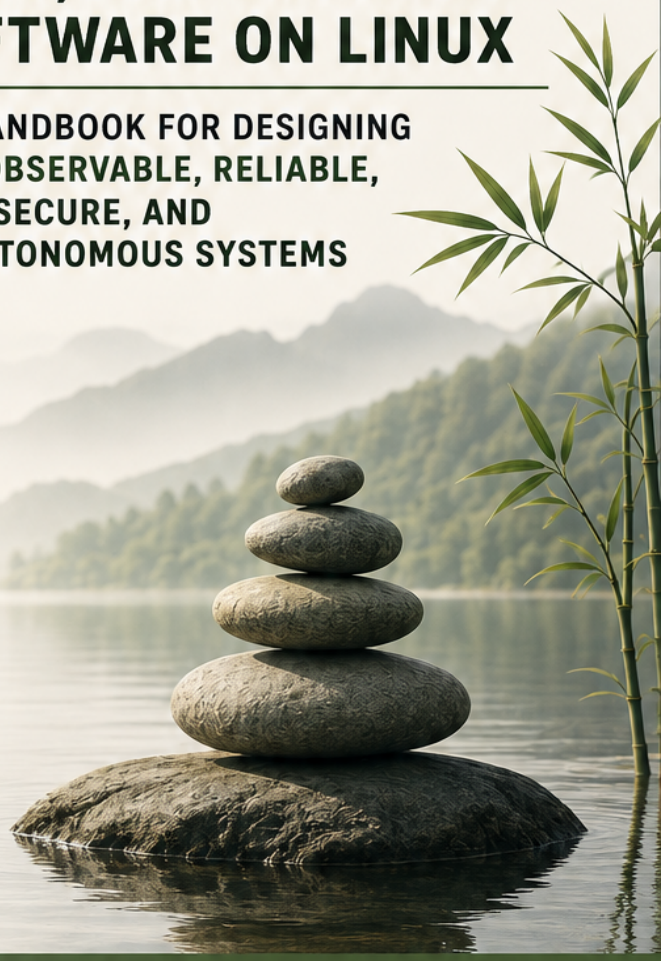
AI-NATIVE SYSTEMS ENGINEERING

BUILDING, RUNNING, AND SECURING AUTONOMOUS SOFTWARE ON LINUX

A PRODUCTION-FOCUSED HANDBOOK FOR DESIGNING USEFUL, CONTROLLABLE, OBSERVABLE, RELIABLE, MAINTAINABLE, SECURE, AND PRODUCTION-READY AUTONOMOUS SYSTEMS

- DESIGN INTELLIGENT AGENT ARCHITECTURES
- BUILD MEMORY AND RETRIEVAL SYSTEMS THAT SCALE
- SECURE AUTONOMOUS SYSTEMS WITH DEFENSE-IN-DEPTH
- GAIN OBSERVABILITY WITH OPENTELEMETRY
- ENGINEER RELIABILITY AND OPERATIONAL EXCELLENCE
- DEPLOY FROM SYSTEMD SERVICES TO KUBERNETES CLUSTERS



LINUX-NATIVE FOUNDATION | OPEN STANDARDS FIRST | SECURITY BY DESIGN | OBSERVABILITY BUILT IN | PRODUCTION READY

DANIEL SCOTT

AI-Native Systems Engineering: Building, Running, and Securing Autonomous Software on Linux

A Production-Focused Handbook for Designing Useful, Controllable, Observable, Reliable, Maintainable, Secure, and Production-Ready Autonomous Systems

Steve Publications

This book is available at https://leanpub.com/ai-nativesystemsengineeringbuildingrunningandsecuringautonomoussoftwareonlinux

This version was published on 2026-08-26

Contents

A Production-Focused Handbook for Designing Useful, Controllable, Observable, Reliable, Maintainable, Secure, and Production-Ready Autonomous Systems **1**

Introduction **2**

Part I: Foundations: What AI-Native Systems Engineering Is and Why It Matters **4**

Chapter 1: Defining AI-Native Systems **5**
What "AI-Native" Means (and Does Not Mean) 5
How AI-Native Differs from Traditional Software, Cloud-Native, MLOps, and Agent Frameworks 6
The Core Properties of Autonomous Software: Agency, Autonomy, Adaptability, Uncertainty 8
Architectural Principles for AI-Native Systems 10
What This Book Builds: The Reference Architecture Roadmap 12

Chapter 2: Linux Foundations for AI Workloads **15**
Process Management, Signals, and Lifecycle Control for Long-Running Daemons 15
Resource Management: cgroups v2, Limits, and Accounting for AI Workloads 20
Namespaces and Isolation Primitives: Building Containment from the Kernel Up 23
GPU and Accelerator Management on Linux: Drivers, MIG, and Resource Slicing 27
Filesystem Layouts, I/O Patterns, and Storage Considerations for AI Systems 31
systemd as the Foundation for Production Deployment 33

Chapter 3: Autonomy, Control, and Determinism **36**
Levels of Autonomy: From Tool-Assisted to Fully Autonomous Operations 36
Designing the Probabilistic-Deterministic Boundary 36
Contracts Between AI Components and Traditional Software 36
Making Agent Actions Idempotent, Auditable, and Reversible 36
Controlling Blast Radius: Scoping Agency and Limiting Damage 36

Part II: Architecture and Core Patterns **38**

Chapter 4: System Architecture for Autonomous Services **39**
Service-Oriented Architecture for AI-Native Systems 39
Event-Driven Patterns: Queues, Topics, and Asynchronous Workflows 39
State Management: Durable State, Checkpointing, and Recovery . . . 39
Workflow Orchestration vs Agent Autonomy: When to Use Each . . . 39
The Reference Architecture: Component Map and Data Flows 39

Chapter 5: Models, Inference, and Runtime Selection **41**
Model Deployment Options: Local, Remote, Hybrid, and Multi-Provider Strategies 41
Inference Runtimes: vLLM, Ollama, llama.cpp, TensorRT-LLM, and Others 41
Performance Characteristics: Latency, Throughput, Memory, and Cost Profiles 41
Structured Outputs, Schemas, and Reliable Parsing 41
Graceful Degradation When Models Fail or Degrade 41

Chapter 6: Tools, APIs, and Function Calling **43**
Tool Definition and Discovery: Schemas, Descriptions, and Capabilities 43
Function Calling Patterns: Direct, Indirect, and Dynamic Tool Selection 43
API Design for AI Consumers: Idempotency, Error Semantics, and Documentation 43
Authentication, Authorization, and Credential Management for Agent Tools 43
Reliability Patterns: Retries, Timeouts, Circuit Breakers, and Fallbacks 43

Chapter 7: Memory, Context, and Retrieval **45**
Short-Term vs Long-Term Memory Architectures 45
Context Engineering: Window Management, Summarization, and Compression 45

Vector Databases and Semantic Search: Architecture and Implementation 45
Retrieval-Augmented Generation (RAG) Patterns and Pitfalls 45
Hybrid Storage: Combining Vector, Relational, and Document Stores . 45

Chapter 8: Agents, Multi-Agent Systems, and Planning 47
What an Agent Actually Is: Definitions, Capabilities, and Misconceptions 47
Single-Agent Architectures: ReAct, Plan-and-Execute, and Variants . 47
When Multi-Agent Systems Help (and When They Hurt) 47
Planning, Reasoning, and Task Decomposition Patterns 47
Inter-Agent Communication, Trust, and Coordination 47

Part III: Security: Designing Defenses for Autonomous Software 48

Chapter 9: Threat Modeling for AI-Native Systems 49
The Expanded Attack Surface: New Vectors Introduced by Autonomy 49
Prompt Injection: Direct, Indirect, and System-Level Attacks 49
Tool Abuse and Privilege Escalation Through Agent Capabilities 49
Data Exfiltration, Credential Leakage, and Information Disclosure . . . 49
Supply Chain Threats: Models, Dependencies, and Knowledge Bases . 49

Chapter 10: Defense in Depth for Autonomous Systems 51
Sandboxing Strategies: Containers, Namespaces, and Process Isolation 51
Linux Security Modules: SELinux and AppArmor for AI Workloads . . 51
System Call Filtering with seccomp: Restricting Agent Capabilities . . 51
Network Security: Segmentation, Egress Controls, and Service Mesh . 51
Secrets Management and Credential Isolation Patterns 51

Chapter 11: Safe Execution, Approval Gates, and Containment 53
Safe Code Execution: Sandboxes, Interpreters, and Restricted Environments 53
Shell Command Safety: Validation, Allowlisting, and Escaping 53
Human-in-the-Loop Controls: Approval Gates for Consequential Actions 53
Kill Switches, Emergency Stops, and Graceful Shutdown Procedures . 53
Containment and Recovery When Agents Are Compromised 53

Part IV: Operations: Running AI-Native Systems in Production 55

Chapter 12: Observability, Testing, and Evaluation 56
Observability Foundations: Logs, Metrics, Traces for AI Systems 56

OpenTelemetry Integration for End-to-End Tracing 56
Testing Strategies for Probabilistic Components and Agent Behavior . 56
Evaluation Frameworks: Correctness, Safety, Reliability, and Performance 56
Simulation and Benchmarking Environments 56

Chapter 13: Reliability, Scalability, and Performance Optimization . . . 58
Fault Tolerance Patterns: Retries, Timeouts, Circuit Breakers, and Idempotency 58
Caching Strategies: Prompt Results, Embeddings, and Vector Search Optimization 58
Concurrency Control and Distributed Systems Concerns 58
Capacity Planning, Cost Engineering, and Performance Optimization 58

Part V: Deployment and Operations: Running in Production 59

Chapter 14: Deployment, Lifecycle Management, and Governance 60
Configuration Management and Environment Strategy 60
CI/CD Pipelines: Building, Testing, and Deploying AI-Native Systems 60
Infrastructure as Code: Terraform, Ansible, and Declarative System Definition 60
Deployment Strategies: systemd Services, Containers, and Kubernetes 60
Upgrades, Rollbacks, and Zero-Downtime Operations 60
Backup, Disaster Recovery, and Incident Response Procedures 61
Governance Frameworks, Compliance, and Ethical Considerations . . 61

Conclusion: The Future of AI-Native Systems Engineering 62
Synthesis: What Makes an AI-Native System Production-Ready 62
The Complete Reference Architecture: Sentinel in Production 62
Deployment Walkthrough: From Clean Host to Running System 62
Security Hardening Checklist 62
Architectural Decision Guidance 62
Forward-Looking: How AI-Native Systems Engineering Will Evolve . . 63

References 64

A Production-Focused Handbook for Designing Useful, Controllable, Observable, Reliable, Maintainable, Secure, and Production-Ready Autonomous Systems

This book teaches you how to design, build, deploy, and operate production-grade autonomous software systems on Linux. It covers the complete lifecycle from architectural foundations through security hardening and operational maturity, using a progressive reference implementation that evolves across every chapter. You will learn Linux internals relevant to AI workloads, inference runtime selection, agent architectures, memory and retrieval systems, threat modeling for autonomous software, defense-in-depth security controls, observability with OpenTelemetry, reliability engineering patterns, and deployment strategies ranging from systemd services to Kubernetes clusters. The goal is not a survey of AI agents but a cohesive systems-engineering handbook grounded in Linux-native, open standards, vendor-neutral approaches that produce real systems you can run, trust, and operate.

Introduction

Consider a system that monitors your infrastructure, detects anomalies, investigates root causes by querying logs and metrics, drafts remediation plans, executes approved fixes, writes post-incident reports, and learns from each event to improve future responses. It operates continuously, makes decisions without waiting for human input on every step, and integrates with dozens of external services through APIs, SSH sessions, and database connections.

This is not science fiction. It is what AI-native systems engineering enables. And it is fundamentally different from building a traditional microservice, deploying a machine learning model, or wiring together an LLM chat interface.

Traditional software engineering produces deterministic systems: given the same inputs, you get the same outputs. Cloud-native engineering optimizes for scalability and resilience of those deterministic systems. MLOps manages the lifecycle of statistical models that transform data into predictions. LLM application development connects prompts to APIs and renders responses in a UI. None of these disciplines fully addresses what happens when software can reason about its goals, select tools dynamically, maintain state across interactions, and take actions with real-world consequences based on probabilistic reasoning rather than fixed rules.

AI-native systems engineering fills that gap. It is the discipline of designing autonomous software that is useful enough to delegate work to, controllable enough to trust in production, observable enough to understand when things go wrong, reliable enough to operate continuously, maintainable enough for a team to evolve over years, and secure enough that compromise does not become catastrophe.

This book takes you from first principles through advanced production architectures. It defines what AI-native systems engineering is and how it differs from related disciplines. It covers Linux internals relevant to AI workloads, inference runtime selection, agent architectures, memory and retrieval systems, threat modeling for autonomous software, defense-in-depth security controls, observability with OpenTelemetry, reliability engineering patterns, and deployment strategies ranging from systemd services to Kubernetes clus-

ters. Throughout, it progressively builds a single reference implementation that demonstrates every major concept in working code.

The technology landscape evolves rapidly. This book focuses on architectural principles, concrete implementation patterns, and production-hardening techniques that remain valid regardless of which specific model or framework is current today. Where specific technologies are discussed, versions, capabilities, and limitations are stated explicitly so you can make informed decisions and update as the ecosystem changes.

Part I: Foundations: What AI-Native Systems Engineering Is and Why It Matters

Chapter 1: Defining AI-Native Systems

What "AI-Native" Means (and Does Not Mean)

The term AI-native has become ubiquitous, which is precisely why it needs careful definition. In casual usage, AI-native describes anything that uses artificial intelligence in some way. A website with a recommendation engine is AI-native. A code editor with autocomplete is AI-native. A support portal with a chatbot is AI-native. Under this broad definition, the term loses all analytical power.

For the purpose of this book, AI-native has a specific technical meaning. An AI-native system is one in which artificial intelligence is embedded into the core control logic and decision-making pathways rather than serving as an add-on feature or isolated capability. In an AI-native system:

- The AI model participates in control flow decisions, not just data transformation. It chooses which operations to perform, in what order, and when to stop.
- The system maintains persistent state across interactions, including memory of past actions, learned patterns, and contextual knowledge that influences future behavior.
- Autonomy is a designed property, not an accident. The system operates independently within defined boundaries, making decisions and taking actions without requiring human intervention for every step.
- Probabilistic reasoning is first-class. The architecture explicitly accounts for the fact that AI components produce outputs with uncertainty, and deterministic software compensates through validation, constraints, retries, and fallbacks.

Consider the contrast with AI-enabled systems. An AI-enabled system uses AI as a tool within an otherwise deterministic workflow. A fraud detection service that scores transactions and flags suspicious ones is AI-enabled: the model produces a score, but the surrounding system decides what to do with it

based on fixed rules. The control flow is predetermined; the AI simply provides input data.

An AI-native fraud response system would be different. It receives an alert, investigates by examining transaction patterns, account history, and related events, formulates hypotheses about what happened, queries additional data sources as needed, determines whether intervention is required, selects the appropriate response from available tools, executes that response, documents its reasoning, and updates its understanding for future cases. The control flow emerges from the AI's reasoning process rather than being hardcoded in advance.

This distinction matters because it changes everything about how you design, secure, test, and operate the system. Deterministic systems are predictable: you can reason about their behavior by reading the code. AI-native systems require a fundamentally different approach to engineering that treats uncertainty as a core constraint rather than an edge case.

What AI-native does not mean is that everything in your system should be driven by AI models. The most effective AI-native architectures keep deterministic logic where determinism matters and delegate to probabilistic reasoning only where it adds value. Authentication is deterministic. Authorization policies are deterministic. Database transactions are deterministic. Error handling, retries, timeouts, and circuit breakers are deterministic. These components form the reliable infrastructure that makes autonomous AI behavior safe and predictable.

The phrase AI-native also does not imply that you should build everything from scratch using raw model APIs. Modern AI-native development leverages frameworks, protocols, and abstractions that encode proven patterns. The Model Context Protocol standardizes how applications connect to tools and data sources [1]. OpenTelemetry provides vendor-neutral observability instrumentation [2]. Structured output mechanisms with JSON schema enforcement make model responses parseable and reliable [3]. These building blocks are not optional extras; they are essential infrastructure for production AI-native systems.

How AI-Native Differs from Traditional Software, Cloud-Native, MLOps, and Agent Frameworks

To understand what is unique about AI-native systems engineering, it helps to contrast it with adjacent disciplines that share vocabulary but solve different problems.

Traditional software engineering builds deterministic systems governed by explicit logic. Given input X, the system executes a defined sequence of operations and produces output Y. Testing involves verifying behavior against specifications. Security focuses on protecting data, controlling access, and preventing unauthorized actions. Reliability engineering ensures the system continues to function correctly under load and during failures. All of these practices remain essential in AI-native systems, but they are insufficient on their own because they do not address the fundamental challenge of integrating probabilistic reasoning into a production architecture.

Cloud-native engineering optimizes for scalability, resilience, and operational efficiency in distributed environments. It provides patterns for containerization, orchestration, service discovery, configuration management, and continuous deployment. Cloud-native principles apply directly to AI-native systems: you still need containers, you still need Kubernetes or systemd, you still need CI/CD pipelines. But cloud-native engineering assumes deterministic workloads. It does not provide guidance for architecting around model latency variability, handling nondeterministic outputs, securing autonomous decision-making, or designing contracts between probabilistic and deterministic components.

MLOps manages the lifecycle of machine learning models: training, validation, deployment, monitoring, and retraining. It addresses data versioning, experiment tracking, feature stores, model registries, drift detection, and infrastructure for GPU-intensive workloads. MLOps is relevant to AI-native systems when you are training or fine-tuning custom models, but it is not the same thing. Most AI-native systems today consume pre-trained foundation models through APIs rather than training their own. More importantly, MLOps focuses on model performance metrics like accuracy and latency; it does not address agent behavior, tool use, autonomy boundaries, or the security implications of giving software the ability to act on its own reasoning.

LLM application development connects large language models to user

interfaces and external APIs through prompts and function calling. It covers prompt engineering, context management, retrieval-augmented generation, and building conversational interfaces. This is a subset of AI-native systems engineering: every LLM application uses AI, but not every LLM application is AI-native in the sense defined above. A chatbot that answers questions based on retrieved documents is an LLM application; it becomes AI-native when it can independently investigate problems, execute multi-step workflows, maintain state across sessions, and take actions with real-world consequences.

Agent frameworks like LangGraph, CrewAI, and AutoGen provide abstractions for building autonomous agents: state management, tool calling, planning loops, and multi-agent coordination [4]. These frameworks accelerate development by encoding common patterns, but they are implementation tools, not a discipline. AI-native systems engineering encompasses framework usage while also addressing the broader architectural, security, operational, and organizational concerns that determine whether an agent-based system works reliably in production or fails spectacularly.

The key insight is that AI-native systems engineering sits at the intersection of all these disciplines while adding unique requirements of its own. It requires:

- Systems thinking to integrate probabilistic and deterministic components into a coherent architecture
- Security expertise adapted to autonomous software with expanded attack surfaces
- Operational maturity for observing, testing, and maintaining behavior that is not fully predictable
- Architectural discipline to define autonomy boundaries, enforce constraints, and control blast radius

You cannot treat AI-native systems as just another type of microservice or just another ML deployment. They require a distinct engineering approach that starts from the unique properties of autonomous, reasoning software and builds outward to infrastructure, security, and operations.

The Core Properties of Autonomous Software: Agency, Autonomy, Adaptability, Uncertainty

Autonomous software exhibits four core properties that distinguish it from traditional applications and determine how it must be engineered. Under-

standing these properties is essential because every architectural decision, security control, and operational practice in this book derives from one or more of them.

Agency is the capacity to take actions with external effects. An agentic system does not just compute answers; it performs operations: calling APIs, writing files, sending messages, executing commands, modifying configurations. Agency transforms AI from a reasoning engine into an actor in the world. This property creates both opportunity and risk. It enables automation of complex workflows that would require human coordination, but it also means mistakes or exploitation have real consequences. Engineering for agency requires designing tools that are safe to use, enforcing authorization boundaries that limit what each agent can do, making actions idempotent so retries do not cause harm, and ensuring every action is auditable so you can reconstruct what happened after the fact.

Autonomy is the capacity to make decisions without requiring human approval for every step. An autonomous system determines its own control flow: it decides which tools to call, when to gather more information, when to take action, and when a task is complete. Autonomy exists on a spectrum from fully supervised (human approves every action) to fully independent (system operates without human involvement). Production systems typically use selective autonomy: the system acts independently within defined boundaries but escalates to humans for high-risk decisions, unusual situations, or actions exceeding configured thresholds. Engineering for autonomy requires defining clear boundaries that constrain what the system can decide on its own, designing escalation paths for situations beyond those boundaries, and building trust through observability so operators understand when autonomy is working correctly versus when intervention is needed.

Adaptability is the capacity to modify behavior in response to changing conditions, new information, or feedback. An adaptive system learns from experience: it remembers past interactions, updates its understanding of the environment, adjusts strategies based on outcomes, and incorporates new knowledge over time. Adaptability distinguishes AI-native systems from traditional automation scripts that follow fixed procedures regardless of context. Engineering for adaptability requires persistent memory architectures that store and retrieve relevant information across sessions, feedback mechanisms that allow humans to correct mistakes and reinforce desired behavior, versioning of prompts, policies, and configurations so changes are tracked and

reversible, and safeguards against harmful adaptation such as drift toward unsafe behavior or incorporation of poisoned knowledge.

Uncertainty is the unavoidable consequence of probabilistic reasoning. Unlike deterministic software where outputs follow directly from inputs through fixed logic, AI models produce outputs with inherent variability. The same prompt may yield different responses. Reasoning can be flawed. Tool selection can be incorrect. Context can be misinterpreted. Uncertainty is not a bug to eliminate; it is a fundamental property of how foundation models work. Engineering for uncertainty requires treating model outputs as untrusted until validated, designing retry and recovery mechanisms for when things go wrong, constraining outputs through structured schemas so errors are detectable rather than silently corrupting downstream systems, building in human oversight for consequential decisions where uncertainty cannot be tolerated, and architecting graceful degradation paths when models fail, produce low-quality output, or become unavailable.

These four properties interact in ways that create emergent complexity. An agentic system with autonomy will take actions based on uncertain reasoning. An adaptive system will change its behavior over time, potentially in unpredictable ways. The combination means that what works in testing may not work in production, and what works today may break tomorrow as models update, environments change, or attackers discover new exploitation paths. This is why AI-native systems engineering cannot rely on traditional verification approaches alone. It requires continuous observation, evaluation, and adaptation of the system itself.

Architectural Principles for AI-Native Systems

The properties above imply a set of architectural principles that guide every design decision in this book. These principles are not arbitrary preferences; they are derived from the operational requirements of production autonomous software.

Principle 1: Probabilistic components must be contained by deterministic infrastructure. The system boundary around any AI model should be enforced by deterministic code that validates inputs, constrains outputs, authorizes actions, handles errors, and manages state. Models make decisions; infrastructure enforces rules. Never allow probabilistic reasoning to bypass security controls, authorization checks, or operational safeguards.

Principle 2: Autonomy must be bounded and explicit. Every autonomous capability must have a clearly defined scope: what actions are allowed, what resources can be accessed, what thresholds trigger escalation, what conditions cause termination. Boundaries should be enforced programmatically through configuration, not relied upon the model to respect through prompting alone. The principle of least agency applies: grant only the minimum autonomy required for each function.

Principle 3: All agent actions must be idempotent, auditable, and reversible where possible. Idempotency ensures that retries do not cause duplicate or compounding effects. Auditability means every action is logged with sufficient context to reconstruct what happened and why. Reversibility allows undoing mistakes when detection occurs after execution. These properties are non-negotiable for production systems.

Principle 4: Observability must cover both system behavior and reasoning processes. Traditional metrics, logs, and traces are necessary but insufficient. You must also observe model inputs and outputs, tool call sequences, decision paths, confidence levels, and behavioral patterns. This dual-layer observability enables debugging when things go wrong and evaluation of whether the system is behaving correctly over time [5].

Principle 5: Security must be designed for autonomous capabilities, not just static applications. Traditional application security assumes a fixed attack surface. Autonomous software dynamically explores its environment, calls tools, processes untrusted data, and makes decisions based on retrieved information. This creates unique vulnerabilities including prompt injection through external content, tool abuse, credential leakage through model outputs, and lateral movement through agent-granted access. Security controls must account for these dynamic behaviors [6].

Principle 6: Failure is expected; resilience is mandatory. Models fail. APIs timeout. Retrieved data is incorrect. Reasoning goes wrong. External services return errors. The system must handle all of these gracefully without cascading failures, data corruption, or unbounded resource consumption. Resilience patterns include retries with backoff, circuit breakers for failing dependencies, fallback models or strategies, bounded context windows to prevent runaway loops, and kill switches for emergency shutdown.

Principle 7: Architecture should prefer simplicity over complexity until proven otherwise. Multi-agent systems, sophisticated planning algorithms,

and elaborate memory architectures are tempting but often unnecessary. Start with the simplest design that solves the problem: a single agent with well-defined tools and clear stopping conditions. Add complexity only when requirements genuinely demand it, and always measure whether the added complexity improves outcomes or just adds failure modes.

These principles form the foundation for every architectural pattern, security control, and implementation decision in this book. They are not theoretical ideals; they are operational necessities derived from what happens when autonomous software runs in production environments with real data, real users, and real consequences.

What This Book Builds: The Reference Architecture Roadmap

This book progresses through a single reference implementation called Sentinel that demonstrates every major concept in working code. Sentinel is designed as a generic automation agent capable of monitoring systems, executing approved workflows, integrating with external services, maintaining durable state, and operating securely within defined boundaries. It is not tied to any specific use case so the patterns apply broadly, but it is concrete enough that you can run it, extend it, and adapt it for your own needs.

The reference implementation evolves across all fourteen chapters of this book. Each chapter adds capabilities while maintaining backward compatibility with code introduced earlier. By the end, Sentinel is a production-hardened autonomous service with security controls, observability, testing, deployment infrastructure, and operational procedures that demonstrate how to build real AI-native systems on Linux.

The technology choices prioritize Linux-native, open standards, vendor-neutral approaches:

- Language: Python 3.12+ for application code; Bash for scripts and automation
- Operating system: Ubuntu 24.04 LTS or Debian 12 as reference distributions
- Process management: systemd for native service deployment

- Containerization: Podman shown primarily (rootless by default, no daemon), with Docker equivalents noted where relevant
- Model serving: Ollama for local development and simple deployments; vLLM for production GPU inference; OpenAI-compatible API abstraction layer for remote models
- Vector store: Qdrant for production deployments; ChromaDB for prototyping
- Database: SQLite for simple deployments; PostgreSQL for multi-instance production
- Message queue: NATS with JetStream for event-driven communication and durable messaging
- Observability: OpenTelemetry for tracing and metrics; Prometheus and Grafana for visualization
- Security: SELinux or AppArmor depending on distribution; seccomp filters; firewall rules via nftables

The architecture follows a layered design that separates concerns:

- The daemon layer runs as a systemd service, manages lifecycle, handles signals, and supervises worker processes
- The agent core implements the reasoning loop: receiving tasks, planning steps, selecting tools, executing actions, maintaining state
- The tool framework provides a standardized interface for external integrations: APIs, filesystem operations, command execution, database queries
- The memory subsystem manages short-term context windows and long-term persistent storage including vector-based retrieval
- The security layer enforces authorization policies, validates actions, manages secrets, and applies sandboxing controls
- The observability layer instruments all components with OpenTelemetry spans, metrics, and structured logs

Each chapter builds specific parts of this architecture. Chapters 1 through 3 establish conceptual foundations without code. Chapter 4 introduces the daemon skeleton and systemd integration. Chapter 5 adds model integration with local and remote inference support. Chapter 6 implements the tool framework and function calling infrastructure. Chapter 7 builds the memory subsystem with vector retrieval. Chapter 8 completes the agent core with planning logic.

Chapters 9 through 11 add security hardening, approval gates, and containment controls. Chapters 12 through 14 cover observability, reliability engineering, deployment, and operations.

The code examples are complete and working, not pseudocode or disconnected snippets. Each chapter provides exact commands to build, run, test, and inspect the system at that stage of development. When educational simplifications are used for clarity, they are explicitly identified before being hardened for production use. By the final chapter, you will have a fully functional autonomous service that demonstrates all the principles and patterns discussed throughout the book.

This approach ensures that concepts are not presented in isolation but as parts of a coherent whole. You see how security controls integrate with agent behavior, how observability instrumentation supports debugging, how deployment infrastructure enables reliable operation, and how all components work together to produce software that is useful, controllable, observable, reliable, maintainable, secure, and production-ready on Linux.

Chapter 2: Linux Foundations for AI Workloads

Process Management, Signals, and Lifecycle Control for Long-Running Daemons

AI-native systems run as long-lived services, not ephemeral scripts. They monitor conditions continuously, respond to events asynchronously, maintain persistent state across hours or days of operation, and must survive restarts without losing critical information. This requires understanding how Linux manages processes, signals, and lifecycles at a level deeper than running Python in the foreground with nohup.

A daemon is a background process that operates independently of any controlling terminal. Modern Linux systems managed by systemd do not require applications to implement traditional double-fork daemonization patterns. Instead, systemd handles process supervision directly: it starts the executable as a regular process, tracks its lifecycle, manages restarts on failure, captures output for logging, and delivers signals for graceful shutdown. The application should run in the foreground, handle termination signals properly, and let systemd do the rest [7].

Signal handling is critical for production daemons. Linux uses signals to communicate events between processes and the kernel. For AI-native systems, three signals matter most:

- SIGTERM (signal 15) requests graceful shutdown. The process should stop accepting new work, complete or checkpoint in-progress tasks, flush buffers, close connections cleanly, and exit within a reasonable time.
- SIGINT (signal 2) is the interactive interrupt, typically generated by Ctrl+C. In production it often maps to the same handler as SIGTERM.
- SIGHUP (signal 1) traditionally means hangup but is commonly repurposed for configuration reload without restart.

A well-behaved daemon installs signal handlers that coordinate shutdown across all components. Consider this minimal pattern for a Python-based AI-native service:

```python
# sentinel/core/lifecycle.py
import asyncio
import signal
import sys
from typing import Callable, Awaitable


class LifecycleManager:
    """Manages graceful startup and shutdown of the daemon."""

    def __init__(self):
        self._running = False
        self._shutdown_event = asyncio.Event()
        self._pre_shutdown_hooks: list[Callable[[], Awaitable[None]]] = []
        self._post_shutdown_hooks: list[Callable[[], Awaitable[None]]] = []

    def on_pre_shutdown(self, hook: Callable[[], Awaitable[None]]) -> None:
        """Register a callback to run before shutdown begins."""
        self._pre_shutdown_hooks.append(hook)

    def on_post_shutdown(self, hook: Callable[[], Awaitable[None]]) -> None:
        """Register a callback to run after shutdown completes."""
        self._post_shutdown_hooks.append(hook)

    async def setup_signals(self, loop: asyncio.AbstractEventLoop) -> None:
        """Install signal handlers for graceful shutdown."""
        for sig in (signal.SIGTERM, signal.SIGINT):
            loop.add_signal_handler(
                sig,
                lambda s=sig: asyncio.create_task(self._handle_shutdown(s))
            )
        # Optional: SIGHUP for config reload
        loop.add_signal_handler(
            signal.SIGHUP,
            lambda: asyncio.create_task(self._handle_reload())
        )

    async def _handle_shutdown(self, sig: int) -> None:
        """Handle shutdown signal by triggering graceful exit."""
        if self._shutdown_event.is_set():
            return  # Already shutting down
        print(f"Received {signal.strsignal(sig)}, initiating graceful
        ↪  shutdown...", file=sys.stderr)
        self._shutdown_event.set()

```

```
    async def _handle_reload(self) -> None:
        """Handle SIGHUP for configuration reload."""
        print("Received SIGHUP, reloading configuration...", file=sys.stderr)
        # Implementation depends on your config system

    @property
    def is_running(self) -> bool:
        return self._running and not self._shutdown_event.is_set()

    async def run_until_shutdown(
        self,
        main_task: Awaitable[None],
        timeout: float = 30.0
    ) -> None:
        """Run the main task until shutdown is requested."""
        self._running = True
        try:
            # Run main task and shutdown wait concurrently
            main_handle = asyncio.create_task(main_task)
            shutdown_handle = asyncio.create_task(self._shutdown_event.wait())

            done, pending = await asyncio.wait(
                {main_handle, shutdown_handle},
                return_when=asyncio.FIRST_COMPLETED
            )

            if shutdown_handle in done:
                # Shutdown requested while main was running
                print("Completing pre-shutdown hooks...", file=sys.stderr)
                for hook in self._pre_shutdown_hooks:
                    try:
                        await asyncio.wait_for(hook(), timeout=timeout /
                        ↪  max(len(self._pre_shutdown_hooks), 1))
                    except asyncio.TimeoutError:
                        print(f"Pre-shutdown hook timed out:
                        ↪  {hook.__qualname__}", file=sys.stderr)
                    except Exception as e:
                        print(f"Pre-shutdown hook failed: {hook.__qualname__}:
                        ↪  {e}", file=sys.stderr)

                # Cancel main task gracefully
                main_handle.cancel()
                try:
                    await main_handle
                except asyncio.CancelledError:
                    pass

            self._running = False

        finally:
```

```
            print("Running post-shutdown hooks...", file=sys.stderr)
            for hook in self._post_shutdown_hooks:
                try:
                    await asyncio.wait_for(hook(), timeout=timeout /
                    ↪  max(len(self._post_shutdown_hooks), 1))
                except asyncio.TimeoutError:
                    print(f"Post-shutdown hook timed out: {hook.__qualname__}",
                    ↪  file=sys.stderr)
                except Exception as e:
                    print(f"Post-shutdown hook failed: {hook.__qualname__}:
                    ↪  {e}", file=sys.stderr)


async def main() -> None:
    """Application entry point demonstrating lifecycle management."""
    loop = asyncio.get_running_loop()
    lifecycle = LifecycleManager()

    # Register shutdown hooks
    @lifecycle.on_pre_shutdown
    async def stop_workers():
        print("Stopping background workers...")
        # Signal workers to finish current tasks

    @lifecycle.on_pre_shutdown
    async def checkpoint_state():
        print("Checkpointing agent state...")
        # Save durable state to disk or database

    @lifecycle.on_post_shutdown
    async def close_connections():
        print("Closing database and API connections...")
        # Clean up resources

    await lifecycle.setup_signals(loop)

    async def run_daemon():
        """Main daemon loop."""
        while lifecycle.is_running:
            # Process tasks, handle events, etc.
            await asyncio.sleep(1)

    await lifecycle.run_until_shutdown(run_daemon(), timeout=30.0)
    print("Shutdown complete.")


if __name__ == "__main__":
    asyncio.run(main())
```

This pattern ensures that when systemd sends SIGTERM during mainte-

nance, upgrades, or failures, the daemon completes in-progress work, saves state, and exits cleanly rather than dying abruptly with potentially corrupted data or orphaned operations.

The corresponding systemd service unit ties process management to the operating system:

```
# /etc/systemd/system/sentinel.service
[Unit]
Description=Sentinel AI-Native Autonomous Service
Documentation=https://github.com/example/sentinel
After=network-online.target nss-lookup.target
Wants=network-online.target

[Service]
Type=exec
User=sentinel
Group=sentinel
WorkingDirectory=/opt/sentinel
ExecStart=/opt/sentinel/venv/bin/python -m sentinel.main
Restart=on-failure
RestartSec=5s
StartLimitBurst=5
StartLimitIntervalSec=60

# Environment and configuration
Environment=PYTHONUNBUFFERED=1
EnvironmentFile=/etc/sentinel/environment
ConfigFilesOptional=/etc/sentinel/config.yaml

# Resource limits (supplement to cgroups)
MemoryMax=4G
TasksMax=256

# Security hardening
NoNewPrivileges=true
ProtectSystem=strict
ProtectHome=read-only
PrivateTmp=true
ReadWritePaths=/var/lib/sentinel /var/log/sentinel
ProtectKernelTunables=true
ProtectKernelModules=true
ProtectControlGroups=true

# Logging
StandardOutput=journal
StandardError=journal
SyslogIdentifier=sentinel

```

```
[Install]
WantedBy=multi-user.target
```

Key directives explained:

- Type=exec tells systemd the service is running when ExecStart completes its exec call. This is more reliable than Type=simple for tracking process state.
- Restart=on-failure combined with StartLimitBurst prevents runaway restart loops while ensuring recovery from transient crashes.
- NoNewPrivileges=true prevents privilege escalation through setuid/setgid binaries.
- ProtectSystem=strict makes the filesystem read-only except for explicitly writable paths, limiting damage from compromised processes.
- StandardOutput=journal sends all output to systemd journal for centralized logging and querying with journalctl.

To install and enable this service:

```
# Create dedicated user
sudo adduser --system --group --home /var/lib/sentinel --no-create-home
↪  sentinel

# Create directories
sudo mkdir -p /opt/sentinel /etc/sentinel /var/lib/sentinel /var/log/sentinel
sudo chown -R sentinel:sentinel /var/lib/sentinel /var/log/sentinel

# Install service file
sudo cp sentinel.service /etc/systemd/system/
sudo systemctl daemon-reload

# Enable and start
sudo systemctl enable --now sentinel

# Check status
systemctl status sentinel

# View logs
journalctl -u sentinel -f
```

This foundation ensures that your AI-native system runs as a proper Linux service with reliable lifecycle management, automatic recovery from crashes, and integration with system logging. Every subsequent layer of the architecture depends on this basic process model working correctly.

Resource Management: cgroups v2, Limits, and Accounting for AI Workloads

AI workloads have distinctive resource characteristics: they consume significant CPU during inference, require large amounts of memory for model weights and context windows, generate bursty I/O patterns, and may need GPU access that must be shared fairly across multiple services. Without proper resource management, a single runaway agent can exhaust system resources, starve other processes, trigger out-of-memory kills, or destabilize the entire host.

Control groups (cgroups) are the Linux kernel mechanism for organizing processes into hierarchical groups and controlling their resource usage [8]. cgroups v2, now the default on modern distributions including Ubuntu 24.04 and Debian 12, provides a unified hierarchy that is simpler and more consistent than the fragmented v1 design.

The unified hierarchy is mounted at /sys/fs/cgroup by default. Processes are organized in directory trees, and each cgroup exposes control files for configuring limits:

```
# Verify cgroups v2 is active
mount | grep cgroup2
# Should show: cgroup2 on /sys/fs/cgroup type cgroup2 (...)

# Create a cgroup for the sentinel service
sudo mkdir -p /sys/fs/cgroup/sentinel

# Enable controllers for this subtree
echo "+cpu +memory +io +pids" | sudo tee
↪  /sys/fs/cgroup/sentinel/cgroup.subtree_control

# Set memory limit to 4GB (soft limit that throttles before OOM)
echo "4294967296" | sudo tee /sys/fs/cgroup/sentinel/memory.high

# Set hard memory ceiling at 5GB (process killed if exceeded)
echo "5368709120" | sudo tee /sys/fs/cgroup/sentinel/memory.max

# Set CPU weight (relative share; default is 100, range 1-10000)
echo "200" | sudo tee /sys/fs/cgroup/sentinel/cpu.weight

# Limit to max 50% of one CPU core
echo "50000 100000" | sudo tee /sys/fs/cgroup/sentinel/cpu.max

```

```
# Limit number of processes/threads
echo "256" | sudo tee /sys/fs/cgroup/sentinel/pids.max

# Move the sentinel process into this cgroup
echo $(pgrep -f "sentinel.main") | sudo tee
↪  /sys/fs/cgroup/sentinel/cgroup.procs
```

In practice, systemd manages cgroups automatically for services. The resource directives in the service unit file translate directly to cgroup settings:

- MemoryMax maps to memory.max (hard limit) and memory.high (throttle point)
- CPUQuota and CPUWeight map to cpu.max and cpu.weight
- TasksMax maps to pids.max
- IOWeight and IOReadBandwidthMax map to io controllers

This means the systemd unit file shown earlier already configures cgroups without manual intervention. However, understanding the underlying mechanism is valuable for debugging resource issues and configuring limits that systemd does not directly expose.

For AI workloads specifically, memory management is critical. A typical production deployment might run multiple components: the agent daemon, an inference server (vLLM or Ollama), a vector database, a message queue, and supporting services. Each needs its own cgroup with appropriate limits:

```
# /etc/systemd/system/vllm.service (inference server)
[Service]
MemoryMax=16G
MemoryHigh=14G
CPUWeight=500
DeviceAllow=/dev/nvidia* rwm
```

The memory.high setting is particularly important for AI workloads. When a cgroup reaches memory.high, the kernel throttles allocations rather than immediately killing the process. This gives the application time to release memory (for example, by evicting cached model weights or reducing context window size) before hitting the hard limit at memory.max where OOM kill occurs. For inference servers that can dynamically adjust batch sizes, this throttling behavior prevents unnecessary restarts during temporary memory pressure.

GPU resource management requires different mechanisms. cgroups can limit CPU and memory but do not directly partition GPU resources. NVIDIA provides Multi-Instance GPU (MIG) for datacenter GPUs (A100, H100, L40S) that partitions a physical GPU into fully isolated instances with dedicated compute cores, memory controllers, and cache [9]. Each MIG instance appears as a separate CUDA device:

```
# Enable MIG mode on an A100 GPU (requires reboot or driver reload)
sudo nvidia-smi -mig 1

# Create MIG instances (example: three 20GB instances)
sudo nvidia-smi mig -gi-all -cgi 2g.20gb --count=3

# Verify created instances
nvidia-smi mig -li

# Use a specific MIG instance in a container or process
CUDA_VISIBLE_DEVICES=mig-1 set +e vllm serve ...
```

MIG provides hardware-level isolation: each instance has its own memory path, L2 cache banks, and compute resources. A fault or resource exhaustion in one instance does not affect others. This makes MIG ideal for multi-tenant AI infrastructure where different services or customers share GPU hardware.

For consumer GPUs without MIG support (RTX 4090, etc.), time-slicing is the alternative: multiple processes share the same GPU through CUDA's scheduling. This provides no memory isolation and can cause interference between workloads, but it works on any NVIDIA GPU. The practical approach for non-MIG hardware is to assign specific GPUs to specific services using CUDA_VISIBLE_DEVICES and rely on cgroups for CPU/memory limits:

```
# /etc/systemd/system/vllm.service
[Service]
Environment=CUDA_VISIBLE_DEVICES=0
MemoryMax=24G
```

Understanding and configuring resource management correctly prevents the most common operational failures in AI-native systems: memory exhaustion killing critical processes, CPU starvation causing inference timeouts, and GPU contention making latency unpredictable. These controls form the foundation for reliable multi-service deployments on shared hardware.

Namespaces and Isolation Primitives: Building Containment from the Kernel Up

Linux namespaces provide isolation by giving processes their own view of system resources. A process inside a namespace sees only the resources allocated to that namespace, not the full system. This is the fundamental technology underlying containers and provides essential containment for AI-native systems where different components should not interfere with each other or escape their designated boundaries.

Six namespace types are relevant for AI workloads:

- PID namespace: Isolates process IDs. Processes in a PID namespace see only their own processes, not system-wide PIDs.
- Network namespace: Isolates network stacks. Each namespace has its own interfaces, routes, firewall rules, and port space.
- Mount namespace: Isolates filesystem mount points. Changes to mounts in one namespace are invisible to others.
- User namespace: Maps user IDs between namespaces. A process can be root inside a user namespace while running as an unprivileged user on the host.
- UTS namespace: Isolates hostname and domain name.
- IPC namespace: Isolates inter-process communication resources (shared memory, message queues, semaphores).

For AI-native systems, isolation serves multiple purposes. It limits blast radius if a component is compromised, prevents noisy neighbor interference between services, enables running untrusted code safely, and supports multi-tenant deployments where different customers or teams share infrastructure.

Containers use namespaces automatically. When you run Podman or Docker, the runtime creates a set of namespaces for the container and configures them according to your options:

```
# Run a container with default namespace isolation
podman run -d \
  --name vllm-server \
  --memory=16g \
  --cpus=8 \
  --device=/dev/nvidia0 \
  --network=host \
  vllm/vllm-openai:latest \
  --model meta-llama/Llama-3.1-8B-Instruct

# Run with full network isolation (separate network namespace)
podman run -d \
  --name isolated-agent \
  --network=bridge \
  --publish=8080:8080 \
  myagent:latest

# Run with user namespace remapping (root inside maps to unprivileged host user)
podman run -d \
  --userns=auto \
  --name sandboxed-tool-executor \
  tool-executor:latest
```

For the Sentinel reference implementation, namespaces are used in several ways. The main agent daemon runs directly on the host managed by systemd with minimal namespace isolation (just cgroups for resource limits). This gives it direct access to system resources it needs for monitoring and management tasks while still being constrained by resource limits and security policies.

Tool execution that involves running untrusted or semi-trusted code uses container namespaces for stronger isolation:

```
# sentinel/tools/sandboxed_executor.py
import subprocess
import json
from dataclasses import dataclass


@dataclass
class SandboxResult:
    stdout: str
    stderr: str
    returncode: int
    duration_seconds: float


class PodmanSandboxExecutor:
```

```
    """Executes code in isolated Podman containers."""

    def __init__(
        self,
        base_image: str = "python:3.12-slim",
        memory_limit: str = "512m",
        cpu_limit: str = "1.0",
        timeout_seconds: int = 60
    ):
        self.base_image = base_image
        self.memory_limit = memory_limit
        self.cpu_limit = cpu_limit
        self.timeout_seconds = timeout_seconds

    async def execute(self, code: str, input_data: dict) -> SandboxResult:
        """Execute Python code in an isolated container."""
        import time

        # Write code to temporary file
        import tempfile
        with tempfile.NamedTemporaryFile(mode='w', suffix='.py', delete=False)
        ↪  as f:
            f.write(code)
            script_path = f.name

        start = time.monotonic()

        try:
            cmd = [
                "podman", "run", "--rm",
                "--read-only",
                "--tmpfs", "/tmp:size=64m,noexec,nosuid",
                "--cap-drop=ALL",
                "--security-opt=no-new-privileges",
                f"--memory={self.memory_limit}",
                f"--cpus={self.cpu_limit}",
                "--network=none",  # No network access by default
                "-v", f"{script_path}:/run/code.py:ro",
                self.base_image,
                "python", "/run/code.py"
            ]

            result = subprocess.run(
                cmd,
                input=json.dumps(input_data).encode(),
                capture_output=True,
                timeout=self.timeout_seconds
            )

            duration = time.monotonic() - start
```

```
            return SandboxResult(
                stdout=result.stdout.decode(errors='replace'),
                stderr=result.stderr.decode(errors='replace'),
                returncode=result.returncode,
                duration_seconds=duration
            )

        except subprocess.TimeoutExpired:
            return SandboxResult(
                stdout="",
                stderr=f"Execution timed out after {self.timeout_seconds}s",
                returncode=-1,
                duration_seconds=self.timeout_seconds
            )
        finally:
            import os
            try:
                os.unlink(script_path)
            except OSError:
                pass
```

This sandboxed executor runs arbitrary Python code in containers with read-only filesystems, no network access, dropped capabilities, and resource limits. Even if the executed code contains malicious logic, it cannot escape the container namespace to affect the host system or other services.

Namespaces are also used for network isolation between components. The Sentinel architecture places different services in separate network namespaces connected through controlled routing:

- Agent daemon namespace: Can communicate with all internal services and approved external endpoints
- Tool executor namespace: Limited to specific tool APIs, no direct internet access
- Inference server namespace: Only accepts connections from the agent daemon on designated ports
- External-facing namespace (if applicable): Public endpoint that forwards requests through authentication to internal services

This network segmentation prevents lateral movement if one component is compromised. An attacker who gains control of the inference server cannot directly access the agent's database or external API credentials because those resources are in different network namespaces with restricted routing.

GPU and Accelerator Management on Linux: Drivers, MIG, and Resource Slicing

AI workloads depend on GPUs for efficient inference, and GPU management on Linux requires understanding driver installation, device access, memory allocation, and multi-tenant resource sharing. Incorrect configuration leads to performance problems, resource contention, or complete failure of AI components.

NVIDIA GPU support on Linux consists of several layers:

1. Kernel module (nvidia.ko): Provides device drivers loaded into the kernel
2. User-space libraries (CUDA toolkit): Enables applications to use GPU compute
3. Management tools (nvidia-smi, nvml): Monitor and configure GPU resources
4. Container runtime integration (NVIDIA Container Toolkit): Allows containers to access GPUs

For Ubuntu 24.04 LTS, installation follows this sequence:

```
# Update package lists
sudo apt update

# Install kernel headers for your running kernel
sudo apt install -y linux-headers-$(uname -r) build-essential

# Add NVIDIA repository and install driver (example: 550 series for newer GPUs)
sudo apt install -y software-properties-common
sudo add-apt-repository ppa:graphics-drivers/ppa -y
sudo apt update
sudo apt install -y nvidia-driver-550

# Reboot to load the driver
sudo reboot

# Verify installation
nvidia-smi
```

The output of nvidia-smi shows GPU model, driver version, CUDA version, memory usage, and running processes. This is your primary tool for monitoring GPU health and utilization.

For containerized inference servers, the NVIDIA Container Toolkit enables GPU passthrough:

```
# Install NVIDIA Container Toolkit on Ubuntu 24.04
curl -fsSL https://nvidia.github.io/libnvidia-container/gpgkey | sudo gpg
    --dearmor -o /usr/share/keyrings/nvidia-container-toolkit-keyring.gpg
curl -s -L https://nvidia.github.io/libnvidia-container/stable/deb/nvidia-con
    tainer-toolkit.list | sed 's#deb https://#deb
    [signed-by=/usr/share/keyrings/nvidia-container-toolkit-keyring.gpg]
    https://#g' | sudo tee
    /etc/apt/sources.list.d/nvidia-container-toolkit.list
sudo apt update
sudo apt install -y nvidia-container-toolkit

# Configure Podman to use NVIDIA runtime
sudo nvidia-ctk runtime configure --runtime=crun
sudo systemctl restart podman

# Test GPU access in container
podman run --rm --device=/dev/nvidia0 nvcr.io/nvidia/k8s/cuda-sample:nbody
    nbody -gpu -benchmark
```

GPU memory management is critical for AI workloads. Model weights are loaded into GPU memory and remain there for the lifetime of the inference process. A 7B parameter model in FP16 precision requires approximately 14 GB of VRAM; an 8B model in FP16 needs roughly 16 GB. Quantization reduces these requirements: INT4 quantization can fit a 7B model in about 4-5 GB of VRAM, enabling deployment on consumer GPUs.

The inference runtime choice affects memory usage and performance characteristics:

- vLLM uses PagedAttention to manage KV cache memory efficiently, enabling high throughput with many concurrent requests [10]. It is designed for production GPU serving with continuous batching.
- Ollama prioritizes simplicity and ease of use, managing model downloads and serving through a single binary. It works well for development and low-concurrency deployments but lacks the advanced batching features needed for high-throughput production.
- llama.cpp focuses on CPU inference and quantized models, making it suitable for hardware without GPUs or for running smaller models alongside GPU workloads [11].

For the Sentinel reference implementation, the architecture supports multiple inference backends through a common interface:

```python
# sentinel/inference/base.py
from abc import ABC, abstractmethod
from dataclasses import dataclass


@dataclass
class InferenceRequest:
    prompt: str
    system_prompt: str = ""
    temperature: float = 0.7
    max_tokens: int = 2048
    model: str = "default"


@dataclass
class InferenceResponse:
    content: str
    tokens_used: int
    model: str
    latency_seconds: float


class InferenceBackend(ABC):
    """Abstract base class for inference backends."""

    @abstractmethod
    async def generate(self, request: InferenceRequest) -> InferenceResponse:
        pass

    @abstractmethod
    async def health_check(self) -> bool:
        pass
```

This abstraction allows switching between local and remote models without changing agent logic. The backend selection is configuration-driven:

```
# /etc/sentinel/config.yaml
inference:
  primary:
    type: vllm
    url: "http://localhost:8000/v1"
    model: "meta-llama/Llama-3.1-8B-Instruct"
    timeout_seconds: 60
  fallback:
    type: openai_compatible
    url: "https://api.example.com/v1"
    api_key_env: "FALLBACK_API_KEY"
    model: "gpt-4o-mini"
```

GPU monitoring should be integrated into the observability stack. The NVIDIA DCGM exporter provides Prometheus-compatible metrics for GPU temperature, utilization, memory usage, power consumption, and errors:

```
# Install DCGM exporter
podman run -d \
  --name dcgm-exporter \
  --gpus all \
  --network=host \
  --privileged \
  nvcr.io/nvidia/k8s/dcgm-exporter:latest
```

These metrics enable alerting on GPU health issues before they cause inference failures and provide capacity planning data for scaling decisions.

Filesystem Layouts, I/O Patterns, and Storage Considerations for AI Systems

AI-native systems have specific storage requirements that differ from traditional applications. They need fast access to model weights (which can be many gigabytes), low-latency retrieval from vector databases, durable logging of agent actions, and efficient handling of large context windows. Filesystem choice and layout directly impact performance and reliability.

Recommended directory structure for the Sentinel reference implementation:

```
/opt/sentinel/              # Application code and dependencies
  venv/                     # Python virtual environment
  src/                      # Source code (sentinel package)
  tests/                    # Test suite

/etc/sentinel/              # Configuration files
  config.yaml               # Main configuration
  environment               # Environment variables for systemd
  prompts/                  # System prompts and templates

/var/lib/sentinel/          # Variable data owned by the service
  state/                    # Agent state and checkpoints
  memory/                   # Long-term memory storage
  cache/                    # Model output caches, embedding caches
  vector-store/             # Vector database data directory

/var/log/sentinel/          # Log files (if not using journal only)
  agent.log                 # Agent execution logs
  tools.log                 # Tool call logs
  errors.log                # Error-specific logs

/run/sentinel/              # Runtime files (PID, sockets, locks)
```

This layout follows Linux Filesystem Hierarchy Standard conventions: configuration in /etc, variable data in /var/lib, logs in /var/log, runtime files in /run. It enables clean separation of concerns and simplifies backup strategies (configuration is small and version-controlled; state data needs regular backups; logs can be rotated aggressively).

For I/O-intensive workloads like vector databases and checkpointing, consider filesystem choices:

- ext4: Default on most distributions, reliable and well-understood. Adequate for most AI workloads.
- XFS: Better performance for large files and parallel I/O patterns. Good choice for GPU nodes with high-throughput model loading.
- Btrfs: Copy-on-write features enable snapshots for backup and rollback. Useful if you need point-in-time recovery of vector stores or state databases.

SSD storage is strongly recommended for AI workloads. Model loading from HDDs can take minutes; SSDs reduce this to seconds. NVMe drives provide the best latency for vector database queries that require random access patterns across large indexes.

For production deployments with significant data, consider dedicated storage for different workload types:

- Fast NVMe for vector databases and hot caches
- Standard SSD for model weights (loaded once, read many times)
- Higher-capacity storage for logs, backups, and archival data

Filesystem monitoring should track disk usage, I/O latency, and error rates. Alerts on these metrics prevent silent failures where the system continues running but degrades due to slow or failing storage:

```
# Monitor disk space (alert if >85% used)
df -h /var/lib/sentinel | tail -1 | awk '{print $5}' | tr -d '%'

# Check I/O wait
iostat -x 1 3 | grep "avg-cpu" | awk '{print $5}'

# Check for filesystem errors in dmesg
dmesg | grep -i "error.*sd\|error.*nvme"
```

systemd as the Foundation for Production Deployment

systemd is the init system and service manager on virtually all modern Linux distributions. For AI-native systems deployed on Linux, systemd provides essential capabilities that should be leveraged rather than bypassed: process supervision, dependency management, resource control via cgroups, logging integration, socket activation, timer-based scheduling, and standardized service lifecycle operations.

The sentinel.service unit file shown earlier demonstrates production-ready systemd configuration. Beyond basic service management, systemd provides advanced features relevant to AI-native deployments:

Path watchers enable filesystem-triggered actions without polling:

```
# /etc/systemd/system/sentinel-config-watcher.path
[Unit]
Description=Watch for Sentinel configuration changes

[Path]
PathChanged=/etc/sentinel/config.yaml
Unit=sentinel-config-reload.service

[Install]
WantedBy=multi-user.target
```

```
# /etc/systemd/system/sentinel-config-reload.service
[Unit]
Description=Reload Sentinel configuration

[Service]
Type=oneshot
ExecStart=/bin/kill -HUP $(cat /run/sentinel.pid)
```

Timers provide reliable scheduling as a replacement for cron:

```
# /etc/systemd/system/sentinel-maintenance.timer
[Unit]
Description=Daily Sentinel maintenance tasks

[Timer]
OnCalendar=*-*-* 03:00:00
Persistent=true
Unit=sentinel-maintenance.service

[Install]
WantedBy=timers.target
```

Socket activation enables on-demand service startup, useful for inference servers that should only run when needed to conserve GPU resources:

```
# /etc/systemd/system/vllm.socket
[Unit]
Description=vLLM Inference Server Socket

[Socket]
ListenStream=8000
Accept=false

[Install]
WantedBy=sockets.target
```

```
# /etc/systemd/system/vllm.service
[Unit]
Description=vLLM Inference Server
After=vllm.socket
Requires=vllm.socket

[Service]
Type=simple
ExecStart=/opt/vllm/venv/bin/python -m vllm.entrypoints.openai.api_server ...
```

systemd also integrates with security controls. The sandboxing directives in service units (NoNewPrivileges, ProtectSystem, PrivateTmp, etc.) apply kernel-level restrictions that are more reliable than application-enforced security. Combined with SELinux or AppArmor policies and seccomp filters discussed later, systemd provides a defense-in-depth foundation for production deployment.

Understanding and properly configuring these Linux foundations is not optional overhead; it is essential infrastructure that determines whether your AI-native system operates reliably in production or fails unpredictably under real-world conditions. Every subsequent chapter builds on these fundamentals: agent architectures depend on reliable process management, security controls depend on namespaces and cgroups, observability depends on systemd journal integration, and deployment strategies depend on understanding how Linux manages services and resources.

Chapter 3: Autonomy, Control, and Determinism

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-nativesystemsengineeringbuildingrunningandsecurin

Levels of Autonomy: From Tool-Assisted to Fully Autonomous Operations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-nativesystemsengineeringbuildingrunningandsecurin

Designing the Probabilistic-Deterministic Boundary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-nativesystemsengineeringbuildingrunningandsecurin

Contracts Between AI Components and Traditional Software

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-nativesystemsengineeringbuildingrunningandsecurin

Making Agent Actions Idempotent, Auditable, and Reversible

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-nativesystemsengineeringbuildingrunningandsecurin

Controlling Blast Radius: Scoping Agency and Limiting Damage

Part II: Architecture and Core Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-nativesystemsengineeringbuildingrunningandsecurin

Chapter 4: System Architecture for Autonomous Services

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-nativesystemsengineeringbuildingrunningandsecurin

Service-Oriented Architecture for AI-Native Systems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-nativesystemsengineeringbuildingrunningandsecurin

Event-Driven Patterns: Queues, Topics, and Asynchronous Workflows

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-nativesystemsengineeringbuildingrunningandsecurin

State Management: Durable State, Checkpointing, and Recovery

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-nativesystemsengineeringbuildingrunningandsecurin

Workflow Orchestration vs Agent Autonomy: When to Use Each

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-nativesystemsengineeringbuildingrunningandsecurin

The Reference Architecture: Component Map and Data Flows

Chapter 5: Models, Inference, and Runtime Selection

Model Deployment Options: Local, Remote, Hybrid, and Multi-Provider Strategies

Inference Runtimes: vLLM, Ollama, llama.cpp, TensorRT-LLM, and Others

Performance Characteristics: Latency, Throughput, Memory, and Cost Profiles

Structured Outputs, Schemas, and Reliable Parsing

Graceful Degradation When Models Fail or Degrade

Chapter 6: Tools, APIs, and Function Calling

Tool Definition and Discovery: Schemas, Descriptions, and Capabilities

Function Calling Patterns: Direct, Indirect, and Dynamic Tool Selection

API Design for AI Consumers: Idempotency, Error Semantics, and Documentation

Authentication, Authorization, and Credential Management for Agent Tools

Reliability Patterns: Retries, Timeouts, Circuit Breakers, and Fallbacks

Chapter 7: Memory, Context, and Retrieval

Short-Term vs Long-Term Memory Architectures

Context Engineering: Window Management, Summarization, and Compression

Vector Databases and Semantic Search: Architecture and Implementation

Retrieval-Augmented Generation (RAG) Patterns and Pitfalls

Hybrid Storage: Combining Vector, Relational, and Document Stores

Chapter 8: Agents, Multi-Agent Systems, and Planning

What an Agent Actually Is: Definitions, Capabilities, and Misconceptions

Single-Agent Architectures: ReAct, Plan-and-Execute, and Variants

When Multi-Agent Systems Help (and When They Hurt)

Planning, Reasoning, and Task Decomposition Patterns

Inter-Agent Communication, Trust, and Coordination

Part III: Security: Designing Defenses for Autonomous Software

Chapter 9: Threat Modeling for AI-Native Systems

The Expanded Attack Surface: New Vectors Introduced by Autonomy

Prompt Injection: Direct, Indirect, and System-Level Attacks

Tool Abuse and Privilege Escalation Through Agent Capabilities

Data Exfiltration, Credential Leakage, and Information Disclosure

Supply Chain Threats: Models, Dependencies, and Knowledge Bases

Chapter 10: Defense in Depth for Autonomous Systems

Sandboxing Strategies: Containers, Namespaces, and Process Isolation

Linux Security Modules: SELinux and AppArmor for AI Workloads

System Call Filtering with seccomp: Restricting Agent Capabilities

Network Security: Segmentation, Egress Controls, and Service Mesh

Secrets Management and Credential Isolation Patterns

Chapter 11: Safe Execution, Approval Gates, and Containment

Safe Code Execution: Sandboxes, Interpreters, and Restricted Environments

Shell Command Safety: Validation, Allowlisting, and Escaping

Human-in-the-Loop Controls: Approval Gates for Consequential Actions

Kill Switches, Emergency Stops, and Graceful Shutdown Procedures

Containment and Recovery When Agents Are Compromised

Part IV: Operations: Running AI-Native Systems in Production

Chapter 12: Observability, Testing, and Evaluation

Observability Foundations: Logs, Metrics, Traces for AI Systems

OpenTelemetry Integration for End-to-End Tracing

Testing Strategies for Probabilistic Components and Agent Behavior

Evaluation Frameworks: Correctness, Safety, Reliability, and Performance

Simulation and Benchmarking Environments

Chapter 13: Reliability, Scalability, and Performance Optimization

Fault Tolerance Patterns: Retries, Timeouts, Circuit Breakers, and Idempotency

Caching Strategies: Prompt Results, Embeddings, and Vector Search Optimization

Concurrency Control and Distributed Systems Concerns

Capacity Planning, Cost Engineering, and Performance Optimization

Part V: Deployment and Operations: Running in Production

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-nativesystemsengineeringbuildingrunningandsecurin

Chapter 14: Deployment, Lifecycle Management, and Governance

Configuration Management and Environment Strategy

CI/CD Pipelines: Building, Testing, and Deploying AI-Native Systems

Infrastructure as Code: Terraform, Ansible, and Declarative System Definition

Deployment Strategies: systemd Services, Containers, and Kubernetes

Upgrades, Rollbacks, and Zero-Downtime Operations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-nativesystemsengineeringbuildingrunningandsecurin

Backup, Disaster Recovery, and Incident Response Procedures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-nativesystemsengineeringbuildingrunningandsecurin

Governance Frameworks, Compliance, and Ethical Considerations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-nativesystemsengineeringbuildingrunningandsecurin

Conclusion: The Future of AI-Native Systems Engineering

Synthesis: What Makes an AI-Native System Production-Ready

The Complete Reference Architecture: Sentinel in Production

Deployment Walkthrough: From Clean Host to Running System

Security Hardening Checklist

Architectural Decision Guidance

Forward-Looking: How AI-Native Systems Engineering Will Evolve

References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-nativesystemsengineeringbuildingrunningandsecurin