Kailash OS — An Operating System for Building and Securing AI

Shaineel Singh (shain.singh@owasp.org)

kailash-os project · Melbourne, Australia

Revisions 1.3–1.4: at 1.3 the platform layer formerly BUILD was named OPS (categories O-1…O-7): "BUILD" collides with the build-the-distribution senses of the word. At 1.4 the release-naming scheme is pinned to the calendar release-index form (`YYYY.NN` ; staged cuts `2026.01-alpha.1` , the bare tag reserved for GA) — replacing the earlier semver-flavoured scheme before any tag was ever cut. Category scope, ordering, and counts are unchanged from Revision 1.2.

Revision 1.5: the paper gains the authorship statement (Section 8). Content otherwise unchanged from Revision 1.4.

Revision 1.5 — 5 November 2026 · EASE formatting pass · Work in progress. Implementation plan companion: "Kailash OS Implementation Plan & Strategy" (v2.3-draft).

Abstract. Machine-learning systems are now routinely attacked, misconfigured, and abused in production, yet the practitioners who build those systems and the practitioners who attack them work in disjoint environments: builders in language-specific virtual environments with per-project state, breakers on rolling general-purpose penetration-testing distributions. We present Kailash OS, a Linux distribution, built declaratively on NixOS with Nix flakes, that packages the full lifecycle of AI security work in a single reproducible environment: offensive tooling (large-language-model red teaming, prompt injection, agent and Model Context Protocol (MCP) attack tooling, model supply-chain analysis, adversarial ML), defensive tooling (guardrail frameworks, evaluation harnesses, telemetry, model provenance and signing), and the underlying platform (MLOps, vector stores, container orchestration, local inference). Kailash is organised into a CORE substrate plus a four-layer taxonomy of 31 categories spanning approximately 340 curated tool slots, in deliberate imitation of Kali Linux's category navigation and distribution model, but with declarative, bit-for-bit reproducible builds that Kali's mutable packaging cannot provide. We describe the design principles, the taxonomy, the flake-based architecture (a two-repository layout with a tool manifest driving six system views), the release engineering of a reproducible operating system — a release train whose version is the lockfile, refreshed monthly rather than continuously and released on a six-week cadence — and the verification bindings into the OWASP standards stack (ASVS 5.0, AISVS 1.0, MLSVS, and the LLM Top 10 2026), which turn assurance requirements into executable commands. The distribution is additionally designed as a teaching environment: because the build platform and the attack tooling are co-installed and pinned together, builders can attempt the attacks against their own lab deployments without leaving their build environment, closing the gap between building and breaking. It is likewise built for autonomous operators: every catalog query, environment provision, and assurance command has a machine-readable form, and tools whose safety level is active-exploit sit behind scoped, human-minted capability grants so that an unsupervised agent cannot escalate into them. The distribution ships offensive tooling strictly for authorised security work: red-team engagements with written authorisation, academic research, and defender validation.

Keywords — AI security · machine learning security · LLM red teaming · prompt injection · agentic systems · MCP security · OWASP AISVS · OWASP ASVS · MLSVS · NixOS · Nix flakes · reproducible builds · MLOps.

Table of Contents

1. Introduction
    - 1.1 The problem: two professions, two environments
    - 1.2 The design thesis in one sentence
    - 1.3 Contributions
    - 1.4 Paper structure
2. Background and Related Work
    - 2.1 The Kali model, distilled
    - 2.2 Prior Nix attempts at security distributions
    - 2.3 The AI-security landscape the taxonomy must cover
    - 2.4 Related distributions and their constraints
    - 2.5 The verification standards
3. Why Nix, and Why the Kali Shape
    - 3.1 The two requirements in tension
    - 3.2 Why Nix specifically, and what it costs
    - 3.3 Why the Kali shape: the ease of a category-navigated distribution
    - 3.4 Declaratively choosing a desktop environment
4. The Taxonomy
    - 4.1 CORE + four tool layers, 31 categories
    - 4.2 The mapping to ATLAS and OWASP
    - 4.3 The verification-standard mapping
    - 4.4 Category design decisions
    - 4.5 What is deliberately shipped, and what isn't
5. Architecture
    - 5.1 Two repositories, one website
    - 5.2 The module tree as the taxonomy
    - 5.3 The manifest: one truth, six views
    - 5.4 Packaging the three populations
    - 5.5 Platform deployment and the attack pairing
    - 5.6 The CUDA case, stated precisely
6. Release Engineering: Is Declarative Harder to Release?
    - 6.1 What a "release" is, concretely
    - 6.2 Why this is easier than the mutable-packaging loop, not harder
    - 6.3 The mechanisms that make the loop trustworthy
    - 6.4 The cost ledger of declarative release
7. The Lab Thesis: Building and Breaking Together
    - 7.1 Why builders learn to break
    - 7.2 The mechanism: the pairing
    - 7.3 What the lab thesis costs

- 7.4 Agent operability: the OS as something an agent can spin up, build, and use

8. Safety, Governance, and Licensing
9. Limitations and Future Work
10. Conclusion
11. References

1. Introduction

1.1 The problem: two professions, two environments

The security industry has a mature pattern for tool distribution. Kali Linux – descended from WHAX/SLAX through BackTrack and maintained by Kali's team – delivers a curated, pre-installed, category-organised collection of security tooling, distributed as live ISOs and organised so that a practitioner can navigate by task. Its model has been proven for over a decade: install once, boot anywhere, and find the right tool in the right category at the right version.

Machine-learning engineering grew up with its own distribution moment: Anaconda. One installer delivered Python, the numeric stack, and the environment manager that made data science reproducible-enough for people to learn it. The analogy matters: what made Anaconda transformative was not the software itself (all of it was already available) but the packaging – a single, curated, onboarding-quality entry point.

AI security has neither. The modern attack surface – LLM APIs, retrieval-augmented generation, agentic systems, MCP servers, model registries, inference engines – is not even mapped in Kali's taxonomy, let alone packaged. The tooling exists (garak, PyRIT, promptfoo, Adversarial Robustness Toolbox (ART) and dozens more), but it is scattered across pip ecosystems, research repositories, and vendor platforms. A practitioner assembles a working environment from scratch, repeatedly and inconsistently; the environment assembled by a red teamer and the environment assembled by the ML team being tested share almost no components.

Kailash OS is our attempt to close that gap, and to do the same thing for AI security that Kali did for penetration testing and Anaconda did for data science: one declaratively built OS where builders learn to break, and breakers learn to build.

1.2 The design thesis in one sentence

Every tool that attacks AI, every tool that defends AI, every tool that builds AI, and the deployment substrate all of it runs on – expressed exactly once, in Nix flakes, in one reproducible environment shipped as an operating system.

The "exactly once" is the interesting part. Our architecture maintains a single tool manifest from which the system views are generated mechanically (the package overlay, the NixOS option tree, the desktop menu, a command-line interface, the documentation, and a standards-verification evidence matrix). The OS itself is expressed as a NixOS module tree in which each category is an opt-in option, and the distribution artifacts are image builds from that same module tree. Nothing version-drifts, because nothing is installed anywhere: everything is derived.

1.3 Contributions

This paper makes the following contributions:

1. A taxonomy for AI-security tooling organised in four named layers (CLASSIC, ATTACK, DEFENCE, BUILD) over a CORE substrate, with 31 categories and approximately 340 primary tool slots, mapped to the MITRE ATLAS

adversarial-knowledge base and the OWASP verification standards (ASVS 5.0, AISVS 1.0, MLSVS) and risk lists (the LLM Top 10, 2026 edition) rather than invented ad hoc (Section 4).

