AGIF Tasklet Cells

Verifier-Backed Offline AI Artifacts for Native Applications

— *Danish Z. Khan*, Founder, ENFSystems LLC

11th March 2026

Table of Contents

Abstract

AGIF Tasklet Cells are a software-artifact model for embedding bounded, offline AI capabilities directly inside native applications. A Tasklet Cell is a single-task, contract-driven artifact packaged with explicit input and output schemas, integrity metadata, and a Verifier Pack, and it may execute only under a local Runner that validates the artifact and enforces offline-first execution, bounded authority, resource limits, and fail-closed behavior. Rather than treating intelligence as a remote service or an open-ended agent loop, the model supports a hybrid execution pattern in which deterministic tool logic handles explicit and stable operations, compact local knowledge provides bounded task context, and optional micro-models support narrowly scoped roles such as routing, classification, extraction, and ranking.

The paper evaluates this architecture as a bounded local artifact system rather than as a claim of general intelligence. Empirical evidence is anchored to the clean N30 release bundle rooted at commit 78a1635 (2026-03-09), with 30 repeated runs across five suites and 2,400 total evaluated units. In the clean anchor, the release-readiness sweep passes in full, the scoped hallucination-safety benchmark reports zero unsafe-allow events with perfect abstention and reason accuracy, the fail-closed negative suite rejects all provided malformed and adversarial cases with the expected failure classes, the reasoning-trace suite achieves complete schema validity, evaluated-output repeatability, and evidence alignment, and the strengthened A6 benchmark passes with zero MAE for grand total, tax total, and subtotal together with full numeric grounding on the frozen public-seed benchmark. These results support bounded task-quality claims for invoices and receipts in en-US, de-DE, and es-ES, not universal finance-document extraction.

The paper further grounds the architecture through CellPOS, a private-pilot point-of-sale case study in which offline Cells support bounded local memory, reasoning trace, anomaly detection, kitchen bottleneck insight, plain-language diagnostics, and reorder-preference recall inside a packaged offline product workflow. The central claim is therefore narrow but concrete: useful task-specific AI functionality can be shipped as verifiable offline software artifacts rather than as remote services, with a local enforcement boundary and an executable evidence surface strong enough to support a product-shaped implementation path.

1. Introduction

1.1 Why Cloud-Centric Agent Systems Are Hard to Trust, Certify, and Embed

Recent progress in tool-using and agentic AI systems has made it technically feasible to attach language-driven reasoning modules to business software, desktop workflows, and operational tools. What remains much less settled is whether such systems can be trusted, assured, and embedded as stable product features when their dominant deployment model is remote, mutable, and infrastructure-dependent. The core difficulty is not simply that AI systems can make mistakes. It is that many are deployed as externally managed services with broad tool surfaces, evolving behavior, and trust boundaries that sit partly outside the embedding application. From an assurance perspective, that is a profoundly different engineering object from a bounded software component.

Public guidance on cloud security and privacy has long noted that outsourcing data, applications, or infrastructure to public cloud environments introduces security and privacy considerations that are inseparable from the complexity of the service model itself (Jansen & Grance, 2011). In practical terms, application inputs, intermediate representations, or derived outputs may leave the host environment, while correctness and availability become partly dependent on networks, third-party change-management practices, service-side deprecations, and vendor-controlled operational policy. For product teams building native software, this creates a mismatch: the application needs a stable capability with a clear interface and predictable failure semantics, while the AI layer can behave more like an external operator whose privileges, dependencies, and behavior change over time.

The problem becomes sharper when tool use is added. Classical security design principles such as least privilege, fail-safe defaults, and complete mediation assume that authority is minimized, explicitly granted, and checked at each access boundary (Saltzer & Schroeder, 1975). Those principles become harder to maintain when a language-driven system can browse external resources, invoke tools, or operate through mutable orchestration layers. Contemporary security guidance for large language model applications now treats prompt injection, insecure output handling, and supply-chain vulnerabilities as first-order concerns rather than edge cases (OWASP Foundation, 2024). Empirical work on prompt injection against real LLM-integrated applications likewise shows that tool-connected systems can be induced into unsafe behavior through malicious or adversarial content routed through otherwise ordinary workflows (Liu et al., 2023).

These are not merely philosophical objections to cloud AI; they are system-design concerns. The NIST AI Risk Management Framework treats trustworthy AI as a matter of context-sensitive risk management across design, development, deployment, and use rather than as a property that can be assumed once a model performs well on a benchmark (National Institute of Standards and Technology, 2023). For native applications, that framing matters because many useful AI features are not built as stand-alone products. They are embedded inside products that already carry obligations around reliability, bounded behavior, privacy, and maintainability.

This paper begins from a narrower premise than much of the prevailing agentic discourse. Many commercially valuable AI functions are structured, repetitive, and tightly coupled to local application state. Routing, extraction, classification, ranking, policy checks, and bounded diagnostic assistance often do not require open-ended autonomy or remote-service dependence. In those settings, the relevant engineering question is not how to maximize generality at all costs. It is how to package useful capability so that it can be validated locally, bounded operationally, and audited like a serious software component.

1.2 Why Many Useful AI Capabilities Should Ship as Verifiable Local Artifacts

This paper argues that a large class of practical AI features should be shipped not as remote services, but as verifiable local artifacts. The logic is straightforward. If a capability is narrow enough to be described by explicit input and output contracts, bounded enough to run under local resource limits, and important enough to require predictable failure behavior, then it should be possible to deliver that capability as an artifact that is validated before execution and constrained while it runs.

That shift changes the engineering posture in several useful ways. It moves the trust boundary inward, from a vendor-controlled service boundary to a local validation-and-execution boundary. It turns runtime validity into something that can be checked rather than merely assumed. It also separates concerns that are too often blurred together in cloud-agent systems: how a capability is trained or assembled, and how that capability is packaged, verified, and executed at deployment time. This separation aligns well with secure software lifecycle thinking, which emphasizes integrity, provenance, and disciplined release practices across the software development process (Souppaya et al., 2022).

The architectural ingredients required for such an artifact model are not imaginary. Explicit contracts can be expressed using JSON Schema, which provides a mature mechanism for machine-checkable interface definitions (JSON Schema, 2022). Canonicalized JSON representations can be made hash-stable using the JSON Canonicalization Scheme, which exists precisely so structured metadata can be represented in a deterministic, hashable form (Rundgren et al., 2020). Capability-oriented execution environments also provide a useful conceptual precedent. In the WASI design principles, the absence of ambient authority means that code receives only explicitly granted handles rather than inheriting global namespaces and implicit resource access (WebAssembly/WASI, n.d.). For a bounded artifact architecture, that is not a cosmetic preference; it is the difference between a constrained local module and a quietly privileged local agent.

This artifact-oriented framing also clarifies an important distinction between authenticity and validity. A signed or correctly packaged artifact may still be unacceptable for execution in a given environment if its local verifier does not pass, if its declared contracts do not match local policy, or if its requested authority surface is broader than the host is prepared to allow. In other words, provenance matters, but provenance alone is not enough. Validity must be established locally under the target execution boundary.

The project lineage behind Tasklet Cells reaches the same conclusion from a different direction. ENF Technical Note 01 frames cloud- and OTA-centric embedded AI as a lifecycle, privacy, and attack-surface problem, while ENF Technical Note 04 formalizes ENF as a compile-time framework centered on boundedness, determinism, offline operation, and conformance-oriented artifacts. ENF Technical Note 02 further situates that posture against TinyML toolchains, secure boot, energy harvesting, and formal-verification literature. The broader AGIF concept paper then frames AGIF as a research programme rather than as an already achieved general-intelligence system (Khan, 2025a, 2025b, 2025c, 2026). Tasklet Cells translate the relevant constraint logic from that lineage into a software-artifact form suitable for native applications.

The resulting thesis is intentionally narrower than a general argument against remote AI. The claim is not that all AI should be tiny, local, or immutable, nor that remote services have no legitimate place. The claim is that many product features currently implemented as loosely controlled service calls can instead be packaged as bounded, offline-capable, verifier-backed artifacts that are easier to trust, easier to embed, and easier to reason about under failure.

1.3 Scope and Thesis of This Paper

The scope of this paper is deliberately limited. It does not claim that AGIF has been achieved. It does not claim that broad fabric-level coordination, global descriptor exchange, or open-ended multi-cell adaptation have been solved. It does not present Tasklet Cells as general-purpose conversational agents, nor as a universal answer to every AI deployment pattern. Instead, it defines and evaluates a bounded artifact architecture for embedding narrow AI capabilities directly inside native applications.

The central object of study is the Tasklet Cell. A Tasklet Cell is a single-task, contract-driven software artifact packaged with integrity metadata and a Verifier Pack. It may execute only under a Runner that validates the artifact, enforces offline and capability restrictions, applies resource bounds, and fails closed when verification or execution conditions are not satisfied. The paper also defines the surrounding concepts needed to make that architecture coherent in practice: bundle structure, conformance semantics, bounded local memory, replay and observability, gateway enforcement, and the relationship between immutable core logic and reversible personalization.

A key design invariant follows from that framing: a Cell is not valid unless its Verifier Pack passes under the target Runner and target policy. The artifact is therefore not trusted by default. It becomes executable only after local validation under explicit policy. This is one of the main ways in which the proposed architecture differs from both ad hoc plugin systems and loosely bounded agent frameworks.

The paper advances five bounded claims. First, a Tasklet Cell can be defined as a contract-driven artifact rather than an ambient process. Second, a local Runner plus Verifier Pack can serve as an enforcement wall that supports offline-by-default execution and fail-closed behavior. Third, useful capability can be produced through a hybrid pipeline that combines deterministic tools, compact local knowledge, and optional micro-models for narrow tasks such as routing, classification, extraction, and ranking. Fourth, the architecture can be evaluated through executable evidence rather than narrative assertion. Fifth, the model is commercially relevant enough to justify a concrete case study, which this paper provides through CellPOS as a private-pilot point-of-sale embedding.

That is the claim boundary. The broader AGIF fabric remains future work, and the paper treats it accordingly. The supporting evidence should likewise be read at the artifact layer rather than at the level of general intelligence or market-scale deployment. As discussed later in Sections 14.8 and 14.9, the current evidence supports offline verifiable local Cell artifacts, local enforcement, and a product-shaped case-study path; it does not support claims that AGIF has been realized, that task performance is universally robust, or that CellPOS has been commercially validated at scale. The CellPOS proof materials document a real packaged offline product baseline with completed productization, offline licensing, offline signed manual updates, and a recorded “Ready for private pilot” classification, which supports the claim that bounded local intelligence can be embedded into a product-shaped workflow. They do not, by themselves, justify claims of broad commercial rollout, universal field validation, or completed production hardening in every operational dimension (ENFSystems LLC, 2026).

The paper should therefore be read as contributing one disciplined layer of a broader research stack: the artifact layer, the local enforcement layer, and the evidence needed to argue that useful bounded intelligence can be shipped as a software product feature without inheriting the full risk surface of cloud-centric agent systems.

2. Contributions

This paper makes five bounded contributions. They are intentionally narrower than the broader AGIF research agenda and are stated at the level supported by the current implementation and evidence. The contribution surface is therefore artifact-layer rather than fabric-level: the paper defines a bounded deployment unit, a local enforcement model, a controlled capability pipeline, an executable evidence method, and a concrete product-shaped case study. It does not claim that AGIF has been realized, that open-ended multi-cell coordination has been solved, or that the present implementation establishes general-purpose AI capability.

2.1 Bounded Tasklet Cell Artifact Model

First, the paper defines the Tasklet Cell as a bounded software artifact for embedding narrow AI capabilities directly inside native applications. A Tasklet Cell is not presented as an ambient process, a general-purpose conversational agent, or a free-form plugin with implicit authority. Instead, it is defined as a single-task, contract-driven artifact packaged with explicit input and output schemas, integrity metadata, a Verifier Pack, a machine-checkable bundle structure, and a clear boundary between immutable core logic and optional local personalization state.

This formulation contributes a packaging model in which validity is not assumed from origin alone. A Cell becomes executable only after it passes local validation under the target Runner and target policy. In that sense, the artifact model shifts the unit of deployment from a service endpoint to a verifiable local capability, while also making provenance, conformance, supply-chain integrity, and bounded personalization first-class parts of the artifact boundary rather than optional hardening layers. Where software-component inventories are attached to support provenance and supply-chain transparency, recognized SBOM practices provide a standard mechanism for documenting component relationships, including established formats such as SPDX and CycloneDX (CISA, n.d.; CycloneDX, n.d.; National Institute of Standards and Technology, n.d.; SPDX, n.d.).

2.2 Runner, Verifier Pack, and Fail-Closed Enforcement Model

Second, the paper contributes an enforcement model built around a Runner and a Verifier Pack. The Runner acts as the local enforcement wall: it validates artifact structure, checks integrity metadata, enforces offline and capability restrictions, applies resource bounds, and fails closed when validity or runtime conditions are not satisfied. The Verifier Pack is not treated merely as a development-time convenience; it is part of the runtime validity rule.

This contributes a concrete execution model in which a Cell is verifier-backed rather than self-asserting and in which Cells remain untrusted until they are verified under local policy. The result is an architecture with deny-by-default execution semantics, deterministic rejection behavior, explicit enforcement reporting, and default denial of network access and host authority unless such authority is explicitly granted by local policy. This operational stance is consistent with capability-oriented execution designs that avoid ambient authority (WebAssembly/WASI, n.d.). The contribution is therefore not merely that execution is sandboxed, but that validity, execution permission, and failure behavior are all brought under an explicit local control boundary.

2.3 Hybrid Intelligence Pipeline for Task-Specific Offline Capability

Third, the paper contributes a hybrid intelligence pipeline for task-specific offline capability. Rather than assuming that useful behavior must come from a large monolithic model, the architecture combines deterministic tool logic, compact local knowledge, optional micro-models for narrow roles such as routing, classification, extraction, and ranking, and a bounded local memory layer for reversible personalization.

This contribution is architectural rather than purely algorithmic. It shows how useful local intelligence can be assembled from bounded parts under explicit contracts, with deterministic checks wrapped around the points at which probabilistic inference is allowed. The resulting deployment model is aimed at capability per artifact rather than maximal generality, and at controlled composition rather than open-ended agent behavior.

2.4 Executable Evidence Methodology

Fourth, the paper contributes an executable evidence methodology for evaluating the artifact architecture through repeated, named suite execution rather than narrative assertion alone. This emphasis on auditable artifacts and repeatable evaluation is broadly consistent with community artifact-review norms that distinguish artifact availability, evaluation, and results validation (Association for Computing Machinery, n.d.). The purpose of this methodology is not to imply universal benchmark superiority. It is to make the current implementation auditable at the level it actually claims: release readiness, bounded safety behavior, fail-closed rejection, reasoning-trace validity, and scoped extraction performance under a frozen local evidence anchor.

In the clean N30 evidence anchor, this methodology records full release-readiness pass results, zero unsafe-allow events in the tested hallucination-safety suite, complete rejection in the fail-closed negative suite, deterministic reasoning-trace validity with full evidence alignment, and a strengthened A6 benchmark that passes with grounded grand total, tax total, and subtotal extraction on the frozen public seed benchmark. As detailed later in Sections 14.8 and 14.9 and Appendix C, this supports positive architectural claims while preserving a visible limitation boundary. The evidence is strongest at the level of bounded artifact execution and scoped numeric extraction; it is not a claim of universal finance-document understanding or broad open-domain reasoning performance.

2.5 CellPOS Private-Pilot Case Study

Fifth, the paper contributes CellPOS as a private-pilot case study showing how Tasklet Cells can be embedded into a real product domain. The value of this case study is not that it proves broad market deployment or general intelligence. Its value is that it demonstrates practical relevance under realistic product constraints and ties the architecture to a concrete packaged baseline rather than to an abstract integration sketch.

In the CellPOS setting, offline Cells are used to support bounded local memory, structured reasoning traces, anomaly detection, kitchen bottleneck insight, plain-language diagnostics, and reorder-preference recall. More concretely, the case study maps the architecture to four product-visible workflows: fraud and discount-abuse detection, kitchen bottleneck insight, plain-language diagnostics for TSE and runtime issues, and reorder or preference recall. As discussed later in Section 15.5 and Appendices E through H, this grounds the paper's architectural claims in an applied setting while keeping the scope disciplined: CellPOS is evidence of viable embedding in a product-shaped workflow and of private-pilot readiness, not a claim of universal validation or broad commercial release.

Taken together, these five contributions define the paper's actual claim surface. The paper contributes a bounded artifact model, a local enforcement model, a hybrid capability model, an executable evidence method, and a concrete case study. It does not claim to solve the broader AGIF fabric problem, to deliver open-ended agent autonomy, or to establish general-purpose AI capability. Those outcomes remain outside the contribution boundary of this paper, which is deliberately focused on the artifact layer, the local enforcement layer, and the evidence needed to justify embedding bounded intelligence as a product feature.

3. Problem Statement and Design Requirements

The problem addressed by this paper is not how to maximize generality in an unrestricted agent system. It is how to embed useful AI capability inside native applications in a form that is trustworthy, bounded, locally enforceable, and safe to ship as a product feature. In current practice, many AI features are delivered through remote services or loosely bounded agent stacks that can make assurance more difficult because correctness, privacy exposure, and availability become coupled to external infrastructure and broader authority surfaces (Jansen & Grance, 2011). The design problem, therefore, is to define an artifact architecture that can carry narrow AI functionality while preserving explicit control over validity, authority, resource use, and failure behavior.

Tasklet Cells are proposed as one answer to that design problem. A Tasklet Cell is intended to package a narrow capability in a way that is locally verifiable before execution and locally constrained during execution. For that claim to be meaningful, the architecture must satisfy a set of hard requirements, declare a set of non-goals, and maintain a small number of design invariants that are not negotiable. This section defines those requirements, boundaries, and invariants.

3.1 Hard Requirements

The first hard requirement is offline-first execution. This is an architectural choice, not a claim that local execution is automatically secure. Its purpose is narrower: to reduce dependence on remote infrastructure for correctness and to reduce classes of remote exfiltration and governance coupling associated with outsourced execution, while recognizing that local systems still require their own privacy and security controls (Jansen & Grance, 2011; National Institute of Standards and Technology, 2020a). A Tasklet Cell must therefore be executable without requiring a cloud service, remote policy oracle, or network call for basic validity or basic operation. The Runner may deny network access entirely or may expose only a tightly limited local capability surface under explicit policy, but remote dependency cannot be a condition of basic correctness. This requirement also implies a no-ambient-authority posture: a Cell must not inherit implicit filesystem, network, or process access simply because it has been loaded by a host. Any authority granted to a Cell must be explicit, minimal, and policy-bound. This posture aligns with capability-oriented designs that avoid ambient authority, including the WASI design principles and the Capsicum capability framework as a strong operating-system analogue (WebAssembly/WASI, n.d.; Watson et al., 2010).

The second hard requirement is local validity by verification. A Cell is not valid merely because it was produced by a known build pipeline or packaged by a trusted author. It becomes valid only when its Verifier Pack passes under the target Runner and target policy. Validity is therefore runtime-local and policy-sensitive rather than merely historical. The paper consequently requires that a Cell carry the artifacts needed for local validation, including explicit contracts, integrity metadata, and verifier material sufficient to gate execution.

The third hard requirement is bounded execution. The architecture must support enforceable bounds on the resources and authorities exercised by a Cell. At minimum, the Runner must be able to enforce limits on time, steps or fuel, memory, output size, and tool use, and must provide a clear mechanism for refusing execution or aborting execution when those limits are exceeded. The point of boundedness here is not only performance management. It is also to preserve analyzability, predictability, and safe embedding inside larger host systems.

The fourth hard requirement is strict contract discipline. Every Cell must declare explicit input and output schemas so that the host, the Runner, and the Cell share a machine-checkable interface boundary. In this paper, that mechanism is expressed through JSON Schema-compatible contracts (JSON Schema, 2022). A Cell cannot rely on implicit ambient context, ad hoc strings, or host-specific hidden state for correctness. This requirement ensures that Cells remain portable, inspectable, and testable as software artifacts rather than degrading into opaque mini-applications.

The fifth hard requirement is artifact integrity and provenance support. A Cell must carry integrity metadata sufficient for the Runner to detect tampering, mismatch, or structural invalidity before execution. Depending on the deployment profile, this may include hash manifests, canonicalized metadata, optional signatures, provenance attachments, or software-inventory material such as SBOM attachments (Cybersecurity and Infrastructure Security Agency [CISA], n.d.). Where structured metadata participates in hashing or signing, the architecture should support deterministic serialization so that integrity checks do not depend on incidental formatting differences. RFC 8785 provides a concrete canonicalization method for producing hashable JSON representations suitable for that purpose (Rundgren et al., 2020). The exact packaging profile may vary, but the architecture requires integrity and provenance to be first-class concerns at the artifact boundary.

The sixth hard requirement is bounded local memory and reversible personalization. The architecture must support local memory and adaptation without turning the core artifact into an uncontrolled learning system. Any personalization must remain local, bounded, policy-gated, and reversible. The immutable core of the Cell must remain distinguishable from any optional personal layer so that host applications and operators can reason clearly about what is fixed, what is adaptable, and what can be rolled back.

The seventh hard requirement is fail-closed behavior. Verification failure, schema failure, policy denial, integrity mismatch, or resource-bound violation must result in non-execution or rejected output rather than degraded ambient behavior. This requirement operationalizes the classic security preference for fail-safe defaults, least privilege, and explicit mediation rather than optimistic continuation under uncertainty (Saltzer & Schroeder, 1975). A Cell that cannot be validated or constrained cannot be allowed to behave like a partially trusted assistant.

The eighth hard requirement is local observability and replayability. The architecture must support structured local logs, replay records, and regression-oriented conformance replay without requiring cloud telemetry. The goal is not bit-identical logs across every environment; that would be an unnecessary overclaim. The goal is schema-stable, replay-sufficient local evidence that allows validation and execution outcomes to be reproduced under the same artifact, Runner version, and policy boundary. This posture is consistent with established log-management guidance that treats the collection, retention, protection, and analysis of log data as a core security practice (Kent & Souppaya, 2006).

Taken together, these requirements define the minimum bar for the architecture. A system that omits local validation, bounded execution, strict contracts, integrity artifacts, reversible personalization, or fail-closed behavior may still be a useful local AI integration, but it is not a Tasklet Cell system in the sense intended by this paper.

3.2 Non-Goals

The architecture is defined as much by what it excludes as by what it includes. The first non-goal is general-purpose chatbot behavior. Tasklet Cells are not intended to act as open-ended conversational systems with broad latent capability and unconstrained dialogue behavior. Their scope is narrow by design, and their value comes from bounded functionality rather than broad improvisation.

The second non-goal is unbounded tool-loop autonomy. The architecture does not aim to support arbitrarily recursive tool use, indefinite planning loops, or agentic behavior that grows its own authority through repeated interaction. A Cell may use deterministic tools or narrow helper mechanisms within explicit limits, but the system is not designed as an open-ended agent runtime.

The third non-goal is cloud dependency as a requirement. Nothing in the validity model, enforcement model, or capability model should require a remote service to make the architecture function. Remote coordination, remote update distribution, or remote orchestration may exist in adjacent deployment profiles, but they are not part of the core requirement set.

The fourth non-goal is ambient host authority. Tasklet Cells are not intended to inherit broad access to the filesystem, network, process space, or application internals simply by being loaded by a host. Any authority provided to a Cell must be explicit, minimal, and policy-bound.

The fifth non-goal is silent core rewriting or hidden model drift. The architecture does not aim to support hidden adaptation of the core artifact, background weight mutation, or invisible rewriting of decision behavior over time. If personalization exists, it must be constrained to a bounded local layer with explicit controls and rollback.

The sixth non-goal is maximal model complexity for its own sake. The paper is not trying to prove that the most capable system is the one with the largest model, widest tool surface, or richest orchestration layer. The design target is controlled usefulness, not maximal openness.

These non-goals matter because they prevent the architecture from being misread as a restricted version of a conventional agent framework. It is not a broad agent system with features removed. It is a different design center altogether.

3.3 Design Invariants

The architecture is anchored by a small set of design invariants that must hold across the rest of the paper.

The first invariant is the validity invariant: a Cell is not valid unless its Verifier Pack passes under the target Runner and policy. This rule is the foundation of the artifact model and the enforcement model. It means that origin, packaging, or host loading alone never imply trust.

The second invariant is the local enforcement invariant: the Runner is the enforcement wall. It is the component responsible for validating structure, applying policy, enforcing resource bounds, constraining capability use, and producing fail-closed behavior. External orchestrators, host applications, or packaging tools do not replace that function. They may participate in deployment, but they do not remove the need for local enforcement. In particular, external callers and orchestration layers must be treated as untrusted until their requests are mediated through the local policy and capability boundary. The trust assumptions behind the Runner itself are addressed later in the security and trust model.

The third invariant is the contract invariant: Cells communicate through explicit, machine-checkable interfaces. Inputs and outputs must remain schema-valid, and host integration must respect those contracts. This keeps capability narrow, replayable, and reviewable.

The fourth invariant is the boundedness invariant: execution, authority, and personalization must remain bounded. This includes bounded runtime resource use, bounded tool access, bounded output, and bounded local memory behavior. Without boundedness, the architecture loses the very property that makes it suitable for safe embedding.

The fifth invariant is the offline-first invariant: basic validity and basic execution must remain locally achievable. The architecture may support additional profiles, but it cannot depend on ambient cloud reachability as part of its core semantics.

The sixth invariant is the immutability-versus-personalization invariant: the core artifact remains stable, while any personalization is separate, explicit, local, and reversible. This separation is essential for preserving auditability and for preventing adaptation from turning into hidden drift.

The seventh invariant is the fail-closed invariant: when integrity, schema, policy, or resource conditions are not met, the architecture must prefer refusal over degraded execution. This is not merely an implementation preference. It is a defining property.

The eighth invariant is the evidence invariant: claims about the architecture must be supportable through executable artifacts and observable outcomes, not only through descriptive system diagrams. This invariant matters because the paper's argument depends not just on a clean architecture, but on the existence of evidence that the architecture behaves as claimed under repeated execution. As discussed later in Sections 14.8 and 14.9, that evidence boundary is artifact-layer and benchmark-scoped rather than a claim of universal task performance.

These invariants establish the boundary conditions for the rest of the paper. Later sections define the Tasklet Cell model, the bundle structure, the Runner semantics, the hybrid capability pipeline, personalization, replay, security, evaluation, and the CellPOS case study. All of those sections should be read as refinements of the requirements and invariants stated here, not as opportunities to weaken them.

4. Conceptual Lineage and Scope

This section documents internal design lineage; it is not offered as external validation. The paper sits at the intersection of two related but distinct lines of work. One is the ENF lineage of bounded, deterministic, offline-by-design intelligence. The other is the broader AGIF research agenda, which asks how larger capability might emerge from the composition of many constrained cells rather than from a single monolithic system. Tasklet Cells occupy a specific middle position between those two lines. They translate the boundedness discipline of ENF into a software-artifact form suitable for native applications while remaining substantially narrower in scope than the broader AGIF fabric vision (Khan, 2025a, 2025b, 2025c, 2025d, 2026).

This scope clarification matters because the paper does not claim that Tasklet Cells solve the full AGIF problem. The narrower claim is that a bounded local artifact layer can be defined and evaluated in a way that is consistent with the design logic from which it emerged. The conceptual lineage is therefore important, but the evidence boundary remains deliberately tight.

4.1 ENF Principles and Their Software Continuation

Embedded Neural Firmware (ENF) provides the conceptual starting point for this work. In the ENF whitepaper, ENF is defined as a sealed, offline-only neural program compiled into immutable memory and executed as a deterministic, scope-limited, silicon-bound “Neural BIOS” (Khan, 2025a). ENF Technical Note 01 extends that posture by arguing for fixed-function, statically deployed, OTA-free neural agents whose behavior remains bounded by design rather than managed through continuous remote governance (Khan, 2025b). ENF Technical Note 02 further positions ENF as a modular, task-bounded alternative to RTOS-centric embedded intelligence, emphasizing static composition, deterministic behavior, and strong limits on runtime variability (Khan, 2025c). ENF Technical Note 04 then formalizes ENF at the framework level as a compile-time system whose outputs are deterministic, sealed, OS-free, offline by design, telemetry-free, and free of dynamic allocation (Khan, 2025d).

The key lesson carried into Tasklet Cells is not any single hardware mechanism. It is a design posture: intelligence becomes easier to trust when it is bounded, easier to verify when it is deterministic, easier to deploy safely when it is offline by default, and easier to reason about when conformance artifacts are treated as part of the system boundary rather than as after-the-fact documentation. Tasklet Cells inherit that posture and reinterpret it in software form. Instead of a sealed embedded image, the deployment unit becomes a bounded software artifact. Instead of firmware-gated execution, a local Runner validates and constrains execution. Instead of ambient mutability, the model preserves a stable core artifact and places any optional personalization behind local policy and explicit controls. Instead of assuming that trust follows from build origin or package source, the model requires local verification before an artifact is treated as valid.

The software continuation of ENF therefore lies in the preservation of principles rather than in the literal copying of embedded mechanisms. The principles that carry forward most directly are explicit validity conditions, strict authority boundaries, offline-first operation, bounded resource use, conformance-aware packaging, and fail-closed behavior. Tasklet Cells adapt those principles to host applications, where the problem is not sealing a device for decades, but embedding useful local intelligence without inheriting the full fragility and opacity of cloud-centric agent systems. In that sense, Tasklet Cells should be read as a software continuation of ENF’s boundedness and conformance posture rather than as an attempt to reproduce ENF hardware invariants one-for-one inside a desktop runtime.

4.2 Tasklet Cells as the Artifact Layer

Tasklet Cells are the artifact layer that emerges from that continuation. They are the unit at which narrow capability is packaged, verified, and executed. In this paper, a Tasklet Cell is not defined as an abstract cognitive unit or as a biological metaphor. It is defined concretely as a single-task software artifact with explicit contracts, integrity metadata, a Verifier Pack, and a local execution boundary enforced by a Runner.

That position matters because it fixes the level of abstraction at which the paper makes its claim. The paper is not primarily about large-scale multi-Cell coordination, social behavior among agents, distributed learning, or emergent fabric-level cognition. It is about the software artifact that a host application can actually load, validate, run, reject, replay, and reason about. In that sense, Tasklet Cells are where the architectural discipline becomes operational.

Seen in this way, Tasklet Cells function as a safe deployment primitive. They give the host application a narrow and inspectable unit of capability. They also give the broader research program a concrete layer that can be evaluated through executable evidence rather than speculative diagrams alone. That is why the paper focuses so heavily on bundle structure, runtime verification, fail-closed enforcement, bounded local memory, replay, and observable behavior. Those are the properties that make the artifact layer meaningful as a systems contribution.

Tasklet Cells should therefore be read as the bridge between concept and deployment. They are narrow enough to be product-real, yet structured enough to carry forward the deeper logic of bounded intelligence into a host-application setting.

4.3 AGIF as Broader Fabric-Level Future Work

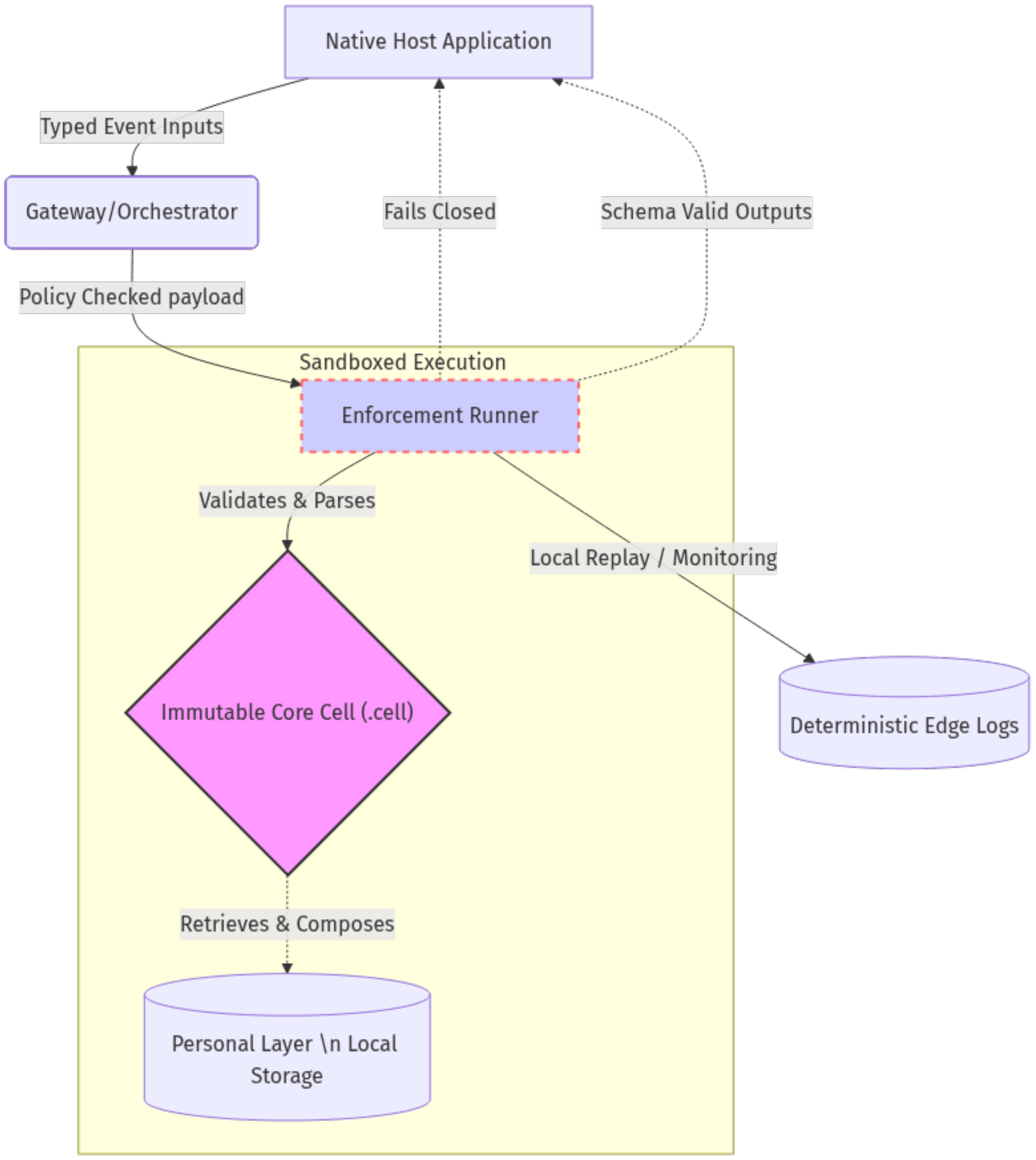
AGIF remains the broader research horizon rather than the accomplishment claim of this paper. In the AGIF concept paper, the central question is how larger capability might arise from many constrained cells that learn continually, exchange compressed skill descriptors, build shared world-model structures, and coordinate behavior through bounded governance and utility mechanisms (Khan, 2026). That question is important, but it is larger than the artifact-layer problem addressed here.

For that reason, this paper treats AGIF as future-facing scope rather than as direct, evidence-backed achievement. It does not claim that Tasklet Cells already realize fabric-level coordination at scale. It does not claim that descriptor gossip, global workspace behavior, broad world-model integration, or open-ended multi-Cell adaptation have been solved. It also does not claim that packaging a bounded local artifact is sufficient, by itself, to produce the broader fabric properties imagined by the AGIF agenda.

What the paper does claim is narrower and more defensible. If a broader AGIF fabric is ever to exist in a trustworthy and product-embeddable form, it will need a local unit that can be validated, bounded, replayed, rejected safely, and integrated without ambient authority. Tasklet Cells are proposed as one candidate for that unit at the software level. They do not complete the AGIF vision, but they do provide a disciplined artifact layer that makes future composition more plausible.

This scope boundary matters for the rest of the paper. Later sections define the Cell model, bundle structure, Runner semantics, hybrid capability pipeline, personalization model, replay and observability model, security posture, and evidence methodology. Those are artifact-layer contributions. As discussed later in Sections 14.8 through 14.10 and Section 16.7, the current evidence bundle is also artifact-layer evidence: it evaluates release readiness, fail-closed behavior, reasoning-trace validity, and bounded runtime properties for local Cells and their Runner boundary. It does not evaluate fabric-level coordination, descriptor exchange, world-model aggregation, or open-ended multi-Cell adaptation. The paper's scope should therefore be read accordingly. Figure 1 summarizes this artifact-layer architecture boundary.

Figure 1
Conceptual Architecture Summary



5. Related Work and Positioning

Tasklet Cells draw on several existing lines of work, but they do not fit neatly within any single one of them. Parts of the proposal resemble plugin packaging systems, software supply-chain frameworks, sandboxed offline runtimes, tool-using assistant architectures, and compact local AI pipelines. The distinctive claim of this paper is not that it invents each of those ingredients from scratch. Rather, the claim is that it combines them into a bounded software-artifact model in which validity is checked locally before execution, execution is constrained locally while it runs, and useful narrow capability can be shipped as a verifier-backed artifact rather than as a remote service.

This section positions Tasklet Cells relative to those neighboring areas. The goal is not to dismiss prior work, but to show where it stops short of the particular combination of properties emphasized here: explicit contracts, local validity gating, offline-first execution, fail-closed enforcement, bounded personalization, replayability, and an executable-evidence posture.

5.1 Plugin and Artifact Packaging Systems

A first relevant body of work concerns packaging itself. Software ecosystems already provide many ways to package executable logic, including shared libraries, plugins, signed packages, container images, and WebAssembly-based modules. There is also extensive work on artifact integrity, provenance, and distribution security, including secure update frameworks, supply-chain attestation systems, and software-bill-of-materials formats. The Update Framework (TUF) is especially relevant because it makes software-update trust and rollback resistance explicit rather than implicit (Samuel et al., 2010). in-toto is similarly relevant because it was designed to cryptographically enforce software supply-chain integrity across the steps that produce an artifact, rather than only at final distribution time (Torres-Arias et al., 2019). More recent work such as the SLSA specification and NIST cybersecurity supply-chain guidance shows that provenance, build integrity, and risk management can be treated as first-class engineering concerns rather than as after-the-fact hardening (National Institute of Standards and Technology, 2024; OpenSSF, n.d.). Sigstore adds a closely related operational example by combining artifact signing and verification with a transparency-log model for auditable software-signing events (Sigstore Project, n.d.). SBOM work adds another closely related layer by turning software composition into an explicit artifact property rather than an undocumented assumption (Cybersecurity and Infrastructure Security Agency, n.d.; National Institute of Standards and Technology, n.d.; OWASP Foundation, n.d.; SPDX Workgroup, 2024).

Tasklet Cells align with this literature in treating the artifact boundary as a first-class engineering object. Like prior packaging and supply-chain work, the model treats integrity metadata, provenance attachments, and inventory material as meaningful parts of the deployment story. However, Tasklet Cells differ in two important ways. First, the paper is not primarily about distribution security. It is about local execution validity. A Cell is not treated as trustworthy merely because it was distributed correctly or signed correctly; it becomes valid only when its Verifier Pack passes under the target Runner and policy. Second, the artifact is not just a payload plus an integrity envelope. It also carries a contract boundary and conformance material that directly influence whether the host is allowed to execute it at all.

In that sense, Tasklet Cells are better understood as runtime-gated capability artifacts than as generic packages. Conventional package systems tell the host what to install and how to verify that it came from somewhere trusted. A Tasklet Cell, by contrast, is designed to tell the host what narrow capability is being offered, under what machine-checkable contract, under what integrity conditions, and with what local validity gate. That added layer of runtime meaning is one of the key differences between the present model and ordinary packaging systems. At the same time, this paper does not claim to solve the full software supply-chain problem. Production signing infrastructure, revocation handling, transparency logging, and related operational controls remain separate concerns that any deployment would still need to address.

5.2 Offline Runtimes, Sandboxing, and Capability Control

A second body of work concerns offline runtimes and sandboxed execution environments. WebAssembly is especially relevant because it provides a standardized low-level execution substrate intended to be portable, compact, and efficiently executable across hosts (World Wide Web Consortium, 2026). WASI is even more directly relevant because its design principles explicitly reject ambient authority: modules are meant to receive only explicitly granted handles rather than inheriting global namespaces or unrestricted host access (WebAssembly/WASI, n.d.). Runtime projects such as Wasmtime further show that resource control is not merely theoretical. They provide practical mechanisms for bounding execution, including fuel-based interruption that is deterministic under the same program and fuel budget (Wasmtime Project, n.d.).

Tasklet Cells are strongly aligned with this runtime tradition. The Runner boundary described in this paper is conceptually compatible with capability-control and mediation-oriented traditions: loaded logic is not trusted by default, authority is granted explicitly, and the execution substrate is responsible for enforcement. This makes offline runtimes and capability sandboxes a natural execution baseline for the model.

At the same time, Tasklet Cells are not equivalent to a "Wasm module plus sandbox." A sandbox can prevent ambient access and constrain raw execution, but it does not by itself define artifact validity, contract semantics, shipped verifier material, bounded personalization, or replay-oriented conformance behavior. Put differently, a runtime substrate explains how code is isolated; it does not automatically explain when that code should count as a valid local capability artifact. Tasklet Cells build on sandboxing and capability control, but they add a higher-level validity and conformance model above the execution substrate itself.

5.3 Tool-Using Local Assistants and Agent Frameworks

A third relevant area is the growing ecosystem of tool-using assistants and agent frameworks. ReAct-style reasoning-and-acting loops and Toolformer-style tool-use induction show that tool calling, intermediate reasoning, and multi-step task composition are already well-established research directions in language-model systems (Schick et al., 2023; Yao et al., 2022). In product and open-source ecosystems, this broader direction has yielded a large family of orchestration frameworks, local assistants, and agent runtimes built around prompts, tools, memory, and execution chains.

This literature is relevant because Tasklet Cells clearly overlap with it at the level of practical intent: both are trying to make AI systems useful in real workflows. Yet the architectural posture is different. Much of the agent-framework ecosystem treats open-ended tool use, iterative planning, and loosely bounded orchestration as a feature. Tasklet Cells treat those properties as risks that must be tightly constrained or excluded. A Cell may combine deterministic tools, compact knowledge, and optional micro-models, but it is not designed as an autonomous planner that grows authority through repeated interaction.

The difference is therefore not simply that Tasklet Cells are smaller. The difference is that they are validity-gated, contract-bounded, and fail-closed by design. In many agent frameworks, correctness depends heavily on orchestration logic, prompt behavior, and runtime tool interactions that remain only partially constrained. Security guidance and empirical attack work now make clear that this broader class of LLM-integrated applications also carries concrete risks such as prompt injection and insecure output handling (Liu et al., 2023; OWASP Foundation, 2024). In the Tasklet Cell model, the central question is whether a narrow local capability can be packaged so that it is validated before execution and bounded while it runs. The result is a closer fit for product features that need predictable behavior, explicit host integration, and auditable failure semantics.

5.4 Small-Model and Hybrid Local AI Systems

A fourth body of relevant work concerns compact local AI and hybrid pipelines. The growing use of quantized models, lightweight inference stacks, and tool-plus-model pipelines weakens the assumption that useful AI capability must depend on large remote models. Quantization work is especially relevant here because it shows how model size, latency, and arithmetic cost can be reduced enough to make constrained local deployment more practical (Jacob et al., 2018). More broadly, hybrid systems that combine deterministic components with model-based inference show that useful capability does not need to come from a single monolithic model.

Tasklet Cells are aligned with this line of work in both spirit and method. The architecture explicitly allows narrow micro-models, compact local knowledge, and deterministic tools-first pipelines. It is compatible with quantized local inference and with hybrid retrieval or feature pipelines in which deterministic or structured stages provide the baseline and learned inference is used only where necessary. This is one reason the paper emphasizes capability per artifact rather than capability through maximal model scale.

Even so, small-model and hybrid local AI systems are not automatically the same thing as Tasklet Cells. A compact model can still be embedded in an opaque or weakly governed runtime. A hybrid local assistant can still lack explicit contracts, runtime validity gating, bounded personalization, or replay-oriented conformance behavior. Tasklet Cells therefore build on the feasibility of small and hybrid local intelligence, but add a packaging-and-enforcement layer that turns those techniques into auditable software artifacts.

5.5 What Tasklet Cells Add Beyond Prior Work

Tasklet Cells do not claim to replace the underlying contributions of packaging systems, secure supply-chain frameworks, sandboxed runtimes, agent frameworks, or compact local AI methods. Instead, they combine pieces from each of those traditions into a more narrowly specified artifact model with a different design center.

What they add can be summarized in five points.

First, they define a runtime-validity model rather than a mere packaging model. A Cell is not treated as valid because it exists, because it is signed, or because it came from a known source. It becomes valid only when its Verifier Pack passes under the target Runner and policy.

Second, they define a local enforcement boundary in which offline-first execution, capability control, resource bounds, and fail-closed rejection are treated as core semantics rather than as deployment preferences.

Third, they define a contract-driven capability unit. The artifact boundary includes explicit schemas and conformance expectations, so the host can reason about the Cell as a narrow software feature rather than as an opaque assistant.

Fourth, they define a bounded personalization model. Optional local memory and adaptation are supported, but they are kept separate from the immutable core artifact and remain local, reversible, and policy-gated.

Fifth, they define an evidence-backed systems posture. In line with community artifact norms, the architecture is evaluated not only through descriptive motivation, but through a released evidence bundle that tests release readiness, safety, fail-closed behavior, reasoning-trace validity, and bounded runtime behavior (Association for Computing Machinery, n.d.). As discussed later in Sections 14.8 through 14.10, the current evidence supports these artifact-layer claims directly while preserving an explicit limitation boundary rather than making benchmark-superiority claims over unrelated systems.

Taken together, these additions position Tasklet Cells as a bridge across several neighboring areas of work. They are more structured than generic plugins, more semantically meaningful than a raw sandboxed module, more bounded than an open-ended agent framework, and more deployment-aware than a standalone small-model pipeline. That combination is the paper’s real point of novelty.

Because specific commercial implementations evolve rapidly, the comparison below uses architectural classes rather than named products. The key comparison axis is whether the deployed unit of capability is a remote service, a locally orchestrated agent with broad host authority, or a bounded artifact whose validity is decided locally under explicit policy and fail-safe refusal conditions (OWASP Foundation, 2024; Saltzer & Schroeder, 1975; WebAssembly/WASI, n.d.). Table 1 summarizes this comparison across architectural classes.

Table 1

Novelty Matrix Versus Closest Architectural Classes

Feature	**AGIF Tasklet Cells**	**Hosted LLM API + Tool Orchestration**	**Local Agent Orchestrators with Broad Host Execution**	**Formal Verification / Proof-Carrying Assurance**
Primary deployment unit	Verifier-backed local capability artifact	Remote model service plus orchestration layer	Local runtime plus prompts, tools, and orchestration scripts	Program, model, or component plus proof obligations
Execution boundary	Local Runner with explicit policy, capability limits, and fail-closed rejection	Provider-controlled remote service boundary	Local host, shell, container, or application runtime	Proof or theorem-proving environment
Interface contract	Explicit machine-checkable I/O contract	API schema and prompt conventions	Prompt templates and ad hoc tool schemas	Formal property specification
Validity decision	Local validity gate: Verifier Pack must pass under target Runner and policy	Service availability, authentication, and application-side checks	Install/startup checks plus runtime orchestration success	Proof obligations satisfied before acceptance
Authority model	Explicitly granted local capabilities; deny by default	Network-mediated service calls; authority shaped by API and connector scopes	Often broad local filesystem, process, or tool authority	Not primarily an authority-management model
Failure posture	Refusal or rejection when integrity, policy, or resource conditions do not hold	Service errors, degraded outputs, or application-side fallbacks	Runtime loops, script failure, partial tool misuse, or host-side errors	Unproved or violated property blocks acceptance
Adaptation model	Optional bounded personal layer, local and reversible	Provider-side model updates and application-side memory layers	Mutable prompts, memory stores, tool sets, and orchestration logic	Typically static once verified
Primary evidence aim	Product-embeddable bounded capability with local assurance	Task performance and service quality	Workflow utility and automation breadth	Mathematical assurance of specified properties

6. Tasklet Cell Model and Terminology

This section defines the core objects used throughout the paper. The purpose is not merely terminological neatness. The paper's claims depend on a precise separation between the capability artifact, the local enforcement boundary, the conformance mechanism, the optional personalization layer, and any external coordination surface. Without those distinctions, the model would collapse back into a vague description of a "local agent," which is exactly what this paper is trying to avoid.

The terms defined here are therefore normative for the remainder of the paper. Later sections on bundle format, Runner semantics, bounded personalization, security, evaluation, and the CellPOS case study should all be read through these definitions.

6.1 Cell

A Tasklet Cell is the paper's basic unit of deployable capability. It is a single-task, contract-driven software artifact intended to embed a narrow function directly inside a host application. In this paper, "single-task" does not mean trivial. It means that the artifact is organized around one bounded capability surface rather than around open-ended dialogue or unconstrained multi-role behavior. A Cell may support routing, classification, extraction, ranking, diagnostics, or another narrowly delimited task, but it is not intended to behave as a general-purpose assistant.

A Cell is defined by five core properties.

First, it is contract-driven. It declares explicit input and output schemas that define the machine-checkable boundary between host and artifact.

Second, it is verifier-backed. A Cell does not assert its own validity. It carries a Verifier Pack that must pass under the target Runner before the Cell may be treated as executable.

Third, it is bounded. Its intended runtime behavior must fit within local policy limits on authority, time, memory, output, and optional tool use.

Fourth, it is artifact-structured. It is packaged with integrity metadata and a defined bundle layout so that the host and Runner can reason about it as a software object rather than as an opaque blob.

Fifth, it is separable from personalization state. A Cell may be associated with an optional Personal Layer, but the core artifact remains conceptually distinct from any local adaptation data.

In this sense, a Cell is best understood as a verifier-backed capability artifact rather than as a miniature autonomous agent. It is what the host loads, what the Runner validates, what the Verifier Pack checks, and what the user or operator can reject, replay, or reason about under explicit rules.

6.2 Runner

The Runner is the local enforcement boundary for Tasklet Cell execution. It is not merely a launch utility or compatibility shim. It is the component that determines whether a Cell is valid for local execution and, if so, under what constraints it may run.

The Runner performs five essential functions.

First, it performs structural validation. It checks that the Cell bundle conforms to the expected artifact structure and that required metadata, contracts, and verification material are present and well formed.

Second, it performs integrity and provenance checks. It evaluates the artifact's integrity metadata and any applicable provenance-related material according to the local policy profile.

Third, it performs Verifier Pack execution. It runs the verifier material that determines whether the Cell is locally valid under the target environment and policy.

Fourth, it performs enforcement. It constrains execution according to offline-first policy, capability restrictions, resource limits, output rules, and deterministic failure handling.

Fifth, it performs reporting. It produces enough structured information about validation, refusal, and execution outcomes to support replay, debugging, and evidence collection.

Architecturally, the Runner should be understood as the local mediation boundary for execution. This is analogous to the historical reference-monitor idea in that execution requests must be routed through a specific mechanism that decides what is allowed and what is not (Anderson, 1972). That analogy should not be overstated. The paper is not claiming that the Runner is a formally verified security kernel. Its trust assumptions, limitations, and threat-model role are addressed later in Section 13. What matters here is the narrower point that the host application may request execution, and an external orchestrator may suggest or trigger work, but neither replaces the Runner's role. A Cell is not trusted because the host invoked it, and an external caller does not gain trust merely by being present in the workflow. All execution requests must still pass through the same local mediation path rather than around it.

6.3 Verifier Pack

The Verifier Pack is the runtime-local conformance and validity package carried by a Cell. It is one of the concepts that most clearly distinguishes this architecture from ordinary plugin systems or weakly governed agent modules.

A Verifier Pack is not merely a collection of development-time tests. In the Tasklet Cell model, it is part of the validity rule itself. Its role is to determine whether the artifact is acceptable for execution under the target Runner and policy.

Conceptually, the Verifier Pack may include several kinds of checks:

- contract conformance checks,
- golden-case comparisons,
- structural or schema checks,
- comparison modes such as exact-match comparison, canonical JSON equivalence, numeric tolerance, or regex-based acceptance,
- and profile-specific checks tied to policy or environment.

The important point is not the exact internal format. The important point is that verification material ships with the artifact and participates in the local decision about whether execution is allowed. This makes the Verifier Pack a runtime gate rather than a separate quality-assurance artifact that can be ignored once deployment begins.

Because the Verifier Pack participates in validity, it also shapes the architecture's trust posture. A Cell without a passing Verifier Pack is not partially trusted. It is invalid for execution in the sense intended by this paper.

6.4 Personal Layer

The Personal Layer is the optional local layer in which bounded, reversible personalization state may be stored. It exists to support practical adaptation without allowing the core Cell artifact to drift silently.

This separation is necessary because useful local intelligence often benefits from user-specific adjustments, memory, or preferences, yet the paper's core architecture depends on a stable and auditable artifact boundary. The Personal Layer resolves that tension by separating what is fixed from what is adaptable.

The core Cell remains the immutable capability artifact. The Personal Layer may hold local memory, episodic preferences, threshold adjustments, approved examples, or other bounded forms of user-specific state, but it does so under three rules.

First, personalization remains local. It is not a hidden remote-learning channel.

Second, personalization remains bounded and policy-gated. It cannot expand the Cell into an uncontrolled learner.

Third, personalization remains reversible. Changes must be attributable, inspectable, and capable of rollback or deletion.

This means the Personal Layer is not part of the core validity claim in the same sense as the Cell artifact itself. Rather, it is a controlled extension point that allows useful adaptation while preserving the auditability of the core system.

6.5 Gateway Profile

The Gateway Profile is the optional interface profile through which external orchestrators, host-side coordinators, or adjacent systems may request access to Cell functionality without widening the Cell's authority boundary.

The need for this concept arises because real systems often include orchestration or coordination logic outside the Cell itself. A host application may have a scheduler, a background reasoning manager, or another local or adjacent process that decides when a Cell should be invoked. The architecture must therefore distinguish between the Cell's narrow capability boundary and any external logic that asks for work to be performed.

The Gateway Profile exists to keep that distinction clean. It is not a broad broker that grants ambient power. It is an explicit mediation surface that can:

- enforce request schemas and accept only the minimum context required by the Cell's declared contract,
- apply allowlists and rate limits or size limits,
- preserve offline-first policy,
- and ensure that external requests still pass through the same local Runner boundary rather than bypassing it.

In that sense, the Gateway follows the same general capability-oriented posture used elsewhere in the architecture: authority must be explicit, attenuated, and mediated rather than ambient (WebAssembly/WASI, n.d.). In this paper, the Gateway Profile is treated as optional because not every deployment needs external coordination. When it does exist, it should be treated as a profile around the artifact layer rather than as the center of the architecture. The Runner remains the enforcement wall, and the Gateway must not create a bypass around local validation or local policy enforcement. The security consequences of that boundary are addressed more fully in Section 13.

6.6 Validity Rule and Trust Boundary

The central validity rule of the architecture is simple and strict: a Cell is not valid unless its Verifier Pack passes under the target Runner and policy.

This rule has several consequences.

First, Cells are untrusted until verified. Package origin, host invocation, or developer intent do not replace local validity checks.

Second, the Runner is the trust boundary for execution. The host may load a Cell and an external orchestrator may request that it run, but neither is allowed to confer trust by itself. All execution decisions must still be mediated through the local Runner boundary.

Third, external callers remain untrusted until they are mediated locally. If an orchestrator or Gateway exists, it must still pass through the same local policy and capability controls.

Fourth, trust is narrower than authenticity. An artifact may be authentic in the sense that it was packaged or signed by an expected source, yet still fail the local validity gate because its verifier checks, policy profile, or runtime constraints do not pass.

Fifth, execution authority is narrower than host proximity. A loaded Cell does not inherit broad access to host resources merely because it resides inside the same application boundary. Authority must be explicitly granted and remains subject to the Runner's enforcement model.

This trust boundary is one of the paper's most important architectural commitments. It explains why Tasklet Cells are not merely plugins with extra metadata and not merely local assistants with a cleaner packaging story. They are artifacts whose validity, authority, and execution are mediated through a local boundary that is intended to be explicit, inspectable, and fail-closed.

The rest of the paper builds on that model. Later sections specify bundle structure, enforcement semantics, bounded personalization, replay, security, evaluation, and the CellPOS case study as refinements of the definitions given here.

7. Bundle Format and Integrity Model

The Tasklet Cell architecture depends on the artifact boundary being explicit, inspectable, and machine-checkable. For that reason, a Cell is not treated as an undifferentiated binary blob or as a loose collection of files assembled by convention. It is treated as a structured artifact with a defined layout, explicit contracts, verification material, and integrity metadata. This section describes that bundle model and explains how integrity, provenance, and conformance are tied to the artifact itself.

The point of the format is not aesthetic tidiness. It is to make local validity possible. If a host and Runner are expected to decide whether a Cell should be allowed to execute, then the artifact must expose enough structure for that decision to be made in a deterministic and policy-aware way. The bundle format therefore serves three purposes at once: it carries the capability payload, it carries the material needed to evaluate validity, and it carries the metadata needed to reason about integrity and provenance.

7.1 Canonical .cell Artifact Structure

A .cell artifact is the canonical packaging unit for a Tasklet Cell. Conceptually, it is a bounded software bundle with a stable internal layout. The exact serialization may vary by profile—for example, archive-based or directory-based packaging—but the logical structure should remain the same. The key requirement is that the Runner can interpret the artifact deterministically and apply the same validation logic regardless of host environment.

At a minimum, the canonical .cell structure should distinguish the following logical regions:

- *manifest*: the primary declaration of identity, version, capability boundary, and policy-relevant metadata;
- *contracts*: explicit input and output schemas that define the machine-checkable interface surface;
- *verifier material*: the Verifier Pack used to determine local validity;
- *logic payload*: the executable capability payload, such as compiled logic or a bounded runtime module;
- *integrity material*: hashes and related metadata used to detect mismatch or tampering;
- *optional support material*: model assets, provenance records, SBOM data, licenses, or profile-specific attachments.

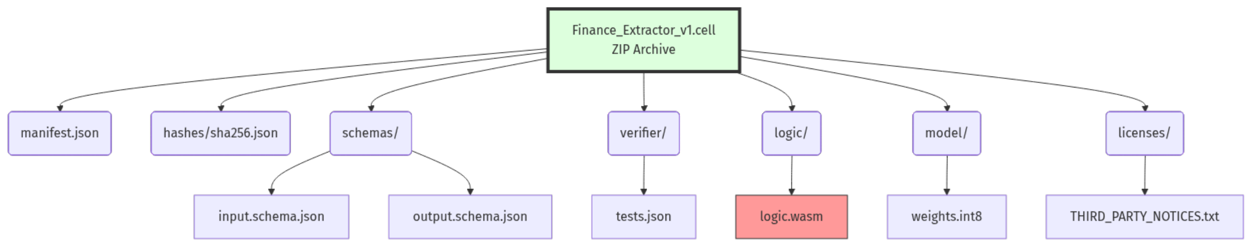
In the current profile illustrated later in Appendix A, a representative minimum layout is:

- manifest.json
- hashes/sha256.json
- schemas/input.schema.json
- schemas/output.schema.json
- verifier/tests.json
- logic/logic.wasm
- licenses/THIRD_PARTY_NOTICES.txt

The architecture does not require that every Cell include every optional directory. It does require that the bundle be interpretable without guesswork. Required components must be discoverable at stable paths or through stable manifest declarations. Optional components must not alter the meaning of required ones through ambiguity or path shadowing.

To preserve deterministic interpretation, the bundle profile should reject structural ambiguity and unsafe archive behavior. That includes, at minimum, rejection of path traversal or link-traversal classes after normalization and rejection of highly compressed or otherwise abusive archive content that could produce unsafe expansion during processing (MITRE, n.d.-a, n.d.-b). Duplicate normalized paths or other archive semantics that make meaning depend on host unpacking quirks are therefore incompatible with the integrity model described here. In that sense, the .cell structure is not merely a convenience for packaging. It is part of the trust boundary. Figure 2 illustrates this representative bundle layout.

Figure 2
Canonical .cell Bundle Layout Tree



7.2 Manifest, Contracts, and Schemas

The manifest is the primary declaration of what a Cell is, what it expects, and under what conditions it should be considered eligible for execution. It is the place where the artifact describes itself to the Runner and host in machine-checkable terms. The manifest should therefore be treated as policy-relevant metadata rather than as descriptive ornament.

At minimum, the manifest should support the following classes of information:

- artifact identity and version;
- declared capability or task role;
- contract references for input and output;
- references to verifier material;
- references to logic and optional model payloads;
- integrity-related metadata such as expected hash references;
- and profile declarations relevant to local policy or execution constraints.

The contracts carried by the Cell define the interface boundary between host and artifact. These contracts should be explicit, machine-checkable, and stable enough to support validation, replay, and host integration. In the baseline model, the most important contracts are the input and output schemas. The input schema defines the minimum typed context required for execution. The output schema defines the structure that a valid result must satisfy if execution succeeds. In this paper, those contracts are expressed through JSON Schema-compatible documents (JSON Schema, 2022).

A concise standards-pinning rule follows from this model: manifests intended for hashing or signing should be canonicalized using RFC 8785 (JCS) to avoid hash instability from key ordering or numeric formatting, while content digests use SHA-256 as specified in FIPS 180-4 (National Institute of Standards and Technology, 2015; Rundgren et al., 2020). Contracts are expressed using JSON Schema Draft 2020-12 (JSON Schema, 2022).

The central purpose of contract discipline is to eliminate ambiguity. A Cell must not depend on hidden ambient context, undeclared host variables, or ad hoc string payloads whose semantics change from one integration to another. By carrying explicit schemas, the artifact allows the host to prepare a valid request, the Runner to validate boundaries before execution, and replay or debugging tools to reproduce the same interaction later.

In this architecture, manifest and schemas are also linked to validity. A bundle whose schemas are missing, malformed, inconsistent with the manifest, or incompatible with the local policy profile is not merely poorly documented. It is structurally invalid for execution.

7.3 Verifier Pack Contents

The Verifier Pack is the bundle region that carries the local conformance and validity material. Its exact file layout may vary by profile, but its conceptual role does not: it is the part of the artifact that allows the Runner to determine whether this specific Cell is acceptable under the target environment and policy.

A Verifier Pack may contain several kinds of material:

- golden cases that map declared inputs to expected outputs or expected acceptance conditions;
- contract conformance checks that ensure the Cell respects its declared schemas;
- comparison rules such as exact match, canonical JSON equivalence, numeric tolerance, or regex-based acceptance;
- profile-specific checks tied to local policy, execution mode, or artifact profile;
- and expected structural conditions used to validate that the bundle is complete and coherent.

What matters most is not the internal packaging style but the architectural status of the Verifier Pack. It is part of runtime validity, not only part of development-time testing. If the Verifier Pack does not pass under the target Runner and policy, the Cell is invalid for execution in the sense intended by this paper. Where canonical JSON comparison is used, it should rely on a deterministic canonicalization method rather than ad hoc pretty-printing or formatter-dependent output (Rundgren et al., 2020).

This requirement creates a stronger artifact model than one in which tests are optional or external. A conventional package may ship with tests, but a Tasklet Cell ships with verifier material that participates directly in the decision to allow or deny execution. The Verifier Pack is therefore a conformance boundary, not merely a quality-assurance supplement.

7.4 Hashes, Provenance, and SBOM Attachments

The integrity model for a Cell begins with hashes, but it does not end there. Hashes provide the most direct mechanism for detecting tampering, mismatch, or unintended drift between declared and observed artifact contents. For that reason, a Cell should carry integrity metadata that allows the Runner to verify the contents of required bundle components before execution. In the current baseline profile, the hash manifest is hashes/sha256.json, and the named hash algorithm is SHA-256, which is specified in the Secure Hash Standard (National Institute of Standards and Technology, 2015).

Where structured metadata participates in integrity checks, the model should support deterministic serialization so that hashing and signing do not depend on incidental formatting differences. This matters especially for manifests and other structured declarations that may be generated by different tools or rewritten during packaging. If two logically identical metadata objects can hash differently because of unstable serialization, then the integrity model becomes unnecessarily brittle. RFC 8785 provides a concrete JSON canonicalization method suitable for that purpose (Rundgren et al., 2020).

Beyond hashes, the architecture can also support richer provenance and inventory material. Depending on the packaging profile, a Cell may carry:

- provenance metadata describing where the artifact came from or how it was produced;
- optional signatures or attestations bound to local policy or distribution requirements;
- SBOM material or equivalent inventory describing the artifact’s declared software components;
- and license notices or compliance-related metadata.

These attachments are not all required in every deployment. Their role is profile-dependent. However, the bundle model treats them as first-class optional attachments rather than as extraneous paperwork. This is important because provenance and inventory are often necessary for practical deployment decisions even when they are not, by themselves, sufficient to establish local validity. In the broader SBOM ecosystem, CISA and NIST frame an SBOM as a formal record of software components and their supply-chain relationships, while SPDX and CycloneDX provide concrete standard formats that can be carried as bundle attachments (Cybersecurity and Infrastructure Security Agency, n.d.; National Institute of Standards and Technology, n.d.; OWASP Foundation, n.d.; SPDX Workgroup, 2024).

The trust boundary remains narrow: provenance, authenticity, and inventory can support trust decisions, but they do not replace the local validity gate. A Cell may be authentic yet still invalid under local verification or local policy. In the Tasklet Cell model, integrity and provenance enrich the decision boundary; they do not bypass it.

7.5 Conformance Expectations

The bundle model implies a corresponding model of conformance. A conformant Cell is not merely one that can be unpacked or parsed. It is one that satisfies the structural, contractual, and verification conditions expected by the Runner and policy profile.

At a minimum, conformance should mean the following:

- the bundle structure is complete and unambiguous;
- required manifest and schema material is present and well formed;
- integrity metadata can be evaluated successfully;
- verifier material is present and executable in the expected profile;
- the logic payload is bound to the declared contracts and verification surface;
- and optional attachments do not create ambiguity about required artifact meaning.

Conformance in this sense is stronger than mere syntactic validity. A bundle may be parseable yet still fail conformance because contracts do not match, verifier material fails, integrity expectations do not hold, or policy-relevant profile requirements are violated. This is why the architecture treats conformance as part of the runtime decision boundary rather than as a static packaging checklist. Appendix A should be read as a concrete example of this model in the current evidence bundle, including the staged Runner validity sequence and the representative manifest, verifier, and hash-manifest paths.

Seen as a whole, the bundle format and integrity model define the Cell as a locally interpretable, locally verifiable software artifact. The host does not merely load code. It loads a structured capability artifact whose meaning, validity, and execution eligibility can be evaluated through explicit material at the artifact boundary. That is the core contribution of the bundle model, and it is what enables the Runner, Verifier Pack, bounded personalization, replay, and security sections that follow.

8. Runner, Enforcement, and Fail-Closed Execution

The Runner is the operational core of the Tasklet Cell architecture. The bundle defines what a Cell is; the Runner determines whether that Cell is valid here, under this policy, and under these execution constraints. For that reason, the Runner is not a convenience layer, a launch wrapper, or a host-specific implementation detail. It is the local enforcement boundary through which every execution request must pass.

This section defines the Runner's role at the level of system behavior. The central principle is simple: execution is not granted by default. A Cell must first survive validation, verification, and policy checks, and it must remain within explicit capability and resource limits while it runs. If those conditions are not met, the architecture must fail closed. In classical security terms, this aligns the Runner with fail-safe defaults, least privilege, and complete mediation rather than optimistic continuation under uncertainty (Saltzer & Schroeder, 1975).

8.1 Validation and Verification Pipeline

The Runner's first responsibility is to decide whether a Cell is locally valid for execution. That decision is made through a staged validation-and-verification pipeline rather than through a single yes-or-no flag.

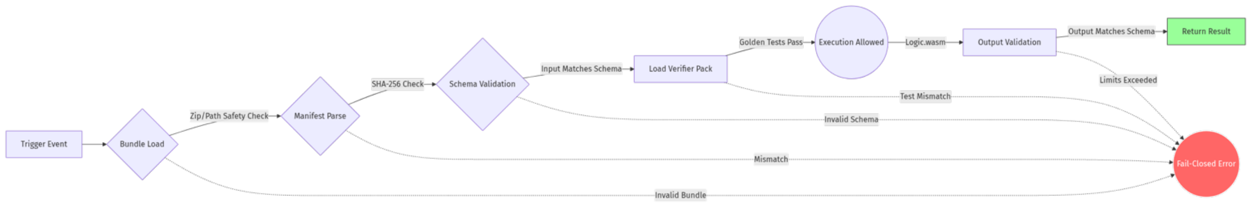
At a high level, the pipeline consists of six stages.

1. *Bundle ingestion and structural validation.* The Runner opens the .cell artifact, applies archive-safety rules, and checks that the required structural components are present and well formed. This includes stable path interpretation, rejection of ambiguous or unsafe archive semantics, and validation that the required manifest, schema, verifier, logic, and integrity regions are discoverable in the expected way.
2. *Manifest and contract validation.* The Runner parses the manifest and verifies that the declared contracts, profile fields, and artifact references are coherent. Input and output schemas must be present, machine-checkable, and internally consistent with the manifest's declarations.
3. *Integrity and provenance checks.* The Runner evaluates the integrity material carried by the artifact. At a minimum, this includes checking content hashes for required bundle components. Depending on the deployment profile, this stage may also include canonicalization-sensitive metadata checks, provenance validation, signature checks, or policy-based evaluation of inventory attachments such as SBOM material.
4. *Verifier Pack execution.* The Runner executes the Verifier Pack associated with the Cell. This is the decisive validity stage. The Cell is not treated as executable unless the verifier material passes under the local Runner and local policy profile.
5. *Policy binding and execution preparation.* If the artifact passes structural, integrity, and verifier checks, the Runner binds the Cell to the local execution policy. This includes the offline and capability profile, runtime bounds, comparison policy, output rules, and any profile-specific restrictions that must be enforced during execution.
6. *Execution enablement.* Only after the preceding checks pass may the Runner allow the logic payload to execute against a schema-valid request from the host.

The important architectural point is that validity is not assumed at any earlier stage. A syntactically correct bundle is not yet valid. An authentic or signed bundle is not yet valid. Even a structurally well-formed bundle with correct hashes is not yet valid. Local validity exists only after the artifact passes the full verification path under the target Runner and policy.

This pipeline is what makes the architecture more than a package format with a sandbox attached. It creates a local decision boundary in which structural correctness, integrity, verifier success, and policy compatibility all participate in the decision to allow execution. The same pipeline must apply regardless of whether the request originated directly from the host or indirectly through an optional Gateway Profile; there is no alternate fast path that bypasses local validation. That no-bypass posture is a direct application of complete mediation in operational form (Saltzer & Schroeder, 1975). Figure 3 summarizes the Runner validation and execution pipeline.

Figure 3
Runner Validation and Execution Pipeline



8.2 Offline-by-Default Execution

The second defining responsibility of the Runner is to enforce offline-by-default execution. In this paper, offline-first does not mean merely "usually local" or "able to cache results." It means that the basic validity and execution path of a Tasklet Cell must not depend on ambient cloud reachability.

Operationally, this implies three rules.

First, the Runner should deny network access by default. If a deployment profile allows any communication surface at all, it must be explicit, minimal, and policy-bound rather than ambient.

Second, the Runner should deny ambient host authority by default. A loaded Cell does not inherit unrestricted access to files, processes, networks, or host internals simply because it resides inside the application boundary.

Third, the Runner should expose only the minimum capability surface required by the Cell's declared contract and execution profile. The host supplies typed request context; the Runner decides what, if any, additional capabilities may be made available.

This design matters for two reasons. The first is safety: it prevents a Tasklet Cell from turning into an uncontrolled local agent simply because the host happens to sit near powerful resources. The second is portability of trust: if execution depends on ambient network access or undeclared host privileges, then the same artifact may behave differently in ways that are difficult to inspect, replay, or certify. This posture is consistent with capability-oriented execution designs that reject ambient authority and require explicit handles or grants instead (WebAssembly/WASI, n.d.).

Offline-by-default does not mean that every deployment is permanently isolated from all coordination. A Gateway Profile may mediate adjacent requests, and some profiles may allow tightly bounded local tools. But those are explicit deviations from the zero-ambient-authority baseline, not the default operating assumption.

8.3 Resource Bounds and Capability Limits

A third responsibility of the Runner is to enforce bounded execution in operational terms. The architecture does not treat boundedness as a vague aspiration. It requires concrete limits on what a Cell may consume or invoke while running.

At a minimum, the Runner should be able to apply bounds in the following categories:

- *time:* maximum wall-clock execution duration or equivalent timeout semantics;
- *steps or fuel:* maximum interpreter steps, instruction budget, or equivalent execution quota;
- *memory:* maximum memory consumption or growth permitted to the payload;
- *output size:* maximum size of returned structured output or emitted result material;
- *tool use:* maximum number of tool calls, plus per-tool allowlists and limits where tools are permitted;
- *request size:* maximum accepted input size and maximum validated request complexity;
- *personalization scope:* bounded read and write access to any optional Personal Layer data.

The exact mechanisms may depend on the runtime substrate and policy profile, but the architectural requirement is constant: these limits must be locally enforceable by the Runner rather than left to convention. Where the runtime substrate supports fuel-based interruption, a concrete example exists in Wasmtime, whose documentation states that fuel-based interruption is deterministic for the same program given the same fuel budget. This is useful as an implementation example for bounded execution, but it should not be overstated as proof of universal cross-platform determinism for every layer of the stack (Wasmtime Project, n.d.).

Capability limits are the authority-side analogue of resource limits. Even if a Cell stays within a time or memory budget, it should not be allowed to exceed the capability surface declared for it. This means that permitted tools, local stores, or helper functions must be explicitly enumerated and mediated. A Cell that attempts to cross those limits should not degrade into partial success through hidden fallback behavior. It should be refused or terminated according to the same fail-closed principles that govern validity.

The purpose of these bounds is not merely defensive programming. It is to keep the Cell understandable as a bounded feature rather than as an expanding process. Once authority and consumption are allowed to drift, the paper’s central claim about safe embedding begins to collapse.

8.4 Deterministic Rejection Behavior

The architecture requires not only rejection when conditions fail, but deterministic rejection behavior. A fail-closed system that refuses unpredictably, returns unstable error structures, or produces different boundary behavior from one run to another is still difficult to reason about and difficult to test.

For that reason, the Runner should produce a stable rejection envelope whenever validation, verification, policy, or runtime conditions are not satisfied. The purpose of the rejection path is twofold: to protect the host and to preserve the interpretability of system behavior.

At a conceptual level, rejection should have the following properties:

- *fail-closed semantics:* invalid or out-of-bounds Cells do not execute, and invalid outputs are not applied;
- *stable error classes:* distinct failure categories should map to explicit error codes rather than vague free-form messages;
- *deterministic structure:* the rejection payload should remain stable enough for replay, testing, and comparison under the same Runner version, policy hash, artifact hash, and validated input;
- *no hidden privilege escalation through fallback:* denial at one layer must not silently reappear as execution through another path.

A representative error envelope may therefore separate success from failure in a fixed structure, for example:

- success: { ok: true, data: ... }
- failure: { ok: false, error: { code: ..., message: ... } }

The exact syntax may vary, but the system should preserve stable semantics. Timestamps, unstable identifiers, or incidental host-specific noise should not be allowed to turn rejection behavior into an effectively nondeterministic surface.

Typical rejection classes in this model include structural invalidity, integrity mismatch, verifier failure, policy denial, resource-limit breach, schema invalidity, and output-conformance failure. The point is not to list every possible code here. The point is that rejection is treated as a first-class outcome with machine-checkable meaning, not as a side effect of execution failure. As discussed later in Section 14.4 and Appendix D, the current evidence supports complete rejection on the provided malformed and adversarial bundle set with explicit failure-mode reporting under repeated execution; it does not support any stronger claim about every possible future attack class or host-compromise scenario.

8.5 Enforcement Reporting

The final responsibility of the Runner is enforcement reporting. Because the architecture depends on local mediation and bounded execution, the system must provide enough structured information to show what was actually enforced, what failed, and under what local conditions the decision was made.

An enforcement report should not be confused with telemetry. It is a local systems artifact whose purpose is replay, debugging, audit, and evidence generation. Depending on the deployment profile, an enforcement report may include:

- the artifact identity and version;
- the bundle hash or equivalent artifact identifier;
- the Runner version;
- the policy profile or policy hash under which the decision was made;
- the enforcement modes that were active, such as strong, best_effort, or unavailable;
- the concrete limits applied to time, fuel, memory, output, and capability use;
- the final decision outcome, including rejection or execution-result class;
- references needed for local replay or conformance re-execution;
- and, where validation results are cached, a cache key bound at minimum to the artifact identifier, Runner version, and policy hash.

This reporting layer matters because enforcement that cannot be observed locally is difficult to verify in practice. A host may believe that a Cell was run offline, within bounds, and under policy, but without a structured local report that claim is weaker than it should be.

The broader architectural point is that the Runner does more than admit or deny execution. It produces a record of how that decision was made. That record supports evidence-backed evaluation, debugging, and audit without turning the architecture into a cloud-observability system.

Taken together, these five aspects define the Runner as the architecture's operational spine. It validates the artifact, verifies local validity, enforces offline-first execution, constrains resources and authority, rejects deterministically when conditions fail, and reports what it actually enforced. The sections that follow build on this enforcement model rather than treating it as an implementation detail.

9. Hybrid Intelligence Pipeline

The Tasklet Cell architecture does not assume that useful AI capability must come from a single large model or from an open-ended agent loop. Instead, it adopts a hybrid pipeline in which different forms of capability are assigned to the components best suited to perform them safely, efficiently, and predictably. Deterministic logic handles the parts of a task that should remain explicit and stable. Compact local knowledge provides bounded factual or structural support. Optional micro-models handle narrow pattern-recognition roles for which probabilistic inference is genuinely useful. The resulting capability is therefore compositional rather than monolithic.

This section explains that design choice. The aim is not to argue that hybrid systems are novel in the abstract. Rather, it is to show how the Tasklet Cell architecture gives hybrid capability a bounded, verifier-backed, product-embeddable form. Research lines such as ReAct and Toolformer already establish that tool use and model-based reasoning can be composed in practical systems (Schick et al., 2023; Yao et al., 2022). The claim here is narrower: a useful local capability can be assembled from bounded parts without turning the artifact into an unconstrained assistant.

9.1 Deterministic Tools First

The first principle of the pipeline is deterministic tools first. Whenever a task can be performed by explicit logic, schema-aware transformation, rule-based validation, or another bounded deterministic mechanism, that path should be preferred over probabilistic inference. This is not an aesthetic preference. It is an architectural choice that preserves auditability, replayability, and controllability.

Deterministic tools are especially appropriate for operations such as:

- schema validation,
- rule-based normalization,
- threshold or range checks,
- exact or canonical comparisons,
- field projection and transformation,
- structured filtering,
- explicit host-action gating,
- and any other operation whose semantics are better expressed as code than as inference.

Using deterministic tools first has several advantages.

First, it narrows the role of learned components. A micro-model does not need to decide everything; it only needs to decide the part of the task that genuinely benefits from statistical generalization.

Second, it improves explainability in the practical engineering sense. A result produced through explicit rule application, contract checking, or canonical transformation is easier to replay and debug than one that emerges from a latent model boundary with no explicit decomposition.

Third, it supports fail-closed execution. If a deterministic tool rejects a request, identifies a contract violation, or refuses a transformation, the Runner and host can respond with a stable and auditable rejection path rather than forcing a model to guess past the boundary.

In the Tasklet Cell architecture, deterministic tools should therefore be understood as the first layer of capability, not as a fallback after a model has already spoken. They define the safe surface on which the rest of the pipeline operates. This matters especially because open-ended tool loops have already been shown to expand security risk surfaces such as prompt injection and insecure output handling (Liu et al., 2023; OWASP Foundation, 2024).

9.2 Compact Knowledge Pack

The second component of the pipeline is the compact knowledge pack. A Tasklet Cell often needs access to local, task-relevant knowledge, but that does not imply the need for a large remote knowledge service or an open-ended retrieval stack. Instead, the architecture assumes that many useful tasks can be supported by small, bounded, locally shipped knowledge resources.

A compact knowledge pack may include:

- fixed mappings or dictionaries,

- field templates,
- controlled vocabularies,
- small domain tables,
- known aliases or normalization maps,
- ranking hints,
- local lexical retrieval assets such as term indexes or keyword-scoring resources,
- and, where the profile permits, narrowly scoped local retrieval assets appropriate to the Cell's declared task.

The important property is not merely size. It is boundedness and locality. The knowledge pack should be small enough to ship with the artifact or with a tightly coupled local deployment profile, stable enough to be hashed and versioned as part of the artifact boundary, and narrow enough to support the declared task without silently expanding the Cell's capability surface.

In practice, this layer allows a Cell to be more useful than either a raw model or a raw rule set alone. Deterministic tools can enforce structure; the knowledge pack can supply local domain context. For example, a Cell performing structured extraction or ranking may rely on local domain aliases, approved labels, fixed normalization resources, or local lexical retrieval without requiring a remote lookup service. In richer local profiles, a compact retrieval layer may also combine a bounded knowledge pack with lightweight vector or nearest-neighbor retrieval, provided that the retrieval path remains local, explicit, and subject to the same artifact and policy boundary.

This layer also improves controllability. Because the knowledge is local and bounded, it can be audited, versioned, replayed, and replaced under explicit artifact management rather than through hidden prompt edits or opaque server-side updates. That property is important for product embedding, where silent knowledge drift can be just as problematic as silent model drift.

9.3 Optional Micro-Models

The third component of the pipeline is the optional micro-model. The key word is optional. A Tasklet Cell is not defined by the presence of a neural model. Some Cells may be entirely deterministic. Others may include a small local model because the task requires pattern recognition, soft classification, ranking, or probabilistic routing that would be brittle or impractical to express purely through fixed rules. In the intended profile, such models should be compact, local, and compatible with bounded inference paths such as quantized or otherwise tightly controlled execution. Quantization literature is relevant here because it shows that integer-oriented inference can reduce model size and improve latency for on-device execution, making constrained local deployment more practical (Jacob et al., 2018).

In this paper, micro-models are intended for narrow roles such as:

- routing among bounded processing paths,
- classification over a declared label set,
- extraction support for structured fields,
- ranking among finite candidate sets,
- or other scoped judgments in which learned inference improves utility without redefining the artifact as an open-ended agent.

Several properties follow from that design choice.

First, the model role remains subordinate to the artifact boundary. The presence of a model does not dissolve the surrounding contracts, verifier checks, Runner policy, or fail-closed semantics.

Second, the model remains local and bounded. It is shipped with the Cell or with its profile-defined local assets, and it is executed inside the same local enforcement boundary as the rest of the artifact.

Third, the model remains task-specific. The paper does not require or assume a general-purpose reasoning model inside the Cell. The model is a narrow capability component, not the whole identity of the system.

Fourth, the model remains surrounded by deterministic checks. Inputs still pass through schema validation and policy controls. Outputs still must conform to declared output structures. Deterministic post-processing may still apply thresholds, candidate filtering, result gating, or other schema-aware checks before the host is allowed to treat the output as usable. Where repeatability matters, the surrounding profile should also prefer stable featurization and bounded numerical behavior over unconstrained floating-point or parallel execution paths.

This is why the paper speaks of micro-models rather than simply models. The point is not their exact parameter count. The point is that the model is only one bounded component inside a larger verifier-backed pipeline. Table 2 summarizes the baseline requirements for this neural Cell profile.

Table 2
Neural Cell Profile v1 Requirements

Component	Baseline Requirement	Description
Model family	Tiny classifier / router / extractor	Strictly limited to routing, classification, extraction, or ranking. Free-form generation is outside the profile.
Format and quantization	Flat Int8 tensors or equivalent bounded local format	Integer-oriented inference can reduce model size and latency and can improve repeatability by reducing floating-point variability in constrained runtimes. Deterministic execution claims should remain tied to measured evidence rather than assumed from quantization alone.
Feature extraction	Deterministic featurizer	Fixed tokenization and exact dimensional mappings with a pinned hash seed to reduce semantic drift across runs.
Allowed mathematical operations	matmul, add, bias, ReLU, plus bounded normalization where required	Minimal operator set intended to keep inference behavior inspectable and implementation scope narrow.
Decision boundary	Deterministic post-processing	Probabilistic outputs must pass through strict argmax, thresholding, or equivalent schema-aware gating before generating final structured outputs.

9.4 Composition as the Source of Capability

The architectural claim of this section is that capability comes from composition. A Tasklet Cell becomes useful not because one component is maximally powerful in isolation, but because deterministic tools, compact local knowledge, optional micro-models, and local enforcement are combined in a disciplined way.

That compositional view has several consequences.

First, it changes how capability should be evaluated. A useful Cell is not simply the one with the most expressive model. It is the one whose components work together under contracts, within bounds, and under local validation to produce reliable task-specific behavior.

Second, it changes how capability should be packaged. Instead of shipping a monolithic agent and trying to constrain it after the fact, the architecture packages a bounded pipeline whose parts already have differentiated roles.

Third, it changes how failure should be interpreted. Because capability is composed, failure in one component does not imply that the rest of the system must guess. A deterministic gate can reject. A schema can fail. A policy can deny. A model output can be discarded if it does not satisfy post-processing or conformance constraints. The system does not need to behave as though every task must be answered at all costs.

Fourth, it changes how local usefulness should be understood. The paper's argument is not that Tasklet Cells are "small AGI." The argument is that commercially useful local intelligence can often be achieved by composing bounded parts inside a verifier-backed artifact. In many product settings, that is a better engineering target than trying to embed a single unconstrained assistant and then hoping policy layers can domesticate it afterward.

Seen as a whole, the hybrid intelligence pipeline explains why Tasklet Cells can be both narrow and useful. They are narrow because each component is bounded and role-specific. They are useful because the architecture composes those components into a coherent local capability. As discussed later in Sections 14.8 through 14.10, the current evidence bundle is consistent with this artifact-layer framing: it supports routing-, classification-, extraction-, and trace-related claims at the level of bounded local execution, while still leaving a clear limitation boundary around full numeric extraction quality. That compositional structure is what allows the rest of the paper—training and packaging, bounded personalization, replay, security, evaluation, and the CellPOS case study—to remain product-real rather than drifting into open-ended agent design.

10. Training, Export, and Packaging Lifecycle

One of the easiest ways for an artifact architecture to become vague is to discuss runtime behavior at length while leaving the lifecycle that produced the artifact underspecified. This paper does not take that route. A Tasklet Cell may contain only deterministic logic, or it may include a micro-model as part of a hybrid capability pipeline, but in either case the deployment artifact is the endpoint of a concrete lifecycle: a task is defined, data are curated, a bounded capability is trained or assembled, the result is exported into an artifact-compatible form, verifier material is generated, and the final bundle is packaged for local execution under a Runner.

This section makes that lifecycle explicit. Its purpose is twofold. First, it prevents the architecture from sounding as though useful local intelligence appears without a training and packaging discipline behind it. Second, it draws a strict line between what may happen before deployment and what is allowed after deployment. Training may be expensive, exploratory, and iterative. Deployment, by contrast, must remain bounded, verifier-backed, and policy-controlled.

10.1 Dataset and Task Definition

The lifecycle begins with task definition. A Tasklet Cell is meaningful only if its capability boundary is defined narrowly enough to be packaged, verified, and evaluated. For that reason, dataset design and task definition are inseparable in this architecture.

A valid task definition should specify, at minimum:

- the task role to be performed, such as routing, classification, extraction, ranking, or a bounded diagnostic function;
- the expected input structure and output structure;
- the acceptance conditions for a correct or acceptable result;
- the boundary between deterministic handling and learned inference;
- and the conditions under which the task should reject, abstain, or defer rather than guess.

The dataset should then be constructed to match that boundary. In the strongest form of the architecture, the dataset is not merely a pile of examples. It is a structured task corpus that reflects the declared input/output contract and the intended decision surface of the Cell. This may include:

- positive examples of expected task behavior;
- negative or adversarial examples;
- edge cases;
- abstention or rejection cases;
- and examples that exercise the boundary conditions at which deterministic logic and learned components meet.

This requirement matters because the Tasklet Cell model is contract-driven. If the task boundary is vague, the data will be vague, and the resulting artifact will either overreach or fail to conform to its declared role. By contrast, a well-defined task corpus makes it possible to align dataset structure, exported model behavior, verifier material, and final runtime expectations.

10.2 Training Scope and Boundary Conditions

Once the task is defined, the next question is what exactly is being trained and what is not. The architecture does not assume that “the model” is the entire solution. Instead, the training scope is bounded by the hybrid pipeline defined earlier in the paper.

In many Cells, only a narrow portion of the capability needs to be learned. Deterministic tools may already cover schema validation, rule-based normalization, filtering, gating, and explicit host-side safety constraints. The learned part may therefore be limited to a narrower role such as:

- selecting among bounded routing paths;
- classifying among a declared set of labels;
- supporting structured extraction;
- or ranking a finite candidate set.

Defining those boundary conditions before training is important. It prevents the training process from drifting into a broader capability than the artifact is supposed to carry. It also makes verifier generation more meaningful, because the verifier can later test the exported capability against the actual intended role rather than against an inflated or ambiguous objective.

The architecture therefore encourages a disciplined training boundary:

- deterministic tasks should remain deterministic;
- learned tasks should be narrow, declared, and role-specific;
- abstention or rejection conditions should be specified rather than treated as accidents;
- and the deployment artifact should not inherit open-ended behavior simply because training was allowed to become unconstrained.

This is also the point at which training-time flexibility must be distinguished from deployment-time flexibility. During training, data may be curated, labels may be refined, thresholds may be tuned, and multiple candidate models may be compared. None of that implies that the deployed Cell may continue to mutate freely afterward. The training boundary is where experimentation stops and artifact formation begins.

10.3 Export and Quantization

After training or capability assembly, the result must be exported into a form suitable for bounded local execution. In the Tasklet Cell model, export is not a trivial conversion step. It is the point at which a trainable or exploratory representation becomes part of a deployable artifact.

The export process should therefore make explicit:

- the final model or logic representation;
- the inference or execution format expected by the Cell;
- the stable metadata needed to interpret that representation;
- and any numerical constraints needed to preserve boundedness and reproducibility.

Where the profile includes a learned model, export may also include quantization or other bounded-representation steps. The reason is straightforward: a deployable Cell should prefer forms of inference that are compact, local, and operationally stable. Quantization is therefore not just a performance trick. It is one way to make the model more consistent with the architecture's bounded execution posture. Prior work on integer-oriented quantization shows that such representations can reduce model size and improve latency for on-device execution, making constrained local deployment more practical (Jacob et al., 2018). In practice, export should make the numerical profile explicit enough that the deployed representation can be interpreted later, including the representation format, any quantization mode, and the stable metadata needed to reproduce or verify the exported payload.

Export should also preserve a clear boundary between the logical payload and its supporting metadata. The model or compiled logic is one artifact component; the manifest, contracts, hashes, verifier material, and profile declarations are others. The bundle format described earlier in the paper depends on this separation.

A useful way to think about export is that it freezes the deployable capability surface. Before export, the system is still a trainable or editable candidate. After export, it becomes the basis of a local artifact that must survive validation, verification, and bounded execution.

10.4 Goldens and Verifier Generation

A Tasklet Cell is not complete when a model has been exported. It becomes a proper artifact only when verifier material has been generated alongside the capability payload.

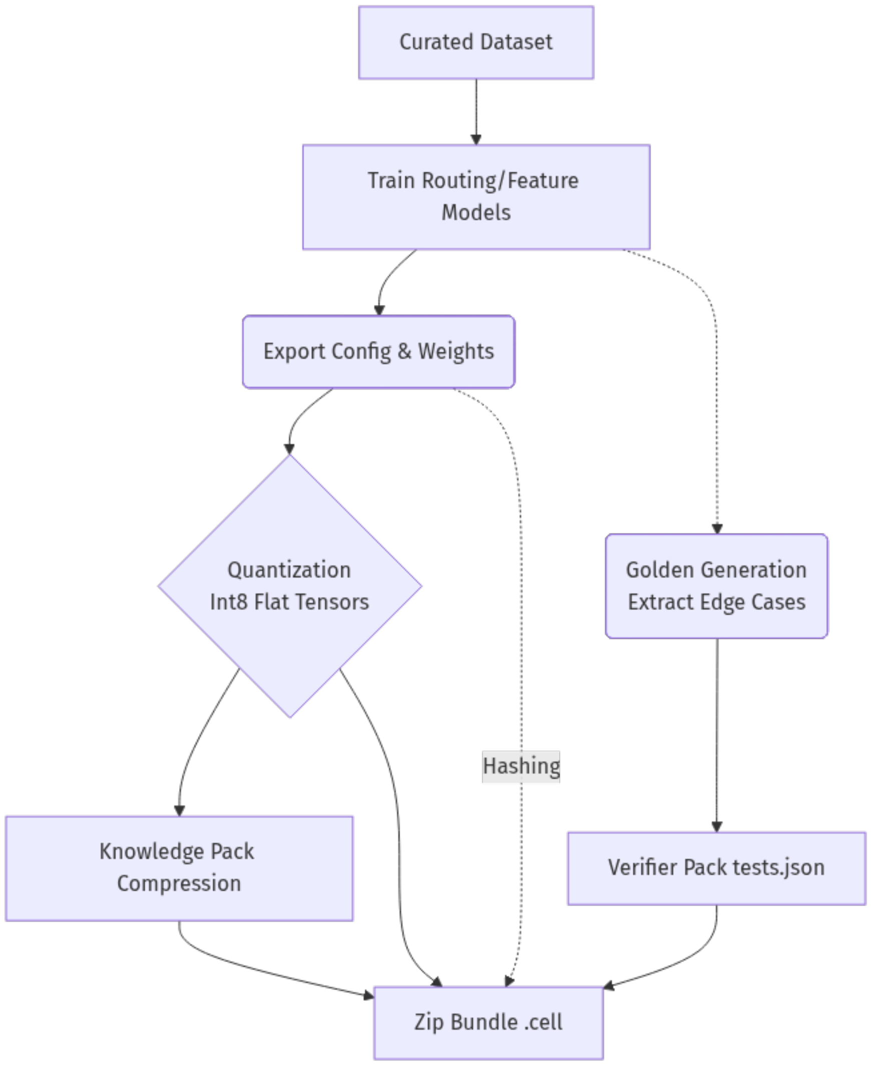
This means that the lifecycle must produce not only the deployable logic, but also a golden set and the associated Verifier Pack content required for local validity checks. In practical terms, this may include:

- canonical task examples with expected outputs;
- schema-conformance examples;
- rejection or abstention cases;
- comparison rules such as exact match, canonical JSON equivalence, numeric tolerance, or other profile-declared comparison modes;
- and profile-specific cases that reflect local policy or execution assumptions.

The golden set should not be understood as a generic test corpus bolted onto the artifact at the end. It is the bridge between the development lifecycle and runtime validity. The verifier material says, in effect, "this is the bounded behavior that must still hold when the artifact is evaluated here, under this Runner and this policy."

This is one of the architecture's key distinctions from ordinary deployment pipelines. In many systems, training ends with model export, and testing remains external to the deployed object. In the Tasklet Cell model, verifier generation is part of artifact formation. The local validity of the artifact depends on carrying enough conformance material to test the exported capability in deployment. Figure 4 summarizes this lifecycle.

Figure 4
Tasklet Cell Training Lifecycle


Curated Dataset
Train Routing/Feature Models
Export Config & Weights
Quantization Int8 Flat Tensors
Golden Generation Extract Edge Cases
Hashing
Knowledge Pack Compression
Verifier Pack tests.json
Zip Bundle .cell

10.5 Packaging and Immutable Deployment Boundary

After export and verifier generation, the capability is packaged into the .cell artifact. This step defines the immutable deployment boundary of the core Cell.

Packaging should bring together:

- the logic payload;
- the manifest;
- the contracts and schemas;
- the Verifier Pack;
- integrity metadata;
- and any optional profile attachments such as model assets, provenance data, SBOM material, attestations, or license notices.

At this stage, the most important architectural transition occurs: the capability stops being a trainable candidate and becomes a bounded software artifact. The core Cell is now expected to be stable enough for hashing, verification, replay, and policy-governed execution.

This does not mean that all adaptation is forbidden forever. The architecture still allows a Personal Layer for bounded local memory and reversible personalization. What it does mean is that the core artifact should no longer drift silently. The immutable deployment boundary is what makes it possible to reason about the Cell's identity, integrity, and conformance under repeated execution.

In that sense, packaging is not merely archival. It is the formal creation of the deployable object to which the rest of the paper's claims apply. In the current evidence stack, the packaged-lifecycle claim is later grounded by the documented CellPOS artifact and acceptance surface in Appendices F and G and interpreted within the broader reporting boundary in Section 14.9.

10.6 Reproducibility Metadata

The final part of the lifecycle is reproducibility metadata. If the architecture claims that a Cell is verifier-backed, replayable, and evidence-evaluable, then the production of the artifact must leave behind enough structured metadata to support those claims.

At minimum, reproducibility metadata should capture:

- artifact identity and version;
- model or logic version;
- training or assembly configuration identifiers;
- data snapshot or corpus identifiers;
- export and quantization configuration;
- verifier and golden-set versioning;
- manifest and bundle hashes;
- build or commit identifiers;
- Runner or execution-environment versioning;
- reproduction entry points or commands where available;
- and any policy-relevant profile declarations needed to interpret execution later.

The exact metadata surface may vary by profile, but the architectural intent is clear: a Cell should not be an opaque product of an undocumented pipeline. It should be possible to connect the runtime artifact to the lifecycle that produced it strongly enough to support replay, evidence generation, debugging, and bounded trust.

At the same time, reproducibility claims should be stated with care. The Reproducible Builds project defines a build as reproducible when the same source, build environment, and build instructions yield bit-for-bit identical specified artifacts, and its documentation explains that deterministic build systems avoid hidden variation from timestamps, unstable ordering, and other environmental factors (Reproducible Builds Project, n.d.-a, n.d.-b). This paper adopts that framing as a methodological reference point, not as an unstated claim that every Cell build is already bit-for-bit reproducible. As discussed later in Section 14.9, the current evidence distinguishes output-level determinism and replayability from stronger artifact-level reproducibility claims.

This requirement does not imply that every training detail must be embedded inside the runtime artifact itself. Rather, it means that the lifecycle should leave behind enough machine-readable linkage between the deployable bundle and its originating process so that the artifact can later be interpreted, compared, or reproduced under controlled conditions.

Taken together, these six stages define the lifecycle discipline of the Tasklet Cell architecture. Dataset and task definition establish the capability boundary. Training scope determines what is learned and what remains deterministic. Export and quantization produce a bounded payload. Golden generation creates the local validity surface. Packaging creates the immutable deployment boundary. Reproducibility metadata ties the artifact back to the process that created it. That full chain is what allows the architecture to claim not just bounded local execution, but bounded local capability with a traceable lifecycle.

11. Bounded Local Memory and Personalization

A useful local artifact often needs some memory of prior interactions, preferences, or corrections. At the same time, the Tasklet Cell architecture depends on a stable artifact boundary that can be validated, replayed, and trusted under explicit rules. The challenge, therefore, is not whether local memory should exist at all. The challenge is how to support local memory and personalization without allowing the core Cell to drift into an uncontrolled learner.

This section defines the architecture's answer to that problem. Memory is allowed, but it must remain bounded, local, policy-gated, and separable from the immutable core artifact. Personalization is allowed, but it must remain reversible, attributable, and constrained by explicit controls. The design aim is to preserve the practical benefits of remembered context while preventing hidden model mutation, silent privilege expansion, or irreversible behavioral drift. In privacy terms, the goal is to reduce exposure surface by keeping memory local and retention-limited, while recognizing that local storage still requires explicit security, retention, and user-control discipline (Cavoukian, 2011; National Institute of Standards and Technology, 2020a).

11.1 Session Memory

The first and most limited memory form is session memory. Session memory is scoped to the current interaction or host-session context. It exists to support continuity within a bounded execution window without changing the long-term identity of the Cell.

Typical session memory may include:

- prior validated inputs within the current task flow;
- intermediate structured state for multi-step local processing;
- short-lived preferences or user selections relevant only to the active session;
- and bounded working context required to complete the declared task.

The key properties of session memory are locality, boundedness, and disposability. It should be tied to the current execution or host session, bounded in size and structure, and safe to discard without compromising the integrity of the core Cell. Session memory is useful because many product workflows are not strictly single-shot. A user may import, correct, retry, and confirm within one bounded interaction. Session memory allows the artifact to preserve continuity across that local sequence without pretending that the Cell has acquired durable knowledge.

For that reason, session memory should not be treated as a hidden adaptation channel. It is an execution-time aid, not a mechanism for silently rewriting the artifact's long-term behavior.

11.2 Episodic Memory

The second form is episodic memory. Episodic memory persists longer than a single session and stores bounded records of prior, local, task-relevant events. Its role is to support practical continuity across repeated use while remaining far narrower than open-ended long-term learning.

Episodic memory may include:

- prior accepted corrections;
- approved aliases or mappings;
- previously observed local preferences;
- records of past artifact decisions that are useful for replay or explanation;
- and compact histories that allow the system to recall what happened in similar recent cases.

The architecture treats episodic memory differently from the immutable core artifact. It is persistent, but it is not part of the Cell's core executable identity in the same sense as the manifest, Verifier Pack, or logic payload. This distinction is essential. It allows the system to remember useful local facts without smuggling hidden adaptation into the core artifact. Because episodic memory may contain user-specific corrections, preferences, or histories, it must also be handled as a privacy-sensitive local store rather than as an informal cache.

Episodic memory must also remain bounded. The paper does not assume an unbounded personal data store or an ever-growing long-term agent memory. Instead, episodic memory should be constrained by explicit storage limits, retention policy, schema discipline, and replay-friendly structure. If memory cannot be bounded, it becomes difficult to reason about conformance, privacy, and rollback. This posture is consistent with privacy-by-design and privacy-risk-management approaches that emphasize default protection, lifecycle handling, and user-respecting controls rather than uncontrolled accumulation (Cavoukian, 2011; National Institute of Standards and Technology, 2020a).

11.3 Gated Personal Layer

The architecture resolves the tension between stability and adaptation through the Personal Layer. The Personal Layer is the explicit boundary at which bounded local personalization may occur. It is separate from the immutable core Cell and exists only to hold local, policy-governed adaptation state.

This separation matters because the paper's central claims depend on a clear distinction among three things:

- the immutable core artifact;
- transient execution context;
- and durable but bounded personalization state.

The Personal Layer may contain approved examples, local preferences, ranking adjustments, threshold shifts, alias maps, or other bounded per-user or per-site adaptation state. However, it is not an unrestricted writable extension of the Cell. It remains subject to four rules.

First, it is local. The Personal Layer is not a hidden cloud-learning channel.

Second, it is bounded. It must remain within explicit schema, storage, retention, and policy limits.

Third, it is gated. Changes to the Personal Layer must pass through a controlled update path rather than being silently absorbed at runtime.

Fourth, it is separate from core validity. The immutable core artifact is still the thing that is hashed, verified, and validated by the Runner. The Personal Layer does not replace that boundary.

This design makes bounded personalization possible without pretending that the artifact itself is constantly rewriting its own identity. The Cell remains the deployable capability artifact. The Personal Layer is an explicit local adaptation envelope around it.

11.4 Reversible Updates and Rollback

Because the Personal Layer may change over time, the architecture requires that those changes remain reversible. Reversibility is not only a usability feature. It is also an accountability and trust feature.

A local personalization update should therefore be:

- attributable to a user action, approval event, or declared policy rule;
- representable as a bounded change against prior local state;
- inspectable after the fact;
- and removable or reversible without rebuilding the core artifact.

Rollback is essential because local adaptation is not always beneficial. A threshold-calibration change may overfit. An alias map may become stale. An approved example may later be recognized as an error. More broadly, continual-learning research shows that new updates can interfere with previously useful behavior, a failure mode commonly described as catastrophic forgetting (Kirkpatrick et al., 2017). The Tasklet Cell architecture does not claim to solve continual learning in general. It does take that risk seriously enough to require rollback and bounded regression checks for local personalization updates.

For that reason, the Personal Layer should support explicit rollback semantics. At a minimum, this means that changes should be versionable or state-differentiable enough to allow deletion, reset, or reversion to an earlier local state. In stronger profiles, rollback may also support selective forgetting, scoped deletion, or policy-driven expiration.

This reversibility requirement also reinforces the paper's distinction between the artifact and the adaptation around it. If personalization can be reversed without rebuilding the core Cell, then the architecture has preserved the separation it claims to preserve. As discussed later in Section 15.3 and Appendix D, the current evidence stack is consistent with this model: local bounded memory is product-useful, but learning-layer changes are still expected to remain inspectable and reversible rather than silently self-authorizing.

11.5 Learning Gate and User Control

The mechanism that keeps personalization bounded is the learning gate. In this paper, "learning" does not imply unconstrained online training of the core artifact. It means a controlled local update path through which bounded personalization state may be proposed, validated, accepted, rejected, or rolled back. In the intended profile, this is closer to bounded post-hoc calibration, approved-example storage, or controlled alias and preference updates than to free-form retraining of the core model.

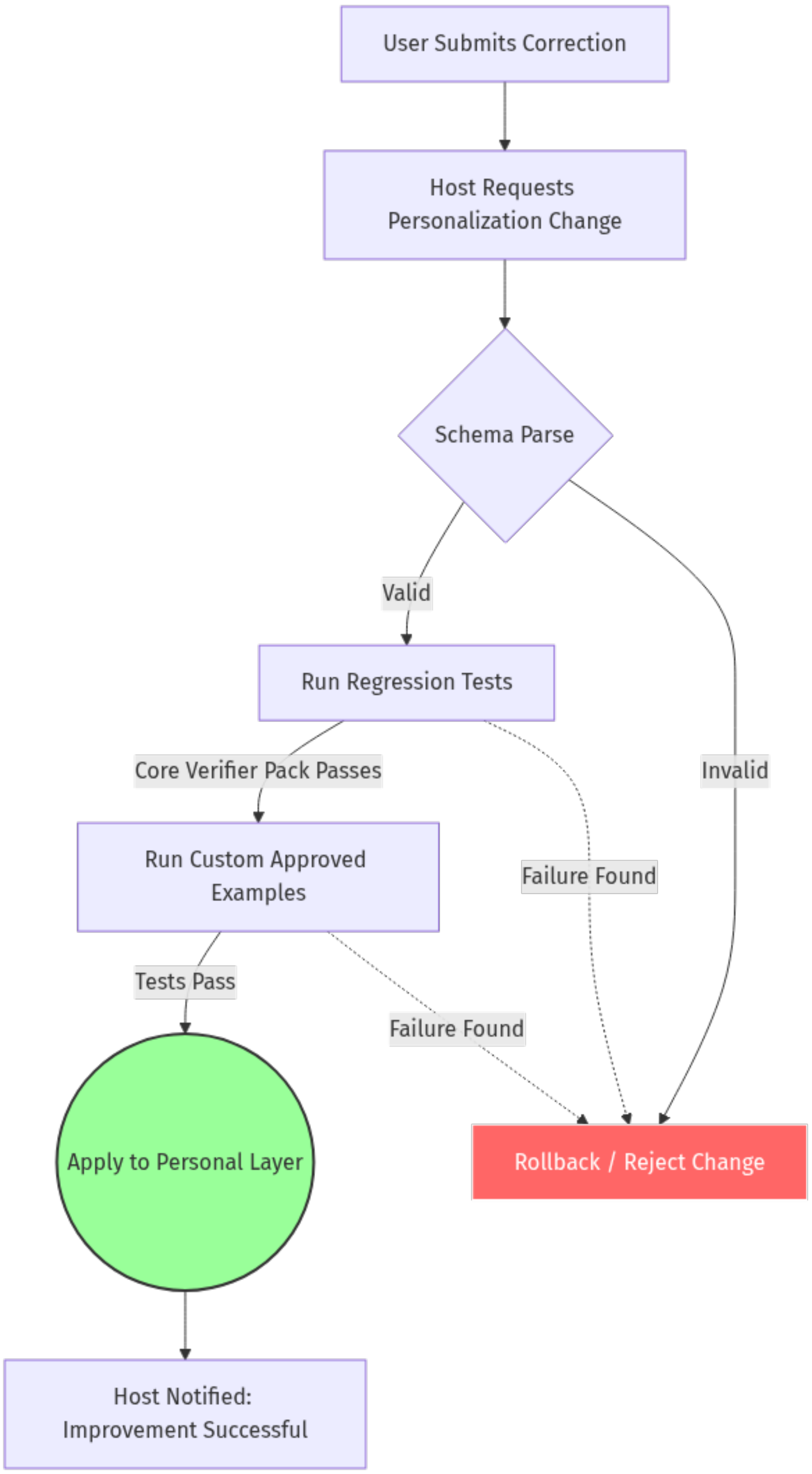
A learning gate should include, at a minimum, the following stages:

1. *Input and schema validation.* The proposed personalization change must be well formed and must fit the declared schema for the Personal Layer.
2. *Policy evaluation.* The system must determine whether this class of update is allowed under the local policy profile. Some deployments may allow threshold calibration but not alias additions; others may allow approved examples but only under user confirmation.
3. *Boundary and regression checks.* The update must remain within declared storage, scope, and behavior limits. Where appropriate, local regression or conformance checks should be used to ensure that the change does not immediately violate the bounded behavior expected of the artