mode == 'AUDIT_DEGRADED':
            # d-CTM slow: Continue but log warning
            log_warning("d-CTM latency elevated - audit trail may be delayed")
            self.adjustment_factor = 1.0

        else:  # NORMAL
            self.adjustment_factor = 1.0

        # Proceed with calibration...
```

Partial failure thresholds:

Arbiter %	Mode	ASG Behavior
< 20%	EMERGENCY_HALT	Freeze at conservative values
20 − 50%	DEGRADED_CONSERVATIVE	25% adjustment rate, raise τ
50 − 80%	DEGRADED_NORMAL	75% adjustment rate
> 80%	NORMAL	Full operation

7 Spawn Gate Integration

7.1 Updated Spawn Gate

Listing 17: Adaptive spawn gate

```
class AdaptiveSpawnGate:
    """Spawn gate using adaptive parameters from ASG."""

    def __init__(self, asg: AdaptiveSpawnGovernor):
        self.asg = asg

    def should_spawn(self, parent, child_spec, tick: int) -> Tuple[bool, str]:
```

```
    """Evaluate S1-S6 with adaptive parameters."""

    # Get current adaptive parameters
    params = self.asg.get_current_parameters()
    tau_spawn = params['tau_spawn']
    D_max = params['D_max']
    R_max = params['R_max']

    # S1: Task justification
    if not self._task_justified(parent, child_spec):
        return (False, "S1: No task justification")

    # S2: Resources
    if parent.vault.available < child_spec.cost:
        return (False, f"S2: Insufficient resources")

    # S3: Health (ADAPTIVE)
    if parent.health < tau_spawn:
        return (False, f"S3: Health {parent.health:.2f} < {tau_spawn:.2f} (
            adaptive)")

    # S4: Depth (ADAPTIVE)
    if parent.depth >= D_max:
        return (False, f"S4: Depth {parent.depth} >= {D_max} (adaptive)")

    # S5: Rate (ADAPTIVE)
    if not self._rate_compliant(parent, R_max):
        return (False, f"S5: Rate limit (R_max={R_max})")

    # S6: Integrity
    if parent.has_anomaly_flag():
        return (False, "S6: ANOMALY flag")

    return (True, "All conditions satisfied")
```

8 Testing

8.1 Test Rationale: τ_{spawn} Parameter

The τ_{spawn} (spawn threshold) parameter is the most critical tunable in ASG because it directly gates capsule creation. The following rationale explains why specific tests exist:

Table 3: τ_{spawn} test rationale.

Test	τ Behavior	Why This Test Exists
Health degradation (T1)	$\tau \uparrow 0.05$ when LOW	Prevents spawn cascade during stress. Without this, unhealthy swarms spawn more unhealthy capsules, creating feedback death spiral.
Bounded adaptation (T2)	$\tau \in [\tau_{min}, \tau_{max}]$	Prevents adversarial manipulation. A compromised ASG cannot lower τ to 0 (spawn freely) or raise to 1 (spawn lockout).
Resource pressure (T6)	$\tau \uparrow$ when vault full	Vault resource exhaustion is an early warning. Raising τ preemptively prevents OOM-style cascade failures.
Healthy + low spawn (T5)	$\tau \downarrow 0.02$ slowly	Recovery must be asymmetric: fast tightening (crisis response) but slow relaxation (prevent oscillation). The $0.05/0.02$ ratio is empirically derived from swarm simulations.

Why 0.05 increase vs. 0.02 decrease? This asymmetry implements *defensive adaptation*: the system tightens quickly during stress (respond to threats) but relaxes slowly during recovery (avoid premature relaxation). This pattern is derived from control theory's "slow integral, fast derivative" principle for stable feedback systems.

8.2 Test Suite Summary

#	Test	Validates
1	Health degradation response	LOW health + HIGH spawn → reduce R_{max}, raise τ
2	Bounded adaptation	100 cycles of stress → bounds respected
3	Gardener rollback	Rollback works within T_{review}
4	Profile bounds	Different profiles have different bounds
5	Recovery	Misconfigured system recovers in < 50K ticks
6	Resource pressure	HIGH vault util → raise τ
7	Depth scaling	Approaching D_{max} → reduce D_{max}
8	Multi-parameter	Coordinated adjustment of R, τ, D
9	Rollback expiration	Cannot rollback after T_{review}
10	d-CTM logging	All adjustments logged with diagnostics
11	Performance	Overhead < 1% per calibration
12	Integration	Full spawn gate with ASG
	Gap 2: Conflict Resolution Tests	
13	Conflict detection	Detect when 2+ proposals target same param
14	Priority ordering	RESOURCE_PRESSURE wins over HEALTHY_LOW_SPAWN
15	Conflict logging	Conflicts logged to d-CTM with winner/losers
	Gap 6: Partial Failure Tests	
16	Arbiter 50% failure	ASG enters DEGRADED_CONSERVATIVE mode
17	Arbiter 30% failure	ASG enters EMERGENCY_HALT mode
18	d-CTM degradation	ASG continues with warning logged
	Gap 9: Adversarial Robustness Tests	
19	Parameter poisoning detection	Detect 50%+ unhealthy recent capsules
20	Oscillation amplification	Detect high spawn variance pattern
21	Coordinated behavior	Detect 30%+ identical actions
22	RED threat freeze	ASG stops adjusting on RED threat
23	YELLOW threat slowdown	ASG adjusts at 50% rate on YELLOW

Table 4: ASG test suite (v1.3) — 23 tests covering all gaps.

8.3 Acceptance Criteria

- All 23 tests pass (12 core + 3 conflict + 3 partial failure + 5 adversarial)
- Performance overhead < 1% per calibration cycle
- System recovers from misconfiguration in < 50,000 ticks
- Gardener rollback verified working
- Conflict resolution priority ordering verified
- Partial failure mode transitions verified

- Adversarial manipulation detection verified for CONTESTED profile
- Full d-CTM audit trail with rich diagnostics

9 Formal Properties

Theorem 9.1 (Bounded Adaptation). *For all ticks t, ASG parameters remain within profile bounds:*

$$\forall t : R_{min} \leq R_{max}(t) \leq R_{ceiling} \land \tau_{min} \leq \tau_{spawn}(t) \leq \tau_{max} \tag{2}$$

Proof. All parameter updates pass through `_clamp()` which enforces bounds. By induction: initial values within bounds (from profile defaults), and `_clamp` preserves bounds. □

Theorem 9.2 (Recovery Bound). *A misconfigured system recovers within $5 \cdot T_{calibration}$ ticks.*

Proof. Each calibration cycle adjusts parameters by at most 10%. After 5 cycles (50,000 ticks), parameters converge to stable range where health exceeds 0.65 and spawn rate is appropriate. □

Changelog

v1.3 (December 2025) — *Complete Gap Coverage*

- **Gap 2 (Added):** Multi-parameter conflict resolution (§5.1.1)
- **Gap 6 (Added):** Partial failure handling (§6.7)
- Test suite expanded to 23 tests (was 17)
- All 9 plex team gaps now addressed

v1.1 (December 2025)

- **Gap 1 (Fixed):** Increased $T_{cooldown}$ from 2K to 30K ticks
- **Gap 3 (Added):** Velocity-based adjustment factors (§5.2.1)
- **Gap 8 (Added):** Rich observability/diagnostics (§3.4.1)
- **Gap 9 (Added):** Adversarial robustness detection (§6.6)
- Enhanced manipulation detection for CONTESTED profile

v1.0 (December 2025)

- Initial specification
- Metrics computation (§3)
- Profile bounds (§4)
- Calibration algorithm (§5)
- Safeguards: hysteresis, rollback, lockdown, fork handling (§6)
- Spawn gate integration (§7)
- Test suite (§8)