IMMUNIS

Product Requirements Document & Build Guide

Full-Spectrum System Security for the NeuroGraph Ecosystem

Version 1.0 — Unified Architecture February 2026

CONFIDENTIAL DRAFT

Table of Contents

1. Executive Summary

Immunis is the full-spectrum system security module for the E-T Systems / NeuroGraph ecosystem. It covers everything TrollGuard does not: viruses, malware, keyloggers, rootkits, network attacks, privilege escalation, supply chain threats, file system anomalies, process behavioral anomalies, and zero-day attacks.

Immunis does not use signature-based or heuristic-based security as its primary defense. It learns what healthy system behavior *feels like* through the NG-Lite substrate. Novelty detection catches zero-days by recognizing behavioral shape deviations even when the specific attack has never been seen before. A mature Immunis with a rich substrate has exponentially greater odds of catching zero-days compared to static or heuristic approaches, because it has learned thousands of causal chains — attack *sequences*, not just attack *signatures*.

Known signatures serve as a fast prefilter (the same role Cisco/YARA plays for TrollGuard Layer 1), but the real power is substrate-based behavioral classification and, at Tier 3 with full NeuroGraph, causal chain recognition that lets Immunis intervene in attack sequences before they complete.

1.1 Core Innovation: Substrate-Learned Behavioral Security

Traditional security systems match patterns. Immunis learns *shapes*. The NG-Lite substrate's Hebbian learning means that every system event — process spawn, file modification, network connection, resource spike — gets embedded and fed to the substrate. Over time, the substrate develops a rich topology of what "healthy" looks like and what "under attack" looks like. When a novel threat arrives, the substrate recognizes that the *behavioral shape* is anomalous even if the specific payload, binary, or exploit has never existed before.

At Tier 3 (full NeuroGraph), this extends to temporal causality via hyperedges and STDP (spike-timing dependent plasticity). A suspicious network connection at T1, followed by a file system modification at T2, followed by a new process spawning at T3 — that's a causal chain. Full NeuroGraph encodes that chain as a hyperedge connecting all three events with their temporal relationships preserved. The next time the system sees that network connection pattern, the substrate has already learned the *sequence* — it recognizes the opening move and intervenes early, before the file modification and process spawn occur.

1.2 Relationship to Existing Modules

Immunis is one of three security/health modules in the E-T Systems ecosystem:

Module	Domain	Biological Analog
TrollGuard	Text entering AI system. Prompt injection, jailbreaks, PII.	Skin — perimeter defense
Immunis	Entire host and everything running on it (non-text). Viruses, malware, rootkits, network attacks, supply chain, zero-days, process anomalies, file system threats.	Immune system — T-cells, white blood cells
The Healing Collective	Repair, optimization, antibody creation, antibody distribution.	B-cells creating antibodies + tissue repair

A fourth module, **Cricket** (forthcoming), serves as the conscience/stability guardian — preventing catastrophic actions, deliberate or accidental. Cricket enforces behavioral boundaries ("don't delete system32"). Cricket is NOT part of Immunis and has a separate domain.

A fifth peer module, **Bunyan** (forthcoming), provides NG-Lite-based smart logging with causal story format. Bunyan presents system events as narrative chains rather than isolated log entries, showing what caused what and when. Bunyan's substrate contributions are high-value input for Immunis — correlated event narratives are exactly the signal Immunis needs for causal chain learning.

1.3 How This Document Is Structured

This document serves as both PRD and build guide. Every section contains enough specification detail that a coding agent can implement without interpretation. Where a design decision is specified, the rationale is given. Where a value is specified, the units and constraints are given. Where a pattern is specified, the canonical source file is referenced.

The coding agent implementing this PRD MUST NOT:

- Invent new vendored files beyond the four specified in §16
- Create APIs, HTTP endpoints, or direct function calls between modules

- Add dependencies not listed in §13.2
- Modify the OpenClawAdapter base class interface
- Create new communication patterns between modules

The coding agent implementing this PRD MUST:

- Follow the vendored file pattern exactly as specified in §16
- Use the OpenClawAdapter subclass pattern from `trollguard_hook.py` and `healing_collective_hook.py`
- Use the `et_module.json` v2 schema from The Healing Collective
- Reference §-numbers in all changelog entries (e.g., "PRD §4.3 specifies…")

---

2. System Architecture Overview

Immunis operates as a single Python module that integrates with the E-T Systems ecosystem via the standard vendored file pattern. It does NOT function as a service, daemon, or server. It is an OpenClaw skill that participates in the NG-Lite substrate alongside all other modules.

2.1 What Immunis Is

Immunis is a threat detection and response engine that monitors system-level signals:

- File system events (creation, modification, deletion, permission changes)
- Process activity (new processes, unexpected parent processes, resource consumption anomalies)
- Network connections (unexpected outbound connections, unusual ports, traffic volume spikes)
- Package/dependency integrity (new packages, changed checksums, supply chain anomalies)
- System logs (auth failures, brute force patterns, privilege escalation attempts)
- Memory patterns (unusual allocation, potential buffer overflow signatures)
- Signals from peer modules via the NG-Lite substrate (TrollGuard scan outcomes, THC failure patterns, Bunyan causal narratives)

2.2 What Immunis Is Not

Immunis does NOT:

- Scan text for prompt injection, jailbreaks, or PII (that is TrollGuard's domain)
- Repair broken systems or optimize performance (that is THC's domain)
- Create antibodies or distribute them (that is THC's domain — see §3.2)
- Enforce behavioral constraints on what the ecosystem is allowed to do (that is Cricket's domain)
- Provide a web interface, REST API, or any other network-accessible service
- Send data directly to any other module (see §2.3)

2.3 The Substrate Model — No Module Sends Data to Any Module

This is the most critical architectural principle in the entire document. Every implementation decision flows from it.

No module sends anything to any module. There is no sender and receiver. There is the **substrate**, and every module is a participant in it.

When Immunis records a threat outcome via `record_outcome()` , it is not dispatching a message. It is strengthening a pathway on the shared substrate. THC does not "receive" that information. THC's next query through the substrate returns different results because the topology has changed. The learning propagates through the peer bridge the same way a signal propagates through neural tissue — not as a routed packet, but as an activation pattern that reshapes the landscape for everyone connected.

At Tier 1, learning is local. A module learns in isolation. At Tier 2, the peer bridge means co-located modules share a substrate — Immunis detecting a malware signature changes the activation landscape that THC queries when it encounters the resulting damage. They do not coordinate. They share a nervous system.

At Tier 3 — full NeuroGraph — hyperedges encode temporal causality and STDP encodes the order of events. This is where causal chain learning becomes possible.

Implementation constraint: The coding agent MUST NOT create any mechanism by which one module directly calls a function, sends a message, writes to a queue, or otherwise communicates with another module outside of the NG-Lite substrate and peer bridge. The vendored files (`ng_lite.py` , `ng_peer_bridge.py` , `ng_ecosystem.py` , `openclaw_adapter.py`) are the ONLY inter-module communication mechanism. The one exception is the Autonomic Nervous System vendored state file (§8), which is a fast-read coordination mechanism for ecosystem-wide threat level.

---

3. Module Triad Boundaries

3.1 TrollGuard's Domain

TrollGuard's domain is **text entering the AI system**. It scans strings using a 4-layer pipeline (Cisco static analysis → ML classifier → Swarm Audit with Semantic Air Gap → Runtime Vector Sentry). If the input is text that an LLM will process, TrollGuard handles it.

TrollGuard does NOT monitor file systems, processes, network connections, or any other system-level signal. TrollGuard does NOT create antibodies — it applies rules that were strengthened by the substrate.

3.2 The Healing Collective's Domain

THC's domain is **repair, optimization, and antibody creation/distribution**. When something breaks — a process crashes, a config is corrupted, a service degrades — THC diagnoses it via the 7-step pipeline (Observe → Recognize → Recall → Propose → Validate → Execute → Learn), applies repair primitives, and records outcomes.

THC also creates antibodies and distributes them via the peer bridge. Biological mapping: Immunis = T-cells and white blood cells (fighting the infection). THC = B-cells (creating antibodies for immune memory) + tissue repair.

The antibody handoff is NOT a handoff. Immunis records threat outcomes on the substrate. THC's NG-Lite substrate naturally reflects those topology changes. When THC queries its substrate for patterns relevant to a failure it's repairing, the Immunis-recorded threat data is part of the landscape THC sees. THC creates antibodies based on its own substrate's learned patterns, which now include Immunis's contributions. Nobody sends anything.

3.3 Immunis's Domain

Immunis's domain is **everything TrollGuard doesn't cover**: the entire host system and everything running on it, excluding text-level AI security. Specifically:

- Viruses, malware, ransomware, keyloggers, rootkits
- Network attacks (unauthorized outbound connections, port scans, data exfiltration)
- Privilege escalation attempts
- Supply chain threats (malicious packages, dependency tampering)

- File system anomalies (unauthorized modifications, suspicious new files, permission changes)
- Process behavioral anomalies (unexpected process trees, resource abuse, unauthorized spawning)
- Zero-day attacks (via substrate-learned behavioral shape recognition)
- Substrate drift across all modules (are TrollGuard's weights being poisoned? Is THC's DVS accumulating adversarial entries?)