2. A distribution architecture in which a single manifest drives the six system views, expressed in a two-repository flake-based layout with declarative NixOS modules per category (Section 5); the sixth view, the standards-verification evidence matrix, is what makes "verified against ASVS 5.0 / AISVS 1.0 / MLSVS" an executable rather than a spreadsheet.
3. An argument and a mechanism for reproducible OS release engineering: release trains keyed to a lockfile bump, image builds as derivations, size budgets, golden-image diffs, and bit-for-bit two-machine reproducibility – with an honest treatment of where the complexity actually lands (Section 6).
4. A pedagogy-first position: the co-installation of build platform and attack tooling as a deliberate design decision that converts an OS into a lab, so that builders can red-team their own deployments without leaving their build environment (Section 7), with the verification-standard toolkit (ASVS/AISVS/MLSVS) making "verified against the standard" an executable command rather than a spreadsheet exercise (Section 4, Table 1).
5. An agent-operability design: machine-readable output on every catalog, provision, and assurance command; per-category container provisioning; the catalog exposed over MCP; and agent-scoped capability grants that keep active-exploit tooling out of reach of unsupervised autonomous operators (Section 7.4).

All offensive tooling ships under an explicitly stated responsible-use constraint (Section 8): authorised security work only.

1.4 Paper structure

Section 2 grounds the design in the related work and the tooling landscape it packages. Section 3 presents the reasoning for Nix and the Kali shape. Section 4 the taxonomy. Section 5 the architecture and packaging strategy, Section 6 the release engineering, Section 7 the pedagogy and agent operability, Section 8 safety and governance, Section 9 the limitations, and Section 10 concludes.

2. Background and Related Work

2.1 The Kali model, distilled

Four pillars of Kali's model are the design targets Kailash keeps:

1. Curated pre-installed tooling – not a package-manager list but a curated set, with per-tool usage examples shipped as documentation.
2. Category navigation – the desktop-menu folders organised by task, plus a second, adversary-doctrine navigation (on the Kali tools site, categories mirror the MITRE ATT&CK phases such as Reconnaissance, Initial Access, Lateral Movement), and – Kailash's addition – a third, standards-verification navigation (Section 2.5).
3. Live ISO as a first-class artifact – boot a full environment from removable media, or in a virtual machine, and use it.
4. Menu, CLI, and docs agreeing – three surfaces presenting the same underlying category data.

Kali's model also has a structural cost its users live with: it rolls, so builds are not reproducible across time; its tooling lives in a mutable filesystem, so environments cannot be reconstituted declaratively; and its taxonomy predates the AI attack surface entirely. Kailash keeps the pillars and replaces the substrate.

2.2 Prior Nix attempts at security distributions

Three prior OSS projects attempt the Kali-with-Nix synthesis; their shapes define our starting point. KaliNix (balsoft, 2021) is a single flake with a flat package attribute set and a reference configuration — a proof of concept without packaging harness, categories, menu, tests, images, or documentation, dormant since 2021. Nix Security Box (fabaff, 2020–2026; 405 stars at the time of writing) is a flat directory of category files (port-scanners.nix, wireless.nix, web.nix, and so on) each setting systemPackages, imported as NixOS modules; its category-file ergonomics are right, and we copy that shape directly into our module tree, but it stops at "install some packages" — no bespoke derivations, no testing, no menu integration, no images, no documentation, and no ad-hoc consumption mode for users who don't run the module. RedNix (redcode-labs, 2021) packages as flake outputs and ships ready-to-use nixos-container shells — the container-first lab posture we adopt — but likewise has no category system, manifest, or release engineering.

Three limitations recur across all three: bespoke packaging for off-nixpkgs tools is absent; verification is absent; and the "distro" surfaces (menu, docs, images, release process) are absent. Kailash's contribution is precisely the harness that turns the category-file idiom into a distribution.

Beyond the security-specific attempts, the machine-learning side of the Nix ecosystem offers maintained patterns Kailash adopts deliberately. nixified.ai packages large AI applications as self-contained flake outputs with per-vendor GPU variants, and its variant-naming pattern (a non-GPU default plus explicit -nvidia outputs, with the binary-cache lines carried in the flake's nixConfig) is adopted verbatim for Kailash's CUDA posture (Section 5.6). nixvital/ml-pkgs maintains composable overlays for the torch and jax families and the data-science stack; Kailash consumes it as the documented fallback when nixpkgs' own variants fight back at packaging time — preference order: nixpkgs-native, then the pinned nixvital overlay, then a bespoke derivation, with the manifest recording which path applies. Zero to Nix (Determinate Systems) is the docs-site onboarding track's reference for the flakes-first learning path. The 2023 Kailash organisation repositories (kailash-references' submodules of RedNix, nix-security-box, and kalinix) remain frozen as historical reference; the new reference set lives in the implementation plan's §2.3 and the website's related-work page.

2.3 The AI-security landscape the taxonomy must cover

The offensive tooling is now substantial: garak (NVIDIA) probes for known LLM failure modes with a large probe library; PyRIT (Microsoft) orchestrates multi-turn adversarial campaigns; promptfoo (MIT) generates application-specific attacks and integrates with CI; ART (IBM/Trusted-AI) provides reference implementations of evasion, poisoning, extraction, and inference attacks against classical ML models. On the defensive side: guardrail frameworks (NeMo Guardrails, LLMLingua-style input guards, LLM Guard's production-grade scanners), evaluation harnesses (lm-evaluation-harness, promptfoo evals, robustbench's curated adversarial benchmarks), and provenance tooling (sigstore's model-transparency provides model signing; CycloneDX provides AI bills of materials). Underneath, the platform layer: experiment tracking (MLflow), model serving engines (vLLM, TorchServe, Triton, KServe), vector stores (Qdrant, Chroma, Weaviate, Milvus, pgvector), container orchestration (k3s, Podman), and local inference (Ollama, llama.cpp). The taxonomy's four-layer split exists because these four populations barely intersect: the tools that attack models, the tools that defend models, the tools that build models, and the substrate they run on.

The threat-model documents the taxonomy maps to are themselves first-class references: OWASP's Top 10 for LLM Applications (v2, 2025) enumerates the ten canonical LLM application risks — prompt injection (LLM01:2025), sensitive information disclosure, supply chain, data and model poisoning, improper output handling, excessive agency, system prompt leakage, vector and embedding weaknesses, misinformation, and unbounded consumption — and MITRE ATLAS (2026.05) catalogues 16 adversary tactics and 101 techniques with 77 sub-techniques for attacks against AI systems, with case studies including multi-turn agent-tool poisoning and poisoned-artifact publication.

2.4 Related distributions and their constraints

Kali's own packaging is Debian-based and mutable; its release model – rolling snapshot ISOs – cannot offer bit-for-bit reproducibility and has no declarative reconstitution. Dedicated ML distributions do not exist in the same category: Anaconda's value is the Python environment, not an OS. Spectrum OS (the source-based "bootable system from a flake" design) is acknowledged as the inspiration for the container posture Kailash adopts. The gap between "a pentest distro" and "a machine-learning distribution" is what Kailash is designed to fill, in both directions at once.

2.5 The verification standards

Four OWASP references anchor the assurance side of the taxonomy, at four scoping levels. They matter to distribution design because they make verification requirements testable – and therefore bindable to tools.

- ASVS 5.0.0 (May 2025): the Application Security Verification Standard – 17 chapters, roughly 350 testable requirements – the general application-security baseline. AISVS deliberately scopes itself on top of ASVS: authentication, session management, authorisation, and transport controls belong to ASVS, not AISVS. In Kailash this governs the non-AI layer of the runtime category (Section 4.5).
- AISVS 1.0 (June 2026): the AI Security Verification Standard – 12 chapters at three verification levels, AI-specific and testable: 1 Training Data Governance & Bias Management; 2 User Input Validation; 3 Model Lifecycle Management & Change Control; 4 Infrastructure, Configuration & Deployment Security; 5 Access Control & Identity; 6 Supply Chain Security for Models, Frameworks & Data; 7 Model Behavior, Output Control & Safety Assurance; 8 Memory, Embeddings & Vector Database Security; 9 Autonomous Orchestration & Agentic Action Security; 10 MCP Security; 11 Adversarial Robustness & Attack Resistance; 12 Monitoring, Logging & Anomaly Detection. Its chapter structure is close to a category blueprint for AI-security tooling: that an OWASP standard gives Model Context Protocol (MCP) security its own chapter is direct structural validation of the agentic category in Section 4.
- MLSVS and the Machine Learning Security Top 10: the predictive-AI (classical ML) counterparts – poisoning, evasion, extraction, and inference attacks against non-LLM models. Keyed to the adversarial-ML executors in Layer ATTACK.
- The OWASP Top 10 for LLM Applications, 2026 edition (August 4, 2026), which supersedes the v2:2025 list in Kailash's mappings: eight of ten entries repositioned – prompt injection holds first place, excessive agency climbs to third on the strength of agentic deployments, system-prompt leakage is renamed and re-scoped as hidden context exposure, and the edition cross-maps each risk into the separate Agentic AI Top 10 (ASI01-09) and into MITRE ATLAS techniques. The ranking was weighted 75% community vote, 25% classified-incident data. Kailash's manifest carries both editions' tags per tool, so an engagement can report against either the current list or the v2 corpus it was audited under.