3.4 Cricket's Domain (Forthcoming — Referenced for Boundary Clarity)

Cricket's domain is **behavioral constraint enforcement**. Cricket prevents catastrophic actions, whether deliberate or accidental. If a repair primitive would delete a critical system file, Cricket prevents it. If a module's learned behavior drifts toward dangerous territory, Cricket intervenes.

Boundary scenario — malicious post-install script:

1. An OpenClaw skill downloads a Python package.
2. That package contains a malicious post-install script creating a reverse shell.
3. **TrollGuard** scans the package's text content for injection patterns (its domain: text).
4. **Immunis** detects the post-install script's system-level behavior — new process spawn, outbound network connection to an unexpected IP (its domain: system events).
5. **Cricket** (when implemented) would have prevented the post-install script from executing certain actions if they violated behavioral constraints (its domain: what the ecosystem is allowed to do).
6. **THC** repairs any damage and creates antibodies from the incident (its domain: repair + antibody creation).

All four modules contribute, but through the substrate — not through direct coordination.

---

4. The Quartermaster Pipeline

The Quartermaster is Immunis's threat detection and response pipeline. It is analogous to THC's Diagnosis Engine (7-step pipeline) and TrollGuard's 4-layer defense pipeline.

File: `core/quartermaster.py`

4.1 Pipeline Overview

The Quartermaster processes system-level signals through six stages:

Stage	Name	Role	Analogy
1	DETECT	Receive system-level signals from all sensor types	THC's Observe step
2	CLASSIFY	Known signature match (fast prefilter) vs novel (substrate-based). Categorize: malware, exploit, exfiltration, persistence, lateral movement	TrollGuard's Layer 1 → Layer 2 split
3	ASSESS	Severity scoring via substrate confidence. High-novelty (never seen) vs low-novelty (similar to known)	THC's confidence thresholds
4	RESPOND	Containment proportional to confidence: kill process, quarantine file, block connection, isolate module	THC's repair primitive execution
5	REPORT	Record outcome on substrate. Topology changes for all peer modules.	THC's Learn step
6	LEARN	Was response effective? False positive? Strengthen/weaken substrate pathways.	THC's outcome learning

4.2 Stage 1: DETECT

The DETECT stage receives signals from all sensor types (§5). Each signal arrives as a `ThreatSignal` dataclass:

```
@dataclass
class ThreatSignal:
    """A single system-level signal for the Quartermaster to process.

    Created by sensor modules (§5).  Consumed by the Quartermaster pipeline.
    """
    signal_id: str                  # UUID
    timestamp: float                # time.time()
    sensor_type: str                # "filesystem" | "process" | "network" |
                                    # "dependency" | "log" | "memory" | "substr
    event_type: str                 # Sensor-specific event classification
    raw_data: Dict[str, Any]        # Sensor-specific payload
    embedding: Optional[np.ndarray]   # Pre-computed embedding (sensors embed the
    source_module: Optional[str]      # If signal originated from substrate peer
```

The DETECT stage does NOT filter or prioritize. Every signal enters the pipeline. Filtering is CLASSIFY's job.

Implementation: The Quartermaster maintains a `collections.deque` with a configurable `maxlen` (default: 10000, §11 `quartermaster.signal_buffer_size`). Signals are appended to the deque. A processing loop pops signals from the deque and feeds them through stages 2-6. If the deque is full, the oldest unprocessed signal is dropped and a warning is logged. This is a backpressure mechanism, not a design flaw — if Immunis cannot keep up with signal volume, that itself is a signal (potentially an attack flooding the sensors).

4.3 Stage 2: CLASSIFY

CLASSIFY performs two operations in sequence:

Step 1: Known Signature Prefilter (Fast Path)

Check the signal against The Armory's (§6) known signatures. This is a cosine similarity search against stored threat embeddings, identical in mechanism to THC's DVS search but using threat signatures instead of failure signatures.

If a match is found above the `armory_match_threshold` (default: 0.90, §11), the signal is immediately classified with the matched signature's category, severity, and recommended response. This skips substrate-based classification entirely — the same "fail fast" principle as TrollGuard's Layer 1 catching known threats before expensive ML inference.

Step 2: Substrate-Based Classification (Novel Path)

If no known signature matches, the signal's embedding is fed to the NG-Lite substrate via `get_context()`. The substrate returns:

- `novelty`: How unlike anything the substrate has seen before (0.0 = completely familiar, 1.0 = completely novel)
- `recommendations`: Previously recorded outcomes for similar patterns
- Tier 3 only: `ng_context` with causal chain associations

Classification categories:

Category	Description

`malware`	Malicious software signatures, suspicious binaries, known malware behavior patterns
`exploit`	Privilege escalation, buffer overflow, code injection at system level
`exfiltration`	Unauthorized data leaving the system, suspicious outbound connections
`persistence`	Attempts to maintain unauthorized access (cron jobs, startup scripts, hidden files)
`lateral_movement`	Attempts to spread to other modules or systems
`supply_chain`	Package tampering, dependency substitution, checksum mismatches
`resource_abuse`	Cryptomining, excessive resource consumption, denial-of-service patterns
`substrate_drift`	NG-Lite weight poisoning, adversarial learning pattern injection
`unknown`	Anomalous but uncategorizable. High novelty, no substrate match.

Implementation: CLASSIFY produces a `ThreatClassification` dataclass:

```
@dataclass
class ThreatClassification:
    """Result of the CLASSIFY stage."""
    signal: ThreatSignal
    category: str                     # One of the categories above
    known_signature_match: bool       # True if fast-path matched
    matched_signature_id: Optional[str]
    substrate_novelty: float          # 0.0 to 1.0
    substrate_confidence: float       # How confident the substrate is in this cl
    recommended_response: Optional[str]  # From substrate recommendations if avai
```

4.4 Stage 3: ASSESS

ASSESS determines severity and whether the signal warrants a response, a recommendation to the user, or observation only.

Severity model:

The severity model uses substrate confidence as the primary input, modulated by novelty. This is NOT a static threshold system. These are initial starting values. From the moment Immunis gets its first input, the substrate begins learning and adjusting what these values mean in the context of the specific system it is operating on.

Substrate Confidence	Novelty	Severity	Action
≥ 0.70	Any	CRITICAL	Auto-execute response primitive (§7)
0.40 – 0.70	< 0.50	HIGH	Recommend response to user via feedback popup (§9)
0.40 – 0.70	≥ 0.50	HIGH-NOVEL	Recommend response + flag for causal chain analysis
0.15 – 0.40	Any	MEDIUM	Log + observe. No autonomous action.
< 0.15	< 0.50	LOW	Log only. Substrate records the event for future learning.
< 0.15	≥ 0.50	LOW-NOVEL	Log + observe with elevated attention. May be the first link in an unknown chain.

Important: The 0.70 auto-execute threshold matches THC's `confidence_auto_execute` (§11). The 0.40 recommendation threshold matches THC's `confidence_recommend` . The 0.15 host premium threshold matches THC's `confidence_host_premium` . This is not coincidence — it is deliberate consistency across the module ecosystem. All three values are configurable and all three are adaptive.

4.5 Stage 4: RESPOND

RESPOND selects and executes a response primitive (§7) proportional to the assessed severity.

Response selection logic:

1. If `known_signature_match` is True AND the matched signature has a recorded effective response: use that response.
2. If substrate recommendations include a response with confidence ≥ assessment threshold: use that response.
3. Otherwise, select the least-invasive response appropriate to the severity level:
    - CRITICAL: Kill process OR quarantine file OR block connection (whichever matches the signal type)
    - HIGH: Alert + recommend specific action to user

    - MEDIUM: Log + snapshot for forensics
    - LOW: Log only

Validate-before-execute contract: Every response primitive has a `validate()` method that MUST be called before `execute()`. This is the same contract as THC's repair primitives. `validate()` MUST NOT have side effects. The Quartermaster MUST NOT call `execute()` without a preceding `validate()` that returned `passed=True`. This is Immunis's equivalent of THC's Hippocratic oath.

4.6 Stage 5: REPORT

REPORT records the outcome on the NG-Lite substrate.

```
self._eco.record_outcome(
    embedding=classification.signal.embedding,
    target_id=f"threat:{classification.signal.signal_id}",
    success=response_result.status == "success",
    metadata={
        "source": "immunis",
        "category": classification.category,
        "severity": assessment.severity,
        "response_primitive": response_result.primitive_name,
        "response_status": response_result.status,
        "known_signature": classification.known_signature_match,
        "novelty": classification.substrate_novelty,
    },
)
```

This call changes the substrate topology. The next time ANY peer module (THC, TrollGuard, Bunyan) queries the substrate for patterns similar to this threat's embedding, the results will reflect this outcome. No module "receives" this data. The river flows.

4.7 Stage 6: LEARN

LEARN evaluates whether the response was effective and adjusts the substrate accordingly.

Effectiveness evaluation:

- If the response primitive returned `status="success"` AND no follow-up signals of the same category arrive within `learn_observation_window` seconds (default: 300, §11): the response is marked effective. Substrate pathways for this signal→response pairing are strengthened.
- If the response primitive returned `status="success"` BUT follow-up signals of the same

category arrive within the observation window: the response is marked partially effective. Pathways are weakly strengthened.

- If the response primitive returned `status="failed"` : pathways are weakened. The signal is flagged for user feedback (§9).
- If the signal was a false positive (user feedback via §9 marks it as not-a-threat): pathways are strongly weakened. The signal's embedding is recorded in The Armory as a known false-positive pattern.

Tier 3 causal chain learning:

When full NeuroGraph is available, LEARN also records temporal relationships between signals. If signals A, B, and C arrived within a configurable temporal window and were classified as related (same category, overlapping embeddings), a hyperedge is created connecting them with their temporal order preserved via STDP weights. This is how Immunis learns attack *sequences*, not just individual attack events.

---

5. Sensor Architecture

Immunis monitors all system-level signals. Each signal source has a dedicated sensor module that:

1. Reads raw system data
2. Embeds the signal as a normalized `np.ndarray` vector
3. Wraps it in a `ThreatSignal` dataclass (§4.2)
4. Feeds it to the Quartermaster's signal buffer

All sensors live under `core/sensors/` . Each sensor is a Python class implementing the `Sensor` abstract base class.

File: `core/sensors/base.py`

```
class Sensor(ABC):
    """Abstract base class for all Immunis sensors.

    Each sensor reads a specific system signal source, embeds events,
    and produces ThreatSignal instances for the Quartermaster.

    Subclass this.  Override:
        SENSOR_TYPE  — string identifier (required)
        _poll()      — read raw events from the system (required)
        _embed()     — embed a raw event into a vector (required)
        _is_relevant() — filter irrelevant events (optional, default: True)
    """

    SENSOR_TYPE: str = ""
    POLL_INTERVAL_SECONDS: float = 5.0  # How often to poll (configurable per sen

    @abstractmethod
    def _poll(self) -> List[Dict[str, Any]]:
        """Read raw events from the system.  Return a list of event dicts."""
        ...

    @abstractmethod
    def _embed(self, event: Dict[str, Any]) -> np.ndarray:
        """Embed a raw event into a normalized np.ndarray vector."""
        ...

    def _is_relevant(self, event: Dict[str, Any]) -> bool:
        """Filter irrelevant events.  Override to customize.  Default: all releva
        return True
```

Implementation constraint: Sensors MUST NOT import or call functions from any other module (TrollGuard, THC, etc.). Sensors read system state directly. Any cross-module information comes through the substrate, not through sensor code.

5.1 File System Sensor

File: `core/sensors/filesystem_sensor.py`

Monitors file system events using the `watchdog` library (cross-platform file system event monitoring). Falls back to periodic polling if `watchdog` is unavailable.

Events monitored:

- File created
- File modified
- File deleted
- File moved/renamed
- Permission changed
- New executable file detected
- File in sensitive directory modified (config files, cron directories, startup scripts)

Embedding strategy: File system events are encoded as feature vectors combining:

- File path hash (deterministic, position-independent)
- File extension one-hot encoding
- Directory depth
- File size delta (for modifications)
- Permission bitmask delta (for permission changes)
- Timestamp features (time of day, day of week — attacks often happen at unusual hours)
- Entropy of file name (randomized names are suspicious)

Configuration (§11):

```
sensors:
  filesystem:
    enabled: true
    poll_interval_seconds: 5.0
    watched_paths:
      - "/"                          # Monitor entire filesystem
    excluded_paths:
      - "/proc"
      - "/sys"
      - "/dev"
      - "/tmp/.immunis_*"            # Own temp files
    sensitive_paths:
      - "/etc/cron.d"
      - "/etc/cron.daily"
      - "/etc/systemd/system"
      - "/etc/ssh"
      - "~/.ssh"
      - "~/.et_modules"
```

5.2 Process Sensor

File: `core/sensors/process_sensor.py`

Monitors process activity by reading `/proc` (Linux) at the configured poll interval. Does NOT use `psutil` — it reads `/proc` directly to minimize dependencies.

Events monitored:

- New process spawned (not in previous poll's process set)

- Process with unexpected parent (parent PID does not match known process trees)
- Process consuming excessive CPU or memory (above configurable threshold)
- Process with suspicious name or command line (embedded and compared against substrate)
- Process running from unusual location (e.g., `/tmp`, `/dev/shm`)
- Process with elevated privileges that should not have them

Embedding strategy: Process events are encoded as feature vectors combining:

- Command name hash
- Full command line hash
- Parent PID relationship (hash of parent→child chain up to depth 3)
- User ID
- CPU percentage (normalized)
- Memory percentage (normalized)
- Process age (seconds since spawn)
- Working directory hash
- Open file descriptor count

Configuration (§11):

```
sensors:
  process:
    enabled: true
    poll_interval_seconds: 10.0
    cpu_threshold_pct: 90.0           # Flag processes above this CPU%
    memory_threshold_pct: 80.0        # Flag processes above this memory%
    known_process_allowlist:          # Never flag these (by command name)
      - "sshd"
      - "systemd"
      - "python3"                     # OpenClaw and modules run as python3
    suspicious_locations:
      - "/tmp"
      - "/dev/shm"
      - "/var/tmp"
```

5.3 Network Sensor

File: `core/sensors/network_sensor.py`

Monitors network connections by reading `/proc/net/tcp`, `/proc/net/tcp6`, `/proc/net/udp`, and `/proc/net/udp6` at the configured poll interval. Also monitors `/proc/net/unix` for suspicious UNIX domain sockets.

Events monitored:

- New outbound connection to unknown IP
- Connection to known-bad port (common C2 ports, crypto mining pools)
- Unexpected listening port opened
- Connection volume spike (potential data exfiltration or DDoS participation)
- DNS resolution to suspicious domain (if `/etc/resolv.conf` monitoring is available)
- Connection from a process that should not have network access

Embedding strategy: Network events are encoded as feature vectors combining:

- Destination IP hash (or subnet hash for /24)
- Destination port (normalized to 0-1 range)
- Protocol (TCP=0, UDP=1)
- Connection state (ESTABLISHED, LISTEN, TIME_WAIT, etc.)
- Associated process PID → command name hash
- Bytes sent/received (if available from `/proc/net/tcp`)
- Time since last connection to same destination

Configuration (§11):

```
sensors:
  network:
    enabled: true
    poll_interval_seconds: 15.0
    suspicious_ports:
      - 4444                          # Metasploit default
      - 5555                          # Common reverse shell
      - 8888                          # Common C2
      - 1337                          # Leet port (script kiddie indicator)
    known_good_destinations:
      - "127.0.0.1"
      - "::1"
    max_outbound_connections: 100     # Flag if exceeded
```

5.4 Dependency Sensor

File: `core/sensors/dependency_sensor.py`

Monitors installed packages and dependencies for supply chain attacks. Runs at a longer poll interval than other sensors (default: 300 seconds / 5 minutes) since package changes are infrequent.

Events monitored:

- New package installed (pip, npm, apt)
- Package version changed
- Package checksum mismatch (if checksums were previously recorded)
- Package from untrusted source
- Package name similar to a popular package (typosquatting detection)

Embedding strategy: Dependency events are encoded as feature vectors combining:

- Package name hash
- Version string hash
- Package manager (pip=0, npm=1, apt=2)
- Source repository hash
- Previous version hash (for upgrades)
- Time since last package change

Implementation: The sensor maintains a snapshot of installed packages (stored in `~/.et_modules/immunis/dependency_snapshot.json`). On each poll, it compares the current package list against the snapshot. Any differences are emitted as events. The snapshot is updated after each poll. On first run with no snapshot, the current package list is recorded as the baseline with no events emitted.

Configuration (§11):

```
sensors:
  dependency:
    enabled: true
    poll_interval_seconds: 300.0
    package_managers:
      - "pip"
      - "npm"
    snapshot_path: "~/.et_modules/immunis/dependency_snapshot.json"
```

5.5 Log Sensor

File: `core/sensors/log_sensor.py`

Monitors system logs for security-relevant events. Tails log files and emits events for patterns that the substrate should learn from.

Events monitored:

- Authentication failure (from `/var/log/auth.log` or `journalctl`)
- Repeated auth failures from same source (brute force pattern)
- `sudo` usage by unexpected users
- SSH key addition or modification
- Service start/stop for non-standard services
- Kernel messages indicating security events (from `dmesg` or `/var/log/kern.log`)

Embedding strategy: Log events are encoded using the hash-based embedding fallback (`_hash_embed()`) from the `OpenClawAdapter` base class. This is because log entries are text, and using the same hash-based deterministic embedding ensures consistency with the substrate's learned topology. If a sentence-transformer model is available, it should be preferred (configurable via §11).

Configuration (§11):

```
  sensors:
    log:
      enabled: true
      poll_interval_seconds: 30.0
      log_sources:
        - "/var/log/auth.log"
        - "/var/log/syslog"
      use_journalctl: true              # Use systemd journal if available
      auth_failure_window_seconds: 300  # Time window for brute force detection
      auth_failure_threshold: 5         # Failures in window before emitting event
```

5.6 Memory Sensor

File: `core/sensors/memory_sensor.py`

Monitors system-wide memory patterns for anomalies. This is the lightest sensor — it reads `/proc/meminfo` and per-process memory maps at a moderate interval.

Events monitored:

- System memory usage spike (sudden increase beyond threshold)
- Individual process memory growth rate anomaly
- Unusual memory mapping patterns (executable pages in unexpected locations)

Embedding strategy: Memory events are encoded as feature vectors from `/proc/meminfo` values (total, free, available, buffers, cached, swap) normalized to 0-1 range plus per-process RSS/VSZ ratios.

Configuration (§11):

```
sensors:
  memory:
    enabled: true
    poll_interval_seconds: 30.0
    system_memory_threshold_pct: 95.0
    process_growth_rate_threshold_mb_per_min: 100.0
```

5.7 Substrate Sensor

File: `core/sensors/substrate_sensor.py`

Monitors the NG-Lite substrates of peer modules for drift and anomalies. This is the “watches the watchers” capability.

Events monitored:

- Substrate weight distribution shift (standard deviation of synapse weights deviates from historical mean)
- Node firing rate anomaly (nodes firing too frequently or too rarely compared to baseline)
- Novelty saturation (substrate reporting consistently high novelty — may indicate poisoning)
- Peer bridge connectivity loss

Implementation: This sensor reads the NG-Lite state file of peer modules (via the peer bridge's shared learning directory). It does NOT modify peer state — it only reads. The `_embed()` method encodes substrate health metrics as a feature vector.

Configuration (§11):

```
sensors:
  substrate:
    enabled: true
    poll_interval_seconds: 60.0
    weight_divergence_threshold: 2.0  # Standard deviations from mean
    novelty_saturation_threshold: 0.95
```

---

6. The Armory (Threat Intelligence Store)

The Armory is Immunis's core vector database for threat intelligence. It is structurally identical to THC's Diagnostic Vector Store (DVS) but stores threat signatures, known false-positive patterns, and response effectiveness records instead of failure signatures and repair records.

File: `core/armory.py`

6.1 Entry Types

```
class ArmoryEntryType(str, Enum):
    """Types of Armory entries."""
    THREAT_SIGNATURE = "THREAT_SIGNATURE"     # Known threat pattern
    FALSE_POSITIVE = "FALSE_POSITIVE"         # Confirmed non-threat
    RESPONSE_RECORD = "RESPONSE_RECORD"       # What response worked for what thr
    BEHAVIORAL_BASELINE = "BEHAVIORAL_BASELINE"  # What "healthy" looks like
    CAUSAL_CHAIN = "CAUSAL_CHAIN"            # Tier 3: recorded attack sequence
```

6.2 Entry Schema

```
@dataclass
class ArmoryEntry:
    """A single entry in The Armory."""
    entry_id: str                     # UUID
    entry_type: ArmoryEntryType
    timestamp: float                  # time.time() of creation
    embedding: np.ndarray             # 384-dim normalized vector
    category: str                     # ThreatClassification category
    severity: str                     # Assessment severity
    metadata: Dict[str, Any]          # Entry-type-specific data
    response_primitive: Optional[str] # What response was used
    response_effectiveness: Optional[float]  # 0.0 to 1.0
    access_count: int = 0             # LRU tracking
    last_accessed: float = 0.0        # LRU tracking
```

6.3 Persistence Format

The Armory persists to `~/.et_modules/immunis/armory.msgpack` using msgpack serialization. This matches THC's DVS persistence format for consistency across the ecosystem.

Msgpack was chosen for small size and speed. The format choice is documented here for the Integration Specs (future document).

Fallback: If msgpack is unavailable (missing dependency), the Armory falls back to JSON at `~/.et_modules/immunis/armory.json` with a warning logged. The fallback path exists for bootstrapping only — production deployments should have msgpack installed.

6.4 Search

Armory search routes through the NG-Lite substrate, identical to THC's DVS search:

1. Embed the query
2. Activate the embedding on the NG-Lite substrate
3. Propagate activation through learned pathways
4. Harvest activated entries
5. Rank by multi-factor score: substrate activation × cosine similarity × recency × response effectiveness
6. Return top-k results

Configuration (§11):

```
armory:
  max_entries: 10000
  persistence_format: "msgpack"       # "msgpack" | "json"
  search_top_k: 10
  eviction_policy: "lru"              # Least recently accessed entries evicted w
```

6.5 Signature Prefilter Source

The Armory's initial signature set is **TBD** — to be determined by A/B testing to assess how an Immunis NG-Lite instance develops when pretrained to some degree with known signatures versus learning a specific system from day one with no pretraining.

The PRD specifies the *interface* for signature loading without mandating a specific source. The Armory accepts signatures via a `load_signatures(path: str)` method that reads a JSON file of signature entries. This allows pluggable signature sources (ClamAV exports, YARA rule conversions, OSSEC signatures, or custom training sets) without coupling Immunis to any specific signature provider.

```
def load_signatures(self, path: str) -> int:
    """Load threat signatures from a JSON file.

    File format:
    [
        {
            "category": "malware",
            "severity": "critical",
            "description": "Known ransomware behavioral signature",
            "embedding": [0.12, -0.05, ...],  # 384-dim
            "metadata": { ... }
        },
        ...
    ]

    Returns:
        Number of signatures loaded.
    """
```

---

7. Response Primitives

Response primitives are Immunis's action vocabulary. Each primitive is a small, validated operation that the Quartermaster can learn to apply. All primitives implement `validate()/execute()` — the same contract as THC's repair primitives.

File: `core/response_primitives.py`

7.1 Base Class

The response primitive base class is structurally identical to THC's `RepairPrimitive`:

```
class ResponsePrimitive(ABC):
    """Abstract base class for all Immunis response primitives.
    Every response primitive MUST implement validate() and execute()
    as separate methods.  validate() MUST be called before execute()
    with no exceptions.  validate() MUST NOT have side effects.
    """

    @property
    def name(self) -> str:
        return self.__class__.__name__

    @property
    def severity_floor(self) -> str:
        """Minimum severity level that can invoke this primitive.
        Override to restrict dangerous primitives to higher severities."""
        return "LOW"

    @abstractmethod
    def validate(self, context: Dict[str, Any]) -> ValidationResult:
        ...

    @abstractmethod
    def execute(self, context: Dict[str, Any]) -> ExecutionResult:
        ...
```

`ValidationResult` and `ExecutionResult` are identical dataclasses to THC's:

```
@dataclass
class ValidationResult:
    passed: bool
    reason: str
    preconditions: Dict[str, Any] = field(default_factory=dict)

@dataclass
class ExecutionResult:
    status: str           # "success" | "partial" | "failed"
    detail: str
    rollback_info: Optional[Dict[str, Any]] = None
    duration_ms: float = 0.0
```

7.2 Built-in Primitives

Immunis ships with seven response primitives. This is the starting vocabulary the substrate learns with. Over time, the substrate learns which primitives are effective for which threat categories.

7.2.1 KillProcess

Severity floor: HIGH

Terminates a process by PID. Sends SIGTERM first, waits `kill_grace_seconds` (default: 5), then SIGKILL if the process is still alive.

validate(): Checks that the PID exists, is not PID 1 or PID of Immunis itself, is not in the `protected_pids` config list, and is not a kernel thread.

execute(): `os.kill(pid, signal.SIGTERM)` → wait → `os.kill(pid, signal.SIGKILL)` if needed. Returns rollback info: `None` (process termination is not reversible).

7.2.2 QuarantineFile

Severity floor: MEDIUM

Moves a suspicious file to the quarantine directory (`~/.et_modules/immunis/quarantine/`) with metadata preserved.

validate(): Checks that the file exists, is not in `protected_paths` config, quarantine directory is writable, and file is not currently open by a protected process.

execute(): `shutil.move(src, quarantine_dir / uuid_filename)` . Writes a metadata sidecar file (`{uuid}.meta.json`) recording original path, permissions, timestamps, and the threat classification that triggered quarantine. Returns rollback info: `{"original_path": str, "quarantine_path": str}` .

7.2.3 BlockConnection

Severity floor: HIGH

Blocks a network connection by adding an iptables rule (Linux). This primitive requires root/sudo access. If not available, it falls back to `AlertOnly` .

validate(): Checks that iptables is available, the user has permission to modify rules, and the target IP/port is not in the `protected_destinations` config list.

execute(): `iptables -A OUTPUT -d {ip} -j DROP` (or `-p {protocol} --dport {port}` for port-based blocks). Returns rollback info: `{"rule": str}` for later removal.

7.2.4 RevokePermissions

Severity floor: MEDIUM

Reduces permissions on a suspicious file. Sets the file to read-only (removes write and execute bits).

validate(): Checks that the file exists, is not in `protected_paths`, and current user has permission to change permissions.

execute(): `os.chmod(path, stat.S_IRUSR | stat.S_IRGRP | stat.S_IROTH)`. Returns rollback info: `{"path": str, "original_mode": int}`.

7.2.5 IsolateModule

Severity floor: CRITICAL

Disconnects a module from the NG-Lite peer bridge by removing its shared learning file from the peer bridge directory. This prevents a compromised module's substrate from poisoning other modules.

validate(): Checks that the target module's peer bridge file exists and the target is not Immunis itself.

execute(): Moves the module's peer bridge file to quarantine. Writes a marker file indicating isolation. Returns rollback info: `{"module_id": str, "peer_file_path": str}`.

7.2.6 AlertOnly

Severity floor: LOW

Logs the threat with full metadata and emits a user feedback request (§9). Takes no autonomous action.

validate(): Always passes.

execute(): Writes to the threat log and queues a user notification. Returns `status="success"` always.

7.2.7 SnapshotForensics

Severity floor: LOW

Captures a forensic snapshot of system state at the time of detection. Saves to `~/.et_modules/immunis/forensics/{timestamp}_{signal_id}/`.

validate(): Checks that the forensics directory is writable and disk space is sufficient (configurable minimum: 100MB, §11).

execute(): Captures:

- Current process list (`/proc` snapshot)
- Open network connections (`/proc/net/*`)
- Recent file system changes (from the file system sensor's buffer)
- Current NG-Lite substrate state summary
- The triggering ThreatSignal and ThreatClassification

Returns rollback info: `{"snapshot_path": str}` (for cleanup).

---

8. The Autonomic Nervous System (Vendored)

The sympathetic/parasympathetic nervous system state is an ecosystem-wide phenomenon. It is NOT owned by any single module. It is implemented as a **vendored state file** that is incorporated into all security-relevant modules.

8.1 Design Rationale

The substrate is the primary communication channel for all learned intelligence. But the sympathetic/parasympathetic state needs to be **fast-read** — checking substrate on every operation for a fight/flight flag would be too slow for a time-sensitive response. A vendored state file provides sub-millisecond read access.

8.2 The Vendored File

File: `ng_autonomic.py`

This file is vendored (copied verbatim) into every module that participates in the autonomic nervous system. Initially that is: Immunis, TrollGuard, and Cricket (when implemented). THC and Bunyan read the state but do not write to it — they are not security modules.

The file provides two functions:

```
"""
NG Autonomic Nervous System — Ecosystem-Wide Threat Level State

This is a VENDORED FILE.  Copy it verbatim into any module that
participates in the autonomic nervous system.  Do NOT modify this
file per-module.  Changes propagate by updating the canonical source
in the NeuroGraph repository and re-vendoring.

The autonomic state file lives at:
    ~/.et_modules/autonomic_state.json
```

```
This location is OUTSIDE any individual module's directory because
the state belongs to the ecosystem, not to any module.

State transitions:
    PARASYMPATHETIC (rest/digest) → normal operations
    SYMPATHETIC (fight/flight) → elevated threat, all modules adjust

When SYMPATHETIC:
    - TrollGuard lowers its suspicious threshold
    - Immunis increases sensor poll frequency
    - THC holds off on auto-executing repairs (wait for threat to clear)
    - All modules increase logging granularity
    - Bunyan switches to maximum-detail causal narrative mode

The file wakes when a security module (Immunis, TrollGuard, Cricket)
is detected in the ecosystem.  If no security module is present, the
autonomic state file is inert.

License: AGPL-3.0
"""

import json
import os
import time
from pathlib import Path
from typing import Optional

_STATE_PATH = Path.home() / ".et_modules" / "autonomic_state.json"
_VALID_STATES = {"PARASYMPATHETIC", "SYMPATHETIC"}

def read_state() -> dict:
    """Read the current autonomic state.  Fast path — ~0.1ms.

    Returns:
        {
            "state": "PARASYMPATHETIC" | "SYMPATHETIC",
            "threat_level": "none" | "low" | "medium" | "high" | "critical",
            "triggered_by": str,          # module_id that set the state
            "timestamp": float,           # when the state was last changed
            "reason": str,                # human-readable reason
        }

    If the state file does not exist, returns PARASYMPATHETIC with
    threat_level "none".  This is the default state.
    """
    if not _STATE_PATH.exists():
```

```python
        return {
            "state": "PARASYMPATHETIC",
            "threat_level": "none",
            "triggered_by": "",
            "timestamp": 0.0,
            "reason": "default — no security module has written state",
        }
    try:
        with open(_STATE_PATH, "r") as f:
            return json.load(f)
    except (json.JSONDecodeError, OSError):
        return {
            "state": "PARASYMPATHETIC",
            "threat_level": "none",
            "triggered_by": "",
            "timestamp": 0.0,
            "reason": "state file unreadable — defaulting to PARASYMPATHETIC",
        }


def write_state(
    state: str,
    threat_level: str,
    triggered_by: str,
    reason: str,
) -> None:
    """Write the autonomic state.  Only security modules should call this.

    Args:
        state: "PARASYMPATHETIC" or "SYMPATHETIC"
        threat_level: "none" | "low" | "medium" | "high" | "critical"
        triggered_by: module_id of the calling module
        reason: human-readable reason for the state change
    """
    if state not in _VALID_STATES:
        raise ValueError(f"Invalid state: {state}. Must be one of {_VALID_STATES}

    _STATE_PATH.parent.mkdir(parents=True, exist_ok=True)
    data = {
        "state": state,
        "threat_level": threat_level,
        "triggered_by": triggered_by,
        "timestamp": time.time(),
        "reason": reason,
    }
    # Atomic write: temp file + rename
    tmp_path = _STATE_PATH.with_suffix(".tmp")
    with open(tmp_path, "w") as f:
        json.dump(data, f)
    os.replace(tmp_path, _STATE_PATH)
```

8.3 Who Writes, Who Reads

Module	Reads State	Writes State
Immunis	✅	✅ (primary writer)
TrollGuard	✅	✅ (can escalate on text-level threats)
Cricket (future)	✅	✅ (can escalate on behavioral violations)
THC	✅ (adjusts repair behavior)	❌
Bunyan (future)	✅ (adjusts logging granularity)	❌

8.4 State Transition Rules

Immunis writes SYMPATHETIC when the Quartermaster assesses a CRITICAL severity event. It writes PARASYMPATHETIC when:

1. The threat has been neutralized (response primitive returned success AND observation window passed with no recurrence), AND
2. No other CRITICAL or HIGH severity events are active.

The autonomic state file includes the `triggered_by` field so that if multiple security modules escalate simultaneously, the last one to de-escalate knows to check whether other modules still need SYMPATHETIC state before writing PARASYMPATHETIC.

---

9. Training Wheels: User Feedback Loop

When Immunis first boots on a fresh system with an empty substrate, it has no learned pathways. It does not know what “healthy” looks like for this specific system. The training wheels ensure Immunis does not take harmful autonomous action before the substrate is mature enough to make confident decisions.

9.1 Observe-Only Boot Phase

On first boot (detected by: no Armory file exists AND substrate has zero recorded outcomes), Immunis enters **observe-only mode**:

- All sensors run normally.
- The Quartermaster pipeline runs through DETECT, CLASSIFY, and ASSESS.
- RESPOND is limited to `AlertOnly` and `SnapshotForensics` regardless of assessed severity.
- No processes are killed, no files are quarantined, no connections are blocked.
- Every event is presented to the user for feedback (§9.2).

Graduation criteria: Immunis exits observe-only mode when ALL of the following are met:

- The Armory contains at least `training_wheels.min_armory_entries` entries (default: 50, §11)
- The substrate has at least `training_wheels.min_substrate_outcomes` recorded outcomes (default: 100, §11)
- At least `training_wheels.min_user_feedbacks` user feedback responses have been received (default: 20, §11)
- At least `training_wheels.min_runtime_hours` hours have elapsed since first boot (default: 24, §11)

These thresholds are configurable. A user who wants to skip training wheels entirely can set all values to 0 in config.yaml.

9.2 User Feedback Mechanism

When Immunis needs user input — either because it's in observe-only mode or because a MEDIUM/HIGH severity event needs confirmation — it presents a feedback popup.

File: `core/feedback.py`

The feedback mechanism writes a JSON event to `~/.et_modules/immunis/feedback_queue.json`. The OpenClaw host or a dedicated UI reads this queue and presents the feedback request to the user.

Feedback request format:

```
{
    "request_id": "uuid",
    "timestamp": 1709078400.0,
    "signal_summary": "New outbound connection detected: PID 4521 (curl) → 185.24
    "category": "exfiltration",
    "severity": "HIGH",
    "substrate_confidence": 0.55,
    "substrate_novelty": 0.72,
    "recommended_action": "block_connection",
    "options": [
        {"id": "threat", "label": "Yes, this is a threat", "action": "execute_rec
        {"id": "safe", "label": "No, this is expected", "action": "mark_false_pos
        {"id": "unsure", "label": "I'm not sure", "action": "snapshot_and_observe
        {"id": "approve", "label": "Approve recommended action", "action": "execu
        {"id": "deny", "label": "Deny recommended action", "action": "alert_only"
    ],
    "status": "pending"
}
```

Feedback response format:

```
{
    "request_id": "uuid",
    "response_timestamp": 1709078500.0,
    "selected_option": "threat",
    "user_notes": "Optional free-text field for additional context"
}
```

When a response is received:

- `"threat"` → Execute the recommended response primitive. Record on substrate as confirmed threat. Strengthen pathways.
- `"safe"` → Record as false positive in Armory. Weaken substrate pathways for this signal pattern.
- `"unsure"` → Snapshot for forensics. Continue observing. Do not strengthen or weaken pathways.
- `"approve"` → Same as "threat".
- `"deny"` → Same as "safe" but without marking as false positive. Alert only.

---

10. NG-Lite Substrate Integration

Immunis integrates with the NeuroGraph ecosystem using the same three-tier architecture as TrollGuard and THC. Tiers 1-3 are operational infrastructure, NOT a future roadmap. All three tiers are implemented from Phase 1 scaffolding.

10.1 Vendored Ecosystem Files

Immunis vendors four files, copied verbatim from the canonical source in the NeuroGraph repository:

1. `ng_lite.py` — The NG-Lite substrate (Hebbian learning, novelty detection)
2. `ng_peer_bridge.py` — Tier 2 peer-to-peer module learning
3. `ng_ecosystem.py` — Tier management (Tier 1 → 2 → 3 auto-upgrade)
4. `openclaw_adapter.py` — OpenClaw skill interface base class

Plus one additional vendored file specific to the security ecosystem:

5. `ng_autonomic.py` — Autonomic nervous system state (§8)

These files are NEVER modified per-module. They are byte-for-byte identical across TrollGuard, THC, and Immunis. When the canonical source is updated, all modules re-vendor. See §16 for the complete vendored files specification.

10.2 Tier Behavior

Tier 1 (Standalone): Immunis's NG-Lite learns locally. Every `record_outcome()` call strengthens or weakens connections within Immunis's own substrate. No cross-module learning.

Tier 2 (Peer): The peer bridge shares Immunis's learning with co-located modules. When Immunis records a threat outcome, the peer bridge propagates that topology change. THC's next substrate query returns different results because Immunis's learning is now part of the shared landscape.

Tier 3 (Full NeuroGraph): Full STDP, hyperedges, and predictive coding. Causal chain recognition (§4.7). Cross-module temporal correlation. This is where zero-day defense through sequence recognition becomes genuinely feasible.

10.3 OpenClaw Hook

File: `immunis_hook.py`

```
"""
Immunis OpenClaw Hook — E-T Systems Standard Integration

Exposes Immunis's full-spectrum system security as an OpenClaw skill,
using the standardized OpenClawAdapter base class.

OpenClaw calls get_instance().on_message(text) on every turn.
```

```
The adapter handles all ecosystem wiring (Tier 1/2/3 learning) and
memory logging.  This file implements what's unique to Immunis:

  - _embed():              Sensor embedding dispatcher / hash fallback
  - _module_on_message():  Check Quartermaster status, report active threats
  - _module_stats():       Immunis-specific telemetry

SKILL.md entry:
    name: immunis
    autoload: true
    hook: immunis_hook.py::get_instance
"""

from openclaw_adapter import OpenClawAdapter

class ImmunisHook(OpenClawAdapter):
    MODULE_ID = "immunis"
    SKILL_NAME = "Immunis System Security"
    WORKSPACE_ENV = "IMMUNIS_WORKSPACE_DIR"
    DEFAULT_WORKSPACE = "~/.openclaw/immunis"

    # ... implementation per OpenClawAdapter contract

_INSTANCE = None
def get_instance():
    global _INSTANCE
    if _INSTANCE is None:
        _INSTANCE = ImmunisHook()
    return _INSTANCE
```

The hook does NOT "serve" security to host AI systems. Host AI systems operate, and that operation flows through the substrate where Immunis participates. The hook's `on_message()` is how Immunis participates in the conversation flow — checking Quartermaster status, reporting active threats in its module results, and recording conversation-context signals to the substrate.

10.4 Hardware Adaptation

Immunis adapts to available hardware. The E-T Systems ecosystem includes hardware detection in another module. Immunis reads hardware capabilities at startup and adjusts:

- **GPU available:** Use sentence-transformer model for sensor embeddings. Higher-quality embeddings = better substrate learning.
- **GPU not available (CPU only):** Use hash-based embedding fallback (`_hash_embed()` from `OpenClawAdapter`). Lower quality but always available and fast.

- **Low memory (< 2GB available):** Reduce Armory max entries, reduce signal buffer size, increase sensor poll intervals.
- **High memory (≥ 8GB available):** Increase Armory max entries, increase signal buffer size, decrease sensor poll intervals for finer-grained detection.

Hardware detection is read from the ecosystem's existing hardware detection capability. Immunis does NOT implement its own hardware detection.

---

11. Configuration (config.yaml)

All user-configurable settings are centralized in a single `config.yaml` file. The system ships with sensible defaults that work on a minimal VPS with no GPU. Every default value specified here is the initial starting state — from the moment Immunis gets input, the substrate begins learning and adjusting what these values mean in the specific system it operates on.

File: `config.yaml`

```
immunis:

  # --- Quartermaster Pipeline (§4) ---
  quartermaster:
    signal_buffer_size: 10000         # Max signals in processing deque
    learn_observation_window: 300     # Seconds to watch for threat recurrence af

  # --- Confidence Thresholds (§4.4) ---
  # These are INITIAL VALUES.  The substrate adjusts from moment one.
  thresholds:
    auto_execute: 0.70                # Above this → auto-execute response primit
    recommend: 0.40                   # Above this → recommend to user
    host_premium: 0.15                # Above this → log + observe

  # --- The Armory (§6) ---
  armory:
    max_entries: 10000
    persistence_format: "msgpack"     # "msgpack" | "json"
    search_top_k: 10
    match_threshold: 0.90             # Cosine similarity for known-signature fas
    eviction_policy: "lru"

  # --- Response Primitives (§7) ---
  response:
    kill_grace_seconds: 5             # SIGTERM → wait → SIGKILL
```

```yaml
  quarantine_dir: "~/.et_modules/immunis/quarantine"
  forensics_dir: "~/.et_modules/immunis/forensics"
  forensics_min_disk_mb: 100        # Minimum free disk for forensics snapshots
  protected_pids: []                # PIDs that KillProcess will never target
  protected_paths:                  # Paths that QuarantineFile/RevokePermissio
    - "/etc/passwd"
    - "/etc/shadow"
    - "/etc/group"
    - "/boot"
  protected_destinations:           # IPs/ports that BlockConnection will never
    - "127.0.0.1"
    - "::1"

# --- Sensors (§5) ---
sensors:
  filesystem:
    enabled: true
    poll_interval_seconds: 5.0
    watched_paths:
      - "/"
    excluded_paths:
      - "/proc"
      - "/sys"
      - "/dev"
      - "/tmp/.immunis_*"
    sensitive_paths:
      - "/etc/cron.d"
      - "/etc/cron.daily"
      - "/etc/systemd/system"
      - "/etc/ssh"
      - "~/.ssh"
      - "~/.et_modules"
  process:
    enabled: true
    poll_interval_seconds: 10.0
    cpu_threshold_pct: 90.0
    memory_threshold_pct: 80.0
    known_process_allowlist:
      - "sshd"
      - "systemd"
      - "python3"
    suspicious_locations:
      - "/tmp"
      - "/dev/shm"
      - "/var/tmp"
  network:
    enabled: true
    poll_interval_seconds: 15.0
    suspicious_ports:
      - 4444
      - 5555
      - 8888
      - 1337
    known_good_destinations:
      - "127.0.0.1"
      - "::1"
    max_outbound_connections: 100
  dependency:
    enabled: true
    poll_interval_seconds: 300.0
    package_managers:
      - "pip"
      - "npm"
    snapshot_path: "~/.et_modules/immunis/dependency_snapshot.json"
  log:
    enabled: true
    poll_interval_seconds: 30.0
    log_sources:
      - "/var/log/auth.log"
      - "/var/log/syslog"
    use_journalctl: true
    auth_failure_window_seconds: 300
    auth_failure_threshold: 5
  memory:
    enabled: true
    poll_interval_seconds: 30.0
    system_memory_threshold_pct: 95.0
    process_growth_rate_threshold_mb_per_min: 100.0
  substrate:
    enabled: true
    poll_interval_seconds: 60.0
    weight_divergence_threshold: 2.0
    novelty_saturation_threshold: 0.95

# --- Training Wheels (§9) ---
training_wheels:
  min_armory_entries: 50
  min_substrate_outcomes: 100
  min_user_feedbacks: 20
  min_runtime_hours: 24

# --- NG-Lite Integration (§10) ---
ng_lite:
  enabled: true
```

```
    module_id: "immunis"
    state_path: "~/.et_modules/immunis/ng_lite_state.json"

  # --- Embedding (§5 sensor embedding) ---
  embedding:
    model: "sentence-transformers/all-MiniLM-L6-v2"
    dim: 384
    device: "auto"                    # "cpu" | "cuda" | "auto" (use GPU if avail
    fallback_to_hash: true            # Use hash embedding if model unavailable

  # --- Checkpointing ---
  checkpoint_interval_seconds: 300    # Auto-save substrate and Armory

  # --- Emergency ---
  emergency:
    kill_switch: false                # Manual local kill switch — rejects all pr
```

Configuration loading: Follows the identical pattern to THC's `core/config.py` :

```
@dataclass
class ImmunisConfig:
    """Full configuration for Immunis.
    All values have PRD-specified defaults.  Override via config.yaml."""

    # ... all fields with defaults matching the YAML above ...

    @classmethod
    def from_yaml(cls, path: Optional[str] = None) -> "ImmunisConfig":
        if path is None:
            path = os.path.join(
                Path.home(), ".et_modules", "immunis", "config.yaml"
            )
        # ... identical loading pattern to HealingCollectiveConfig ...
```

---

12. Data Persistence & Logging

12.1 File Locations

All Immunis persistent data lives under `~/.et_modules/immunis/` :

File	Purpose	Format
	User configuration	

`config.yaml`		YAML
`ng_lite_state.json`	NG-Lite substrate state	JSON
`armory.msgpack`	Threat Intelligence Store	msgpack
`armory.json`	Armory fallback (if msgpack unavailable)	JSON
`dependency_snapshot.json`	Dependency sensor baseline	JSON
`feedback_queue.json`	Pending user feedback requests	JSON
`threat_log.jsonl`	Chronological threat event log	JSONL (append-only)
`quarantine/`	Quarantined files + metadata sidecars	Files + JSON
`forensics/`	Forensic snapshots	Directories
`memory/events.jsonl`	OpenClaw memory event log (standard location)	JSONL

12.2 Threat Log (threat_log.jsonl)

Every signal that reaches ASSESS (stage 3 of the Quartermaster) is logged to `threat_log.jsonl` . This is an append-only JSONL file. Each line is a complete JSON object:

```
{
    "timestamp": 1709078400.0,
    "signal_id": "uuid",
    "sensor_type": "network",
    "event_type": "new_outbound_connection",
    "category": "exfiltration",
    "severity": "HIGH",
    "substrate_confidence": 0.55,
    "substrate_novelty": 0.72,
    "known_signature_match": false,
    "response_primitive": "alert_only",
    "response_status": "success",
    "user_feedback": null,
    "autonomic_state": "PARASYMPATHETIC"
}
```

The threat log serves dual purposes:

1. **Human review:** System administrators can review threat history.
2. **Training data:** Historical events can be used to pre-populate a new Immunis instance's Armory.

12.3 Quarantine Directory

Quarantined files are stored with UUID filenames to prevent name collisions. Each quarantined file has a metadata sidecar:

```
quarantine/
├── a1b2c3d4-e5f6-7890.quarantined      # The actual file (renamed)
├── a1b2c3d4-e5f6-7890.meta.json        # Metadata sidecar
├── b2c3d4e5-f6a7-8901.quarantined
└── b2c3d4e5-f6a7-8901.meta.json
```

Sidecar format:

```
{
    "original_path": "/tmp/suspicious_script.sh",
    "original_permissions": 493,
    "original_owner": "www-data",
    "quarantine_timestamp": 1709078400.0,
    "signal_id": "uuid",
    "category": "malware",
    "severity": "CRITICAL",
    "classification_details": { ... }
}
```

12.4 Checkpointing

Immunis auto-saves its NG-Lite substrate state and Armory at the configured interval (default: 300 seconds). Checkpointing also occurs:

- After every successful response primitive execution
- On SIGTERM signal (graceful shutdown)
- When transitioning between PARASYMPATHETIC and SYMPATHETIC autonomic states

Checkpointing uses atomic write (temp file + `os.replace()`) for consistency. This is the same pattern used by THC.

---

13. Deployment

13.1 Minimum Requirements

- Python 3.10+
- Linux (Immunis reads `/proc` directly for process and network sensors)
- 512 MB available RAM (minimum — hash embeddings, reduced buffer)
- 2 GB available RAM (recommended — sentence-transformer model)
- Root/sudo access (optional — required for BlockConnection primitive; degrades gracefully without it)

13.2 Dependencies

File: `requirements.txt`

```
numpy>=1.24.0
pyyaml>=6.0
msgpack>=1.0.0
watchdog>=3.0.0
```

Optional (improve quality but not required):

```
sentence-transformers>=2.2.0    # Better embeddings than hash fallback
torch>=2.0.0                    # Required by sentence-transformers
```

Explicitly NOT dependencies (do not add these):

- `psutil` — Process monitoring reads `/proc` directly
- `requests` — No HTTP communication between modules
- `flask` / `fastapi` — No web server
- `chromadb` / `faiss` — Armory is a custom vector store (like THC's DVS)

13.3 Quick Start

```
# 1. Clone the repo
git clone https://github.com/greatnorthernfishguy-hub/Immunis.git
cd Immunis

# 2. Install dependencies
```

```
pip install -r requirements.txt

# 3. Install (creates ~/.et_modules/immunis/ and registers with ET Module Manager
./install.sh

# 4. Verify installation
python -c "from immunis_hook import get_instance; print(get_instance().stats())"
```

13.4 install.sh

File: `install.sh`

The install script follows the identical pattern to THC's `install.sh`:

1. Create `~/.et_modules/immunis/` directory structure
2. Create default `config.yaml` if not present
3. Register with the ET Module Manager (`et_module.json`)
4. Create the `~/.et_modules/autonomic_state.json` if not present (shared ecosystem file)
5. Verify vendored files are present and match expected checksums
6. Print installation summary

---

14. Project Structure

```
Immunis/
├── immunis_hook.py              # OpenClawAdapter subclass (§10.3)
├── et_module.json               # Module manifest (§14.1)
├── SKILL.md                     # OpenClaw skill definition
├── core/
│   ├── __init__.py
│   ├── config.py                # YAML config with dataclass loader (§11)
│   ├── quartermaster.py         # The Quartermaster pipeline (§4)
│   ├── armory.py                # The Armory — threat intelligence store (§6)
│   ├── response_primitives.py   # Response primitive ABC + 7 built-ins (§7)
│   ├── feedback.py              # User feedback mechanism (§9)
│   └── sensors/
│       ├── __init__.py
│       ├── base.py              # Sensor ABC (§5)
│       ├── filesystem_sensor.py # File system monitoring (§5.1)
│       ├── process_sensor.py    # Process monitoring (§5.2)
│       ├── network_sensor.py    # Network monitoring (§5.3)
│       ├── dependency_sensor.py # Package/dependency monitoring (§5.4)
│       ├── log_sensor.py        # System log monitoring (§5.5)
│       ├── memory_sensor.py     # Memory monitoring (§5.6)
│       └── substrate_sensor.py  # Peer substrate drift monitoring (§5.7)
├── ng_lite.py                   # Vendored — DO NOT MODIFY
├── ng_peer_bridge.py            # Vendored — DO NOT MODIFY
├── ng_ecosystem.py              # Vendored — DO NOT MODIFY
├── openclaw_adapter.py          # Vendored — DO NOT MODIFY
├── ng_autonomic.py              # Vendored — DO NOT MODIFY (§8)
├── config.yaml                  # Default configuration (§11)
├── install.sh                   # One-click installer (§13.4)
├── requirements.txt             # Python dependencies (§13.2)
├── CHANGELOG.md                 # Version history
├── LICENSE                      # AGPL-3.0
├── README.md
└── tests/
    ├── __init__.py
    ├── test_config.py
    ├── test_quartermaster.py
    ├── test_armory.py
    ├── test_response_primitives.py
    ├── test_feedback.py
    ├── test_sensors.py
    └── test_hook.py
```

14.1 et_module.json

```
{
    "module_id": "immunis",
    "display_name": "Immunis",
    "version": "0.1.0",
    "description": "Full-spectrum system security for the NeuroGraph ecosystem",
    "install_path": "~/Immunis",
    "git_remote": "https://github.com/greatnorthernfishguy-hub/Immunis.git",
    "git_branch": "main",
    "entry_point": "immunis_hook.py",
    "ng_lite_version": "0.1.0",
    "dependencies": [],
    "service_name": "",
    "api_port": 0,
    "author": "E-T Systems / NeuroGraph Foundation",
    "license": "AGPL-3.0",
    "capabilities": [
        "system_security",
        "threat_detection",
        "behavioral_analysis",
        "zero_day_defense"
    ],
    "hooks": ["pre_route", "post_route", "post_response"],
    "ng_config": {
        "threat_threshold": 0.40,
        "block_threshold": 0.70
    },
    "priority": 10
}
```

---

15. Implementation Roadmap

Immunis is built incrementally in phases. Each phase produces a working module that can be deployed and tested. No phase introduces technical debt — each phase is built correctly from the start, with the full tier architecture operational from Phase 1.

Phase 1: Scaffolding + Core Sensors

Deliverables:

- `et_module.json`, `install.sh`, `requirements.txt`, directory structure
- Vendored ecosystem files (ng_lite.py, ng_peer_bridge.py, ng_ecosystem.py, openclaw_adapter.py, ng_autonomic.py)
- `immunis_hook.py` — OpenClawAdapter subclass with singleton `get_instance()`
- `core/config.py` — YAML config with all defaults
- `core/sensors/base.py` — Sensor ABC
- `core/sensors/filesystem_sensor.py` — File system monitoring
- `core/sensors/process_sensor.py` — Process monitoring
- `core/quartermaster.py` — Pipeline stages 1-3 (DETECT, CLASSIFY, ASSESS) with signal buffer
- `core/armory.py` — Armory with msgpack persistence, cosine search, LRU eviction
- `core/response_primitives.py` — ResponsePrimitive ABC + AlertOnly + SnapshotForensics
- `core/feedback.py` — User feedback queue mechanism

- `config.yaml` — Default configuration
- Test suite: config, armory, primitives, sensors, hook

Focus: Get the module running, learning from file system and process events, presenting findings to the user. Observe-only mode (training wheels) is the default.

Phase 2: Full Sensor Suite + Response Primitives

Deliverables:

- `core/sensors/network_sensor.py` — Network monitoring
- `core/sensors/dependency_sensor.py` — Package/dependency monitoring
- `core/sensors/log_sensor.py` — System log monitoring
- `core/sensors/memory_sensor.py` — Memory monitoring
- `core/sensors/substrate_sensor.py` — Peer substrate drift monitoring
- Five additional response primitives: KillProcess, QuarantineFile, BlockConnection, RevokePermissions, IsolateModule
- Quartermaster stages 4-6 (RESPOND, REPORT, LEARN) fully operational
- `ng_autonomic.py` — Autonomic nervous system vendored file
- Autonomic state transitions integrated into Quartermaster

Focus: Full sensor coverage, full response vocabulary, autonomous action capability (with confidence thresholds). The substrate is now learning what "healthy" and "threat" look like across all signal types.

Phase 3: Substrate Intelligence + Causal Chains

Deliverables:

- NG-Lite-augmented Armory search (routes through substrate topology)
- Tier 3 causal chain recording (hyperedge creation for temporally correlated events)
- Armory `load_signatures()` interface for pluggable pretraining
- Behavioral baseline recording and drift detection
- False positive feedback loop refinement
- Comprehensive integration tests with TrollGuard and THC peer bridge scenarios

Focus: The substrate becomes the primary detection engine. Known signatures are fast-path only. The real intelligence is in learned behavioral shapes and temporal sequences. Zero-day defense through sequence recognition is now possible.

Phase 4: Hardening + Community

Deliverables:

- Thread safety audit (sensor polling + Quartermaster pipeline run concurrently)
- Performance profiling and optimization
- Documentation: README, CONTRIBUTING.md, threat model documentation
- GitHub repository, AGPL-3.0 license
- Docker support (optional)

Focus: Production readiness, community contribution infrastructure, security hardening of Immunis itself.

---

16. Vendored Files Specification

16.1 What "Vendored" Means

A vendored file is a file copied verbatim from a canonical source into a module's directory. It is NEVER modified per-module. All modules that vendor the same file have byte-for-byte identical copies.

16.2 Canonical Source

All vendored files originate from the NeuroGraph repository:
`https://github.com/greatnorthernfishguy-hub/NeuroGraph`

16.3 Vendored File Inventory

File	Canonical Path	Purpose
`ng_lite.py`	`NeuroGraph/ng_lite.py`	NG-Lite substrate (Hebbian learning, novelty detection)
`ng_peer_bridge.py`	`NeuroGraph/ng_peer_bridge.py`	Tier 2 peer-to-peer module learning

`ng_ecosystem.py`	`NeuroGraph/ng_ecosystem.py`	Tier management (auto-upgrade 1→2→3)
`openclaw_adapter.py`	`NeuroGraph/openclaw_adapter.py`	OpenClaw skill interface base class
`ng_autonomic.py`	`NeuroGraph/ng_autonomic.py`	Autonomic nervous system state (§8)

16.4 Rules for Vendored Files

1. **NEVER modify a vendored file.** If a vendored file needs a change, the change is made in the canonical source and re-vendored to all modules.
2. **NEVER import from a vendored file in a way that depends on module-specific modifications.** The vendored file's public API is the contract.
3. **NEVER create new vendored files** beyond the five listed above without explicit architectural approval. The coding agent MUST NOT create additional vendored files.
4. **NEVER create wrapper modules** that extend or modify vendored file behavior. If a module needs different behavior, it uses the vendored file's existing extension points (subclassing, configuration, etc.).

16.5 Update Procedure

When the canonical source is updated:

1. The ET Module Manager detects the version mismatch via `ng_lite_version` in `et_module.json`
2. `et-modules update --all` copies the new vendored files from the canonical source
3. Each module's `et_module.json` is updated with the new `ng_lite_version`

---

17. Known Limitations & Threat Model

17.1 Linux Only

Immunis reads `/proc` directly for process and network monitoring. It does not support macOS or Windows. This is a deliberate design choice — the primary deployment target is VPS environments running Linux.

17.2 Root Access Degradation

Without root/sudo access, the BlockConnection primitive cannot add iptables rules. It degrades gracefully to AlertOnly. The coding agent MUST implement this graceful degradation, not fail with an error.

17.3 Sensor Evasion

A sufficiently sophisticated attacker who understands Immunis's sensor architecture could theoretically evade individual sensors. The mitigation is cross-sensor correlation via the substrate — an attack that evades the file system sensor may still be caught by the process sensor or network sensor. The substrate learns correlations across sensor types.

17.4 Substrate Poisoning

Immunis monitors other modules' substrates for drift (§5.7), but who monitors Immunis's own substrate? This is a known limitation. The partial mitigation is Cricket (when implemented), which provides an independent behavioral constraint layer. The full mitigation is user review of threat logs and feedback (§9).

17.5 Cold Start

On a fresh system with no learned pathways, Immunis is limited to known-signature matching (if pretrained) and user feedback. The training wheels (§9) mitigate dangerous false actions during this period, but detection quality during cold start is genuinely lower than a mature Immunis instance.

17.6 Signal Volume

On a busy system, sensors can generate thousands of signals per minute. The Quartermaster's signal buffer (default: 10000) provides backpressure, but sustained signal volume beyond buffer capacity means signals are dropped. The mitigation is configurable poll intervals and sensor filtering.

---

18. Coding Agent Prompt

The following prompt can be provided to a coding agent (Claude Code, Cursor, etc.) to initiate the build. It incorporates all architectural decisions from this PRD.

```
Act as a Senior Python Security Architect implementing Immunis, the
full-spectrum system security module for the E-T Systems / NeuroGraph
ecosystem.
```

```
You MUST read and follow the Immunis PRD (this document) for all
design decisions.  Reference §-numbers in every changelog entry.

CRITICAL CONSTRAINTS:
1. Vendor these files verbatim from TrollGuard's repo (do NOT modify):
   ng_lite.py, ng_peer_bridge.py, ng_ecosystem.py, openclaw_adapter.py
2. Create ng_autonomic.py exactly as specified in PRD §8.2
3. Follow the OpenClawAdapter subclass pattern from trollguard_hook.py
4. Use et_module.json v2 schema matching The Healing Collective
5. NO new APIs, HTTP endpoints, or direct module-to-module communication
6. ALL cross-module communication via vendored substrate files only
7. Config via YAML with dataclass loader (pattern: THC core/config.py)
8. Response primitives use validate()/execute() contract (pattern: THC repair_pri
9. Sensors read system state directly from /proc — no psutil dependency
10. Armory uses msgpack persistence with JSON fallback

Phase 1 deliverables: scaffolding, config, filesystem sensor, process
sensor, Quartermaster stages 1-3, Armory, AlertOnly + SnapshotForensics
primitives, feedback mechanism, OpenClaw hook, tests.

Every file must have a changelog header matching the pattern:
# ---- Changelog ----
# [DATE] AUTHOR — DESCRIPTION
#   What: ...
#   Why:  PRD §X.Y specifies...
#   Settings: ...
#   How:  ...
# -------------------
```

Appendix A: Biological Mapping Reference

Biological System	E-T Systems Module	Role
Skin (perimeter)	TrollGuard	Text-level threat filtering
T-cells, white blood cells	Immunis	Detect and fight active threats
B-cells (antibody creation)	The Healing Collective	Create antibodies + distribute via peer bridge
Tissue repair	The Healing Collective	Diagnose and repair system damage

Sympathetic nervous system (fight/flight)	ng_autonomic.py (vendored)	Ecosystem-wide threat escalation
Parasympathetic nervous system (rest/digest)	ng_autonomic.py (vendored)	Normal operations state
Conscience / prefrontal cortex	Cricket (forthcoming)	Behavioral constraint enforcement
Causal memory / hippocampus	Full NeuroGraph (Tier 3)	Temporal sequence encoding via STDP + hyperedges
Smart logging / sensory narrative	Bunyan (forthcoming)	Causal story format event logging

Appendix B: Cross-Reference to Existing PRDs

Topic	TrollGuard PRD Section	THC PRD Section	Immunis PRD Section
Confidence thresholds	§5.4 (0.3/0.7 traffic light)	§2.2.5 (0.70/0.40/0.15)	§4.4 (0.70/0.40/0.15)
Vendored files	README	§0.1.0 changelog	§16
OpenClaw hook	trollguard_hook.py	healing_collective_hook.py	§10.3
Vector store	skills_db.json / quarantine.json	DVS (core/dvs.py)	Armory (core/armory.py)
Pipeline	4-layer defense	7-step diagnosis	6-stage Quartermaster
Config pattern	config.yaml	core/config.py	§11
Validate/execute contract	N/A	core/repair_primitives.py	§7.1
Emergency stop	§12 kill switch	N/A	§11 kill_switch

END OF DOCUMENT

Immunis PRD v1.0 — E-T Systems / NeuroGraph Foundation February 2026