The Kailash consequence: these four standards induce a verification-standards category (Layer DEFENCE, Section 4.3) and a runtime-AppSec-for-AI category (Section 4.5) that no classical distribution has, plus a third navigation mode – standard-first – alongside the layer-first menu and the ATLAS tactic matrix.

3. Why Nix, and Why the Kali Shape

3.1 The two requirements in tension

Any distribution for AI security must satisfy two requirements that pull against each other.

Currency with a moving target. The AI attack surface moves fast. New agent frameworks, new inference engines with new CVEs, and new red-team techniques land monthly; a tool that is six months stale is sometimes not stale

but wrong (probing yesterday's API against today's endpoint). The distribution must therefore track a moving upstream continuously.

Reproducibility. Engagement work has different needs from browsing. A red-team report must be able to state its evidentiary basis exactly – which scanner version, which probe set, which model, on which OS build – or the report's findings are not repeatable. Two machines in the same engagement must have identical tool sets, or the chain of evidence breaks in the mundane ways (a probe behaving differently across versions) that eat engagement time.

Traditional distribution design resolves this one of two ways. Debian-style stable releases maximise reproducibility by freezing: the release pins every package, and currency is bought only at the next release, years later. Rolling distributions maximise currency by mutation: the system changes continuously, and nothing is reproducible. Neither serves this domain – the first is too slow, the second leaves no evidentiary trail. NixOS is the third point: a lockfile determines the exact tool set; bumping the lockfile is the only mutable act; and any previous lockfile state is always exactly reconstructible. Currency and reproducibility stop being a trade-off; they become two read modes over the same append-only history.

3.2 Why Nix specifically, and what it costs

Three Nix properties do the load-bearing work, and one of them is the whole reason the distribution approach is viable at all.

Declarative system definition. The OS – its tools, services, menu, users, and hardening – is an expression, not a history of installation commands. A host profile such as redteam-ai is a configuration value, not a snapshot of somebody's machine; every category of tooling is an option that composes with every other by evaluation rather than by installation order. This is what allows the distribution thesis: the OS is expressible precisely because the expression is complete.

Content-addressed derivation. Every tool is a store path derived from its inputs (sources, build steps, dependencies). Two builds with the same inputs are the same object, not two similar objects. This is what makes a Kailash ISO a reproducible artifact and what powers the cache that makes distributing it feasible.

The lockfile as the single mutable act. A flake lock pins every input revision – the nixpkgs tree, every tool repository, every rule set. Rolling the distribution forward (the release act) is a lockfile update plus the rebuild that consumes it; rolling back (an engagement's reproducibility requirement, or a bad release) is re-evaluating a previous lockfile. Appendix material, but the release engineering in Section 6 leans on this entirely.

The costs are real and worth stating in a design document: the language has a learning curve with no industry-wide pool of practitioners to hire from; tool ecosystems that expect mutable state (pip installs, CUDA driver stacks, Docker images with vendored libraries) resist derivation; some heavy artifacts (TeX Live, vendor-pinned inference servers) must ship as opt-in layers rather than base; and build times for unfrozen dependency trees are real, which is why the binary cache in Section 5 is not optional. We return to the CUDA case explicitly in Section 5.6 because it is the canonical failure mode of naive Nix ML setups.

With those costs acknowledged, the decisive comparison is with the alternatives: Debian's apt gives curated tooling but no declarative reconstitution, no atomic rollback, and no per-tool pinning; Docker-based appliance images give reproducibility of the image but not of the environment (volumes, host state, and layered mutable images drift), and no taxonomy or menu integration; Ansible and other configuration managers are imperative at their core, and the "declarativity" is an abstraction over shell scripts with no content-addressed store underneath. Nix is the only substrate where the distribution thesis – the OS as a derivation tree with a manifest as its spine – is a natural expression rather than an emulation.

3.3 Why the Kali shape: the ease of a category-navigated distribution

The design does not copy Kali's software; it copies Kali's interface to the practitioner, because that interface is the reason Kali became the default environment for security work rather than a package list.

Curation removes the toil that suppresses usage. A practitioner who must assemble twenty tools before an engagement engages with the environment once and then drifts from it; a practitioner whose environment boots ready goes straight to work. This is the Anaconda lesson applied to security: distribution quality is a prerequisite for domain engagement, not a nicety.

Categories are the domain's native navigation. Every security practitioner's professional vocabulary is already categorical – reconnaissance, initial access, privilege escalation – because ATT&CK made the adversary's operating doctrine the shared language. A distribution that speaks this vocabulary reduces the practitioner's translation work to zero; the tool is where the doctrine says it should be. Kailash extends this to the AI threat surface, whose doctrine is ATLAS: the taxonomy in Section 4 is derived from that doctrine rather than invented.

Live ISO is the field posture. Security work happens on unfamiliar hardware, in disposable VMs, on evidence machines; a distribution that lives only as an installed system loses the field. The live ISO is also, less obviously, the teaching posture: a student boots the environment, follows the documentation for the category they are studying, and is running real tooling in minutes – the lab and the lesson are the same boot.

The agreement of menu, CLI, and docs is the integration test. A distribution where the menu says a tool is in one place, the CLI says another, and the documentation says a third is three distributions. Kali's surfaces agree because they are fed from the same category data; Kailash makes that agreement structural (one manifest, six generated views) and machine-checked in CI rather than procedural.

None of this depends on Kali's Debian internals; it depends on the distribution contract Kali established: come curated, speak the domain's categories, boot anywhere, and agree with yourself across all your surfaces. Kailash's claim is that this contract, expressed on NixOS, produces a better security distribution than Kali's own substrate can – and the AI-security category Kali has never mapped is the wedge.

Verification against a standard is the fourth navigation. Beyond the menu and the doctrine matrix, assurance work needs a third way in: for each requirement chapter of ASVS 5.0, AISVS 1.0, and MLSVS – and each risk of the LLM Top 10 – the tools and data packs that satisfy or test it. Section 4.3 develops the mapping; the short form is that an OWASP chapter reference (say AISVS Ch10, MCP Security) resolves to the specific toolset that verifies it, which turns "verified against the standard" from a spreadsheet exercise into an executable command.

3.4 Declaratively choosing a desktop environment

The Kali-style choice of desktop is itself a configuration option rather than an image fork. modules/base/desktop-kde.nix, -gnome.nix, -xfce.nix each set the session, the window manager, and the menu integration; a profile selects one attribute, and the same base system exists under all three desktops. There are no per-desktop image branches to maintain and no divergence between a KDE ISO and an Xfce ISO: they are the same evaluation with one attribute different, and the light ISO (Xfce) and the default (KDE) stay in lockstep by construction.

4. The Taxonomy

4.1 CORE + four tool layers, 31 categories

The taxonomy has two levels. The layer answers the professional question; the category is the tool grouping inside it. Following the scope consolidation of Revision 1.1, Layer CLASSIC is software-only: the wireless/RF, Bluetooth, and hardware-hacking categories of the classic-Kali taxonomy are deliberately excluded (Section 4.5),

and the sole radio-adjacent concession is a signal-processing pack (C-7) kept opt-in because machine-learning signal work – audio, voice, and waveform corpora – is legitimately part of the taxonomy's research workload.

- **CORE** – the substrate: kernel, base userland, hardening, safety defaults, the desktop(s), the CLI and menu engines; hosts no tool categories; every image inherits it.
- Layer CLASSIC, software-only (10 categories, ~141 tool slots). Information Gathering; Vulnerability Analysis; Web Application Assessment; Database Assessment; Password & Offline-Credential Attacks; Reverse Engineering (software artefacts); Signal & SDR Processing (opt-in pack); Exploitation & Post-Exploitation (software targets); Forensics, IR & Reporting; and C-10, Runtime AppSec for AI Systems (Section 4.5). Deliberately curated rather than exhaustive – the layer exists so that an AI engagement's classical prerequisites (recon, credential work, forensics) do not require leaving the environment.
- Layer ATTACK – offensive-AI (8 categories, ~59 tools). AI Reconnaissance & Target Discovery; LLM Prompt Injection & Jailbreaking; Agentic, MCP & Tool-Use Attacks; Model Supply Chain & Artifact Poisoning; Model Extraction, Inversion & Membership Inference; Multi-modal & Physical-AI Attacks; AI System Exploitation; Data Poisoning & Dataset Attacks – mapped to ATLAS tactics and OWASP LLM risk classes (Table 1). Three categories deserve note: A-3 (Agentic, MCP & Tool-Use Attacks, 10 tools, mostly bespoke) is mapped to the ATLAS agentic technique set (2026.07), the OWASP Agentic AI Top 10 (ASI01-09), and AISVS Ch10 – the category no classical distribution has a home for; A-4 (Model Supply Chain & Artifact Poisoning) carries the guarddog/picklescan/modelscan verification sweep; A-7 (AI System Exploitation) ships the attack probes against the very platforms Layer OPS deploys, the co-installation that Section 7 builds the lab thesis on. A-6 (Multi-modal & Physical-AI Attacks) keeps the audio-voice-clone-injection surface – attacks on voice pipelines and audio-model inputs are semantic attacks on AI systems, not radio tooling – but ships no RF transmission capability.
- Layer DEFENCE – defensive-AI & assurance (7 categories, ~46 tools). AI Runtime Defence & Guardrails; AI Detection, Monitoring, Observability & IR; AI Assurance, Evals & Benchmarks; Model Provenance, Signing & Trust; AI Threat Modelling; AI Governance & Regulation Mapping; and D-7, the Verification Standards Toolkit (Section 4.3): the ASVS 5.0 / AISVS 1.0 / MLSVS / Top-10 corpora as data derivations, the requirement-to-tool mapping of Table 2, and the verifier/compliance-export commands that emit an evidence matrix for a chosen standard and level – the difference between "I ran some scanners" and "AISVS Level 2, Ch10 verified, with evidence".
- Layer OPS – platform (7 categories, ~91 tools). MLOps & ML Engineering Suite; Compute & Serving Infrastructure; Vector Stores & Retrieval; Local LLM & Inference Labs; AI Engineering SDKs & Dev Environment; Documentation, Reporting & Authoring; and O-7, Agent & MCP Operability (Section 7.4) – the catalog exposed over MCP, the third-party agent rigs, and the headless boot-to-agent posture.

A tool occupies exactly one primary category; spanning tools (Burp Suite as classic web proxy and offensive-LLM traffic interceptor) carry secondary tags that the CLI's search honours while the menu slot stays single.

4.2 The mapping to ATLAS and OWASP

Layer ATTACK categories are not invented; they are induced from the two reference doctrines (Table 1). The mapping is what makes the tactic-first navigation mode work: the documentation site renders, for each ATLAS tactic, the Kailash tools that map to it – so "I want to attempt AML.T0051 (prompt injection), which tools?" resolves in one page. The mapping is maintained in the manifest as fields, and the category table is the documentation form of the same data.

Table 1: Layer ATTACK categories mapped to ATLAS tactics and the OWASP LLM Top 10

Category	ATLAS tactics / techniques	OWASP LLM:2026
A-1 AI Recon & Target Discovery	AML.TA0002 (Recon), AML.TA0008 (Discovery)	LLM02 (hidden context exposure class)
A-2 LLM Prompt Injection & Jailbreaking	AML.T0051, AML.T0054	LLM01
A-3 Agentic, MCP & Tool-Use Attacks	agentic technique set (2026.07)	LLM03 (excessive agency), LLM01
A-4 Model Supply Chain & Artifact Poisoning	AML.T0010	LLM04 (supply chain)
A-5 Model Extraction, Inversion & MIA	AML.T0024, AML.TA0000	LLM02
A-6 Multi-modal & Physical-AI Attacks	physical/multi-modal techniques	–
A-7 AI System Exploitation	AML.T0010 (platform class)	LLM03
A-8 Data Poisoning & Dataset Attacks	AML.T0020	LLM05 (Poisoning)

(v2:2025 tags are carried in parallel in the manifest; the 2026 edition's excessive-agency promotion to LLM03 is why A-3 and A-7 both map there.)

4.3 The verification-standard mapping

The same mapping discipline applied to the standards stack: each requirement chapter or risk resolves to the Kailash tools and data packs that verify it. This table is the generated core of the verification-standards category (D-7) and of the standard-first navigation mode; the kailash verifier command emits it as an evidence matrix.

Table 2: Requirement chapters to Kailash tool bindings

Standard reference	Content	Kailash binding
AISVS Ch9 – Agentic Action Security	orchestration, agency limits	A-3 adversarial tests (agent-dojo, mcpoison, llm-agent-fuzzer); O-5 build-side framework posture
AISVS Ch10 – MCP Security	MCP servers, tools, OAuth fronts	A-3 scanner suite (mcp-scan, mcp-guardian) + D-4 inspector (mcp-inspector)
AISVS Ch6 – Supply Chain (Models/ Frameworks/Data)	model/framework/ data chain	A-4 verification sweep (guarddog, picklescan, modelscan); AIBOM via cdxgen as evidence
AISVS Ch8 – Memory, Embeddings & Vector Security	vector store controls	A-8 adversarial probes; O-3 deployment posture as build-side controls
AISVS Ch11 – Adversarial Robustness	attack resistance	A-5 sweeps (ART, foolbox, counterfit); robustbench metrics as evidence
AISVS Ch12 – Monitoring & Anomaly Detection	logging, detection	D-2 toolchain (langfuse, phoenix, evidently, whylogs) + detection-validation runs
AISVS Ch1 – Training Data Governance	data lineage, bias	D-3 fairness tooling (fairlearn, aif360) + A-8 poisoning probes for control effectiveness
AISVS Ch2/5/7 – Input Validation; Access Control; Output Control	runtime AI endpoint	C-10 runtime AppSec toolkit (mitmproxy, AI-payload fuzzers, oauth-mcp-prober)
MLSVS / ML Top 10 M01-M10	classical ML surface	A-5/A-8 executors (ART, robustbench, trojai) keyed to predictive-AI testing
ASVS 5.0 Ch1-17	general application baseline	C-10 toolkit for the non-AI perimeter (the layer AISVS scopes out)

Standard reference	Content	Kailash binding
LLM Top 10:2026 LLM01-LLM10	the ten application risks	per-risk tool tags in the manifest; emitted in the verifier evidence matrix

The mapping is data in the manifest (per-tool asvs / aisvs / mlsvs / owasp_llm fields), generated into the documentation and CLI from the same single truth as everything else.

4.4 Category design decisions

Three deviations from a naive Kali-clone are deliberate and worth logging:

1. C-8 consolidates the software exploitation surface (exploitation, webshells, privesc/post-exploitation for software targets) into one category; the network-infrastructure arsenal Kali carries (Active Directory and lateral-movement tooling, sniffer/spoofer arsenals, C2 frameworks) is deliberately not consolidated but excluded – Kailash's four pillars are application, API, model, and AI security (Section 4.5).
2. The wordlists (seclists and the large packs) and the signal-processing pack (C-7) are data derivations in opt-in layers (kailash-data.*) rather than content of the base categories, because the ISO budget in Section 6 is the binding constraint.
3. Layer OPS is capped at seven categories. The platform layer's natural temptation is "add everything platform teams use" (Spark, Vault, Terraform…); those land in the dev-environment category if needed, and new D categories require a taxonomy decision, not a PR.

4.5 What is deliberately shipped, and what isn't

No commercial-licensed tool (Nessus, Cobalt Strike, IDA Pro) ships; the documentation provides licence guidance. Unmaintained tools Kali keeps for familiarity (truecrack, paros) do not ship. Cloud-only SaaS does not ship (exceptions: CLI-only deliverables such as wandb). Where two tools are functionally equivalent and one is clearly maintained (netexec versus crackmapexec), the maintained tool ships and the supersession is documented.

Two scope exclusions define Kailash's boundary and are stated as doctrine rather than omission. First, the wireless/radio/RF/Bluetooth/hardware-hacking toolset (Kali Nethunter's territory – aircrack-family, wifite, kismet, reaver, hcxdumptool, the ubertooth stack, rfcat, spooftooph, chirp, and friends) does not ship: Kailash's four pillars are application security, API security, model security, and AI-system security, and none of them is a radio. The one concession is C-7, a software-only signal-processing pack (GNU Radio, GQRX, hackrf host tools, sox/librosa DSP libraries) kept opt-in – it exists because machine-learning signal research (audio and waveform corpora feeding ML training, and the audio-voice-clone attack research in A-6) is legitimately inside the taxonomy's workload. Second, the network-infrastructure exploitation arsenal (impacket suite, mimikatz/rubeus/BloodHound, responder/netexec, kerberoast tooling) and online credential-attack tooling do not ship either. Revision 1.1 of the implementation plan carries the full purge ledger – every removed tool slot, with the rule that removed it.

5. Architecture

5.1 Two repositories, one website

The distribution is two repositories and a website, whose split mirrors nixpkgs/NixOS's own relationship and the org's existing names:

- kailash-os – the OS repository: the root flake, the module tree (the categories), the base OS substrate, profiles, image generation, the CLI, the documentation engine, tests, and CI. It imports the packages repository below.
- kailash-packages – the overlay repository: every bespoke tool derivation (one directory per tool), the tool manifest, the update automation, and its own CI with its own binary cache. It is independently versioned and can be consumed standalone by users who want the tooling on an existing NixOS host without the OS itself.
- kailash-website – the public site: a Quarto-based documentation and governance site (getting-started, generated layers/tools/tactics/standards pages, the governance pages) whose content is generated from the manifest of a pinned packages-revision at site build time. It deploys independently via GitHub Pages and shares the repository layout's link-check and stub-check gates. The two repositories' separation means the website's truth chain is: packages-repo manifest (pinned rev) → site build → Pages deploy; content drift from the installed system is structurally impossible in the same way the menu and CLI cannot drift (Section 5.3).

The manifest lives in the packages repository – derivations and manifest belong together – and the OS repository pins it by flake input.

5.2 The module tree as the taxonomy

Each of the 31 categories is a NixOS module file with an identical contract (uniformity is what makes generation mechanical):

```
{ config, lib, pkgs, ... }:
{
  options.kailash.layers.classic.categories.c-01-information-gathering = {
    enable = lib.mkEnableOption "Kailash category: information gathering";
    packages = lib.mkOption {
      type = lib.types.listOf lib.types.package;
      default = with pkgs; [ nmap masscan rustscan /* generated from the manifest */ ];
      description = "The information-gathering tool set. Extend, don't replace.";
    };
  };
  config = lib.mkIf config.kailash.layers.classic.categories.c-01-information-gathering.enable {
    environment.systemPackages =
      config.kailash.layers.classic.categories.c-01-information-gathering.packages;
  };
}
```

This contract has three properties that matter: opt-in by default (mkEnableOption defaults to false – a minimal host enables nothing; a full profile enables all 31), extendable (a downstream user appends to the category's package list rather than forking the module), and packaging-free (the category names tools; the derivations live in the overlay).

5.3 The manifest: one truth, six views

The single hand-maintained source is manifest/tools.yaml – one entry per tool, with layer, category, source, licence, packaging status, safety level, and (for Layer ATTACK) ATLAS/OWASP/standard tags. Six system views are generated from it:

1. The overlay – derivations are packaged iff manifest-listed;
2. The category modules – each category's package list is generated (no hand duplication);
3. The desktop menu – .directory / .desktop entries;
4. The CLI (kailash list/search/info/doctor/report) – and a "doctor" per tool;
5. The documentation – 31 category pages, per-tool pages, the ATLAS-tactic matrix, and the standard-first verification pages;
6. The verifier evidence matrix – kailash verifier –standard aisvs –level 2 emits the chapter-by-chapter coverage table (Table 2's data) with the engagement's run evidence attached.

Drift between views is structurally impossible if generation is gated: CI regenerates each view in check mode, and any diff fails the build. The manifest is also baked into the CLI's derivation hash – the binary's content changes when the manifest does – so a kailash list can never disagree with the installed system.

The manifest entry is also the safety record (level: safe / requires-target / active-exploit, plus licence and source), which Section 8's governance leans on.

5.4 Packaging the three populations

Every tool lands in one of three packaging statuses, recorded per entry in the manifest:

1. nixpkgs-native – pin via the lockfile; no derivation work (the majority of the CLASSIC and BUILD layers);
2. stale-override – in nixpkgs but stale; re-pinned from upstream via nvfetcher (fast-moving tools such as nuclei, katana, subfinder);
3. bespoke – not packaged: one derivation per tool in the overlay, built on three shapes (pure-Python pip packages; Rust/Go builds; docker-backed wrappers for the vendor-pinned inference servers). The docker-backed shape is used where upstream ships only container images (Triton, Milvus); the pattern is an honest wrapper package plus a documented Podman service module rather than a pretence of native derivation.

A fourth pattern covers the GPU variants: for tools where acceleration matters (the torch family, vLLM, onnxruntime), Kailash follows nixified.ai's variant-suffix practice – a non-GPU default derivation plus explicit -nvidia variant outputs, built only by labelled CI jobs, never pulled by default. This keeps every default profile free of CUDA dependencies while keeping the GPU path first-class.

The bespoke set is roughly 70 derivations post-consolidation, sequenced by value-per-effort: the flagship offensive five (garak, PyRIT, promptfoo, ART, counterfit) first, as proof of the harness.

5.5 Platform deployment and the attack pairing

Layer OPS ships the defensible lab deployments as service modules (MLflow without auth documented as the lab default; Langfuse; Phoenix; k3s; vLLM with the CUDA dependency explicit), and Layer A-7 ships the attack probes against those same platforms (mlflow-bypass, shadow-ray, vector-db-ripper). The pairing is a deliberate design decision rather than a coincidence of curation, and its pedagogy consequences are developed in Section 7. Its security consequence is immediate: the distribution is its own verification harness – a platform team can stand up the same lab deployment and run the A-7 probes to validate their own hardening.

5.6 The CUDA case, stated precisely

The canonical failure of naive Nix ML setups: CUDA-enabled PyTorch is unfree and not built on the public cache, so source builds of the dependency stack produce the familiar linker failures ("Torch not compiled with CUDA enabled"; cudnn/cublas resolution failures). Kailash's posture has two mechanisms: the variant pattern above (non-GPU defaults, explicit -nvidia outputs with exact pins – the nixified.ai pattern), and a fallback import of nixvital's maintained torch-family overlay when nixpkgs' own variants are inadequate. cudaSupport = false remains the CI-default path; the GPU profile is opt-in with its exact working pins documented in the lock. The GPU path is a first-class target – it is the lab posture – but it is the opt-in, exactly-pinned one.

6. Release Engineering: Is Declarative Harder to Release?

The question deserves a section of its own because it is the first objection practitioners raise: if the whole OS is code, does shipping a new version become a build-engineering exercise? The answer, we argue, is the opposite – but only if the release model is designed around what the substrate actually makes cheap.

6.1 What a "release" is, concretely

A Kailash release is four mechanical steps, no manual assembly:

1. Lockfile bump. Renovate keeps nixpkgs and the flake inputs current on a monthly cadence (first Monday); nvfetcher tracks the bespoke tool repositories on per-tool staleness classes – fast-moving tools checked weekly, steady tools fortnightly, the rest riding the monthly nixpkgs refresh. Both open reviewed pull requests; merging them is the release train.
2. Evaluation and test. nix flake check is the single gate: it evaluates every profile, every derivation, and the 31 category VM smoke tests plus the integration tests (full-boot, mlflow-attack pairing, local-garak nightly variant). Nothing enters a release that has not booted.
3. Image builds. The nine image targets (install ISOs in standard, lite, and full variants; the headless-agent image; red-team and blue-team OVAs; cloud AMI; Proxmox template; Raspberry Pi image) are flake outputs built by CI – derivations, not hand-crafted artifacts.
4. Tag and publish. The lockfile-bump merge is tagged with the version the lock implies – the calendar release-index scheme (YYYY.NN: the year's first train is 2026.01; staged cuts carry the ladder suffix, 2026.01-alpha.1, with the bare tag reserved for the train's GA and a material re-cut posting 2026.01.1) – and the ISO, OVA, AMI, docs (the site, published on Pages), and a machine-generated changelog are attached; all signed.

Cadence, stated as policy because "how often does this update" is the question every declarative project must answer: the system does not update itself – ever. The lockfile is the product; a release is a reviewed, tested lockfile state. nixpkgs inputs refresh monthly; bespoke tools by staleness class; tagged releases ride the six-week train (aligned to two monthly refresh cycles, Kali-like in rhythm, deliberately not bleeding-edge); hot releases (a critical CVE in a shipped tool) reuse the same path with a single-tool nvfetcher PR – no special process. A user's install updates only when they pull a newer lock and rebuild; between releases nothing re-links, re-compiles, or re-downloads.

6.2 Why this is easier than the mutable-packaging loop, not harder

Compare the two release models honestly.

The mutable-packaging release loop (Kali's implicit model) has the release engineer assembling: a package-repo sync, ISO tooling runs, per-image QA on whatever state the mirrors happened to be in at build time, and changelog compilation by a human reading commit logs. Every step is an artifact of a mutable filesystem, so no step is exactly repeatable – including the builds that fail, which cannot be re-run against the same world state.

The classic Kali release failure ("the mirrors moved mid-build; rebuilt the ISO; tools differ between the two ISOs from the same day") is not an accident but the expected behaviour of an unpinned system.

The declarative release loop moves assembly to evaluation time before the human looks at anything: the lockfile bump produces a fully specified world; the test gate boots it; the image builds are deterministic derivations from that specification. The engineer's job is reduced to reviewing diffs (what moved, what's new, size deltas) and approving the tag. The failure modes move earlier: a build break lands in the nvfetcher or Renovate PR, before release day, as a red check on an open pull request – not as a broken ISO after tagging.

Three things declarativity removes entirely from the release path:

- "Works on my machine" release divergence – two release engineers produce bit-identical artifacts from the same lock;
- Silent dep resolution surprises – the lock is the resolved tree; surprises become reviewable diffs;
- Roll-forward-only releases – the immediately previous release's lockfile stays reconstructible, so a bad release is a revert of one file, and engagement reproducibility ("which build produced this evidence?") is a lockfile reference.

And two things it adds, honestly stated: release-day discipline must prevent unreviewed lock drift from sneaking into the train (the auto-merge gates cover this), and image build times on the cache-cold path are real – which is why every release's cache is pre-warmed by the per-PR builds that preceded it, and why the size-budget gate (below) exists to keep the image set from swelling.

6.3 The mechanisms that make the loop trustworthy

- Size budgets, CI-enforced. Each image target declares a compressed-size budget; the build fails on overrun. This is the mechanism that forces the heavy artifacts (TeX Live, the signal-processing pack, the wordlist packs) into opt-in kailash-data.* layers rather than letting "kali-everything" creep double the ISO size each train.
- Golden-image diff. Each release diffs its full ISO against the previous: version bumps and tool additions expected, silent removals or behavioural shifts flagged for review before the tag. The taxonomy purge of Revision 1.1 is the pattern applied prospectively: every removed tool slot is documented with the rule that removed it.
- Two-machine reproducibility as the acceptance test. The v1.0 acceptance criterion is literal: two machines (bare metal and VM) built from the same lock must agree with kailash manifest validate and nix flake show bit-for-bit. This is the property Kali cannot offer, stated as a test.
- Boot-level honesty. The test gate runs the OS in a VM (NixOS's test driver), so "released" has always meant "booted with the category's tools exec-checkable and the pairing probes running against their targets".

6.4 The cost ledger of declarative release

Where the complexity of the declarative approach actually lands, precisely because it is cheaper to be explicit about the ledger than to argue rhetorically. The learning curve of the Nix language (and the thin contributor pool) is the largest real cost and the reason the implementation plan front-loads a bootstrap phase. The build-time cold path is the second. Both are amortised: the language cost once, the build cost by the cache. What is not on the ledger is per-release engineering, because evaluation consumed it. The net claim of this section: declarative construction makes individual releases harder to hand-assemble and impossible to get silently wrong – which is the easier property overall.

7. The Lab Thesis: Building and Breaking Together

The distribution's most deliberate decision is where the platform and the attack tooling sit relative to each other: co-installed, co-pinned, in one environment.

7.1 Why builders learn to break

The practitioner populations in AI security split today between the builders – ML engineers, platform engineers, application developers, who have the platform (registries, serving engines, vector stores) and little security tooling – and the breakers – red teamers and researchers, who have the offensive tooling and typically none of the platform.

The consequence of the split is not merely inconvenience but competence asymmetry: builders who have never attempted a prompt-injection campaign against a RAG pipeline systematically under-provision against it (defence against a technique you have never run is guesswork); breakers who have never deployed MLflow cannot reason about the exploitation surfaces the deployment engineer sees. Both professions are weaker for the split.

7.2 The mechanism: the pairing

Kailash co-installs the platform and the probes against it. The D-layer ships, for example, MLflow as a lab deployment whose unauthenticated default is a documented teaching posture; the A-7 layer ships mlflow-bypass; the docs cross-reference the two. The result:

For a builder: enable the O-1 module and the A-7 category, boot, and attempt the platform exploitation your production hardening claims to prevent – using the same pinned toolset a professional red team would use – then read the hardening appendix and try again. The feedback loop is minutes, in your own environment, not an external engagement's report weeks from now.

For a breaker: the same environment ships the target platform itself: inference engines, vector stores, registries, agent runtimes, at pinned, known-configured lab states. Attacks are developed against an accurate local model of the production surface rather than against internet-exposed systems (with the legal exposure that carries).

For a teacher: the ISO plus docs plus pairing is a complete curriculum artifact: a category page teaches the doctrine (ATLAS tactics, OWASP risks); the pinned packages execute it; the integration test at the repo level is the answer key.

7.3 What the lab thesis costs

Honest ledger again: co-installing attack tooling with build platforms raises the dual-use surface of the whole distribution (the safety module in Section 8 is the answer, and it is the reason the gating posture exists); the lab deployments are teaching-accurate, not production-recommended (the docs carry the posture difference explicitly, per module); and the maintenance burden of the pairing is real – each A-7 probe references the D-layer configuration it was validated against, and a platform version bump can invalidate a probe, which the manifest's validation metadata tracks rather than hides.

The alternative – separating the "build OS" and the "hack OS" into two distributions – is the status quo rebadged: the same disjointness with a shared website. The thesis is precisely that separating them is what perpetuates the competence asymmetry.

7.4 Agent operability: the OS as something an agent can spin up, build, and use

A security distribution designed in 2026 has a second user with no login shell and no desktop session at all: the autonomous operator. Coding agents today already provision environments, run toolchains, and execute multi-step workflows on behalf of a principal; a distribution whose catalog, environments, and services are only reachable through human affordances (menu clicks, interactive installers, prose documentation) excludes them by construction. Agent operability is therefore designed in, not extended on – and it is not a marketing claim but five concrete commitments:

Machine-readable catalog and provision commands. Every kailash command carries –json (and the list/search/info surface is stable against a schema in the manifest). An agent resolves "does this environment have nuclei, and at what pin?" by one command, not by parsing prose; it provisions by kailash container (a NixOS container or Podman container with exactly the categories enabled – the nixos-container posture RedNix proves) rather than by mutating a shared system. Provisioning is idempotent and content-addressed, so an agent can converge a target environment declaratively and verify the result with kailash manifest validate – the same property a human operator relies on, expressed for non-interactive use.

The catalog over MCP. Since the OS ships MCP tooling and the ecosystem is moving to MCP as the de facto agent-tool bus, the catalog itself is exposed as a small MCP server (kailash-mcp): tools for list_categories, list_tools(category), tool_info(id), doctor(id), shell(category), container(category), and verifier(standard, level) – so an MCP-native agent discovers and drives the OS with zero bespoke integration. The MCP server is itself a pinned derivation built from the same manifest; its tool list is derived, not hand-written, so it cannot drift from the catalog either. (The governance mirror is deliberate: the OS's own tooling includes the scanners that test MCP servers for the very flaws – unsafe deserialisation, tool poisoning – this server must be built to resist.)

Third-party MCP rigs as curated deliverables. The popular build-and-hack MCP servers are packaged and pinned alongside the first-party one, because they are precisely what an agent-operator rig mounts today: hexstrike-ai (0x4m4) exposes 150+ security tools – network, web, cloud, binary, CTF arsenals – behind MCP for autonomous offensive workflows, and MCP-Kali-Server (TriV3) bridges Kali-style tooling to MCP clients such as VS Code and Claude Desktop. Both carry manifest safety exp (they are agent-driven offensive tooling), both are therefore grant-gated like any exp tool, and both pair in the docs with the D-layer deployment configs they were validated against. The community malicious-MCP proof-of-concepts (Malicious-MCP's disguise-as-a-server proxy; the deliberately insecure demo servers) are packaged into Layer A-3 as verification lab material – the targets the A-3 scanner suite is run against in CI to prove it detects real tool-poisoning patterns rather than passing on toy fixtures.

Boot-to-agent headless images. Alongside the desktop ISOs, a headless image target boots to SSH with the CLI and the manifest present and no desktop layer. It is the CI VM posture packaged for reuse: an agent (or a CI runner acting as one) boots the image, provisions categories, and works over SSH. This is what "an agent can spin it up to build and use" concretely means: one image, one SSH key, one kailash container invocation.

Agent-scoped capability grants. The safety module (Section 8) extends to autonomous operators: a capability grant is a short-lived, human-minted token that scopes (a) which categories the session may enable, (b) whether services may be started, and (c) whether active-exploit tooling is executable, plus a query budget against live targets. An unsupervised agent holds no grants: it can list, search, and read documentation, but it cannot enable categories, start services, or execute anything whose safety level is active-exploit. Every grant-scoped action is logged. This is the OS subjecting itself to the same excessive-agency discipline it packages tools to test for (LLM03:2026) – a distribution that ships agent-attack tooling should not itself model unsupervised agency.

Deterministic builds are the agent-affordance substrate. The agent-affordance case is not separate from the release-engineering case (Section 6) – it is the same property, pointed at a second user: Renovate/nvfetcher PRs are machine-authored and reviewable by an agent under human approval; flake check gates are machine-

checkable verdicts; the changelog is machine-generated. A principal delegating "roll the October train" to an agent gets reviewable diffs and boot-verifiable images at every step, with the human's act reduced to approvals. Declarativity is what makes every part of that delegation auditable after the fact – the same reason it makes releases trustworthy.

The agent-affordance ledger. The costs: the MCP server and headless image are two more artifacts to test (both land in the plan's CI matrix, so the cost is bounded); stable –json output is a compatibility commitment the CLI takes on from v1; and the capability-grant system is the difference between "the OS can be driven by agents" and "the OS is safe to let be driven by agents" – it is the load-bearing piece and the part most worth external review.

The claim of this section, net: a 2026 security distribution for AI should be operable by the same class of autonomous operators that the profession is beginning to test against, and the substrate that makes it declarative is also what makes that operability auditable.

8. Safety, Governance, and Licensing

Responsible use. Kailash ships offensive AI-security tooling for authorised security work only: red-team engagements with written authorisation, academic research, and defender validation of one's own systems. Users are responsible for the law in their jurisdiction (in Australia: Cyber Security Act 2024 (Cth); Criminal Code Act 1995 (Cth), Div 474; the ISM controls the author's professional framework works within). Every Layer ATTACK documentation page carries the authorised-use notice.

Safety defaults in the OS. Three layers of technical framing: (1) lab mode – tools whose manifest safety level is active-exploit are wrapped to refuse execution unless the host profile and the current shell both explicitly opt in (kailash.labMode = true, KAILASH_LAB_MODE=1); normal Layer CLASSIC tooling runs un-gated, mirroring Kali; (2) no auto-starting services – every D-layer service module defaults off; enabling a lab deployment is a deliberate act, and the lab posture (e.g. unauthenticated MLflow) is documented rather than defaulted; (3) secrets hygiene – keys and model API tokens are sops-nix-managed and read at runtime from the user's configuration directory, never embedded in the manifest, the repo, or the (world-readable) store.

Licensing. The distribution's own artifacts are MIT-licensed. Bundled tools keep upstream licences, recorded per manifest entry (garak Apache-2.0; PyRIT MIT; ART – AGPL-class, verified at packaging time and recorded). Unfree artifacts (Burp Suite) are recorded as unfree and the OS sets the corresponding Nix allowUnfree posture with documentation. Data packs (jailbreak-prompt corpora, payload packs, ATLAS data) carry their upstream licences with use restricted accordingly.

Authorship. Shaineel Singh is the sole author and maintainer of this paper and the Kailash OS project (contact: shain.singh@owasp.org). Funding: none. Conflicts of interest: none declared.

Contribution workflow. PRs to either repository require the CI gates plus maintainer approval; Layer ATTACK tool additions additionally require a safety-level confirmation in the PR template. The manifest is the only hand-maintained tool registry (Section 5.3); hand-additions elsewhere fail CI. Commits are cryptographically signed by the project's convention, and no commit lands directly on a main branch: branch, conventional-commit, pull-request, squash-merge.

Also on the site: governance pages (the responsible-use statement, licensing posture, reporting path) publish at kailash-website – same content as this section, generated from the same source. The site serves at **kailash.site** (GitHub Pages + CNAME); **kailash.tools** is reserved for the tooling plane – the binary-cache front-end and tool-facing services.

9. Limitations and Future Work

Research-grade upstream. The offensive tooling is pip-ecosystem, fast-moving, and often released only as repository heads rather than tagged versions. Kailash pins commit revisions and states pin dates in the docs; the price is that "the same tool" across two releases can drift behaviourally, caught (not fixed) by the golden-image diff.

The ~70 bespoke derivations are the schedule's critical path. The implementation plan sequences them by value-per-effort, but the honest position is that this is the project's largest engineering risk and the reason not every Layer ATTACK entry has a derivation in v1 — the plan explicitly stages "documented starter sets" for the still-unpackaged probes.

CUDA stays hard. The GPU posture is opt-in with exact pins (Section 5.6) rather than a general solution; until the upstream torch-bin ecosystem stabilises further, the CUDA stack is a maintained pin set, not a configuration-free experience.

Taxonomy maintenance. ATLAS's cadence (16 tactics, 101 techniques at 2026.05; the 2026.07 additions include the agentic set) means the Layer ATTACK mapping is a maintained artefact, not a fixed table; the manifest's ATLAS fields are the update surface.

Future work, in the order the plan stages them: ARM64/edge-class support (the Pi target is minimal-profile only today); an Apple-silicon nix-darwin companion module (documented as stretch); the novel agentic probes upstreamed to nixpkgs (the plan's overlay is the staging ground); and an AI-BOM generation flow integrated into the release chain so each OS release ships its own bill of materials by default.

10. Conclusion

Kailash OS is an argument about distribution design: that the AI-security profession needs the distribution contract Kali established — curated tooling, category navigation, live media, self-consistent surfaces — pointed at a surface Kali has never mapped, on a substrate where the distribution itself is a reproducible derivation. The design makes four commitments. First, the taxonomy is derived from the domain's doctrines (ATLAS, the OWASP LLM Top 10) rather than invented, and organised as CORE plus four named tool layers — CLASSIC, ATTACK, DEFENCE, BUILD — because the tool populations barely intersect. Second, a single manifest drives all six system views, making drift between menu, CLI, docs, modules, and overlay structurally impossible rather than procedurally resisted. Third, the platform and the attack tooling ship together, making the OS a lab rather than a package list — because builders who have not attempted the attacks systematically under-provision against them, and breakers who have not run the platforms systematically misread them. Fourth, the scope boundary is doctrine: the wireless/RF/hardware toolset and the network-infrastructure arsenal are excluded as a matter of design, and the taxonomy's 31 categories are counted, curated, and documented as the four-pillar workload — with release engineering, the question this paper was most directly asked, coming out in favour of declarativity: individual releases become reviewable diffs against a locked world, boot-level tests and golden-image gates catching the failure classes that mutable packaging ships. The distribution ships offensive tooling strictly for authorised security work, and the safety defaults are part of the OS rather than documentation beside it. Kailash is a work in progress at the kailash-os project; the implementation phases, the packaging ledger, and the manifest seed live in the companion implementation plan (Revision 1.1), and the public documentation lives on the project website.

11. References

[1] Kali Linux. "Kali Linux Tools." kali.org/tools. Accessed 30 September 2026.

[2] Kali Linux (Offensive Security). “About Kali Linux.” kali.org/about-us. Accessed 30 September 2026. (Distribution model: WHAX/SLAX → BackTrack → Kali lineage, category navigation, live ISO.)

[3] balsoft. “KaliNix.” GitHub repository. 2021. (Single-flake pentest toolbox; proof-of-concept for the flake pattern.)

[4] fabaff. “nix-security-box: Tool set for Information Security professionals.” GitHub repository. 2020–2026. (Category-file NixOS module pattern.)

[5] redcode-labs. “RedNix: Hackable NixOS container and devShells collection.” GitHub repository. 2021. (Container-first pentesting shell posture.)

[6] Spectrum OS. spectrum-os.org. (Source-based, flake-defined bootable system posture; acknowledged design inspiration.)

[7] NixOS Foundation. “NixOS & the Nixpkgs manual.” nixos.org/manual. (Declarative system configuration and module system.)

[8] Nix Community. “NixOS Wiki – Flakes.” wiki.nixos.org/wiki/Flakes. (Flake inputs, locking, purity.)

[9] hercules-ci. “flake-parts.” GitHub repository. (Module-flakified flake structure used by the packages overlay.)

[10] NixOS Discourse and community templates. “Nix flake with CUDA-enabled PyTorch.” discourse.nixos.org (threads 55763, 11272); mschoder. “nix-cuda-template.” GitHub. (The CUDA-under-Nix failure modes and working template patterns.) 2021–2025.

[11] FerranAD. “nixos-cuda-torch.” GitHub repository. (uv2nix-based CUDA PyTorch flake pattern.)

[12] PyTorch Foundation. “Nightly dependency chain compromise notice.” pytorch.org/blog. December 2022. (Supply-chain case: compromise of the nightly dependency chain.)

[13] NVIDIA. “garak: LLM vulnerability scanner.” GitHub repository. Apache-2.0. (Probes/detectors for known LLM failure modes.)

[14] Microsoft. “PyRIT: Python Risk Identification Toolkit for generative AI.” GitHub repository. MIT. (Multi-turn adversarial orchestration: Crescendo, Tree of Attacks.)

[15] promptfoo. “promptfoo: Test your LLM app before your users do.” GitHub repository. MIT. (Application-specific adversarial generation; CI-native evaluation.)

[16] IBM Research / Trusted-AI. “Adversarial Robustness Toolbox (ART).” GitHub repository. (Reference implementations of evasion, poisoning, extraction, and inference attacks.)

[17] Microsoft. “Counterfit: a generic tool for adversarial machine learning.” GitHub repository. 2020–2022. (Endpoint-probing CLI; archived.)

[18] mmaitre314. “picklescan: Security scanner detecting Python Pickle files performing suspicious actions.” GitHub repository. (Pickle payload scanning; also HF-hub integrated.)

[19] HiddenLayer. “modelscan: Model serialization scanner.” GitHub. (Model artifact scanning.)

[20] DataDog. “GuardDog: npm/PyPI malicious package detection.” GitHub repository. (Malicious-package detection for the AI supply chain.)

[21] sigstore. “model-transparency: Supply chain security for ML models.” GitHub repository. (Model signing via Sigstore bundles; transparency-log-backed verification.)

[22] sigstore. “Sigstore: Software signing for everyone.” sigstore.dev. (Keyless artifact provenance.)

[23] OWASP Foundation. “OWASP Top 10 for LLM Applications, v2 (2025).” genai.owasp.org. (Ten canonical LLM application risks; prompt injection, supply chain, poisoning, excessive agency.)

[24] MITRE ATLAS. “Adversarial Threat Landscape for AI Systems, 2026.05.” atlas.mitre.org. (16 tactics, 101 techniques, 77 sub-techniques, 68 case studies; the agentic technique set.)

[25] MITRE ATT&CK. attack.mitre.org. (The adversary-tactics navigation that Kali’s tools-site taxonomy mirrors.)

[26] Greenbone. “Greenbone Vulnerability Management (GVM/OpenVAS).” greenbone.net. (The vulnerability-analysis service stack.)

[27] WithSecure Labs. “Chainsaw.” GitHub repository. (Rust-based artefact forensics; first-class nix derivation target.)

[28] MLflow. “Machine learning lifecycle platform.” mlflow.org. (Experiment tracking/registry; shipped as a lab service module.)

[29] OWASP Foundation. “Application Security Verification Standard (ASVS), Version 5.0.0.” github.com/OWASP/ASVS. May 2025. (17 chapters, ~350 testable requirements; the general application baseline AISVS scopes itself above.)

[30] OWASP Foundation. “Artificial Intelligence Security Verification Standard (AISVS), Version 1.0.” github.com/OWASP/AISVS. June 2026. (12 chapters × 3 verification levels; Ch10 covers MCP Security.)

[31] OWASP Foundation. “Machine Learning Security Top 10" and the “ML Security Verification Standard (MLSVS).” owasp.org/www-project-machine-learning-security-top-10. (The predictive-AI companion standards; M01-M10.)

[32] OWASP Foundation. “Top 10 for LLM Applications: 2026 edition.” genai.owasp.org. August 2026. (Supersedes the v2:2025 ordering in Kailash’s mappings; cross-mapped to the Agentic AI Top 10, ASI01-09, and to ATLAS.)

[33] 0x4m4. “HexStrike AI: MCPAgents – an AI-powered offensive security MCP framework.” GitHub repository. 2025–2026. (150+ security tools exposed to AI agents over MCP with autonomous decision engines; packaged as a Layer 0-7 deliverable.)

[34] TriV3. “MCP-Kali-Server: expose Kali Linux security tools to MCP clients.” GitHub repository. 2025–2026. (The VS Code/Claude-pentest-assistant bridge; packaged as a Layer 0-7 deliverable.)

[35] QuinnBast. “Malicious-MCP: proof-of-concept of an untrusted MCP proxy server.” GitHub repository. 2026. (Packaged into Layer A-3 as verification lab material; the CI target proving the scanner suite detects tool-disguise patterns.)

[36] nixified.ai. “flake: A Nix flake for many AI projects.” GitHub repository. AGPL-3.0. (Per-vendor GPU variant naming; nixConfig binary-cache posture; adopted in Section 5.6.)

[37] nixvital. “ml-pkgs: Machine Learning and Data Science Nix Overlays.” GitHub repository. (torch-family/jax-family/data-utils overlays; the documented fallback path in Section 5.6.)

[38] Determinate Systems. “Zero to Nix: Your guide to learning Nix and flakes.” zero-to-nix.org. (The flakes-first learning path reference for the site’s onboarding track.)

Reading map. The prior OSS designs this work directly builds on are [3]–[6]; the Nix substrate and module system [7]–[9]; the CUDA-under-Nix engineering corpus [10]–[12], [36]–[37]; the offensive tooling ecosystem packaged into Layer ATTACK [13]–[19]; the supply-chain and provenance tooling [18]–[22]; the agent-rig MCP servers packaged into Layer OPS [33]–[34]; and the standards stack that induces the verification-standards category and the standard-first navigation [1], [23], [24], [29]–[32]; the onboarding track [38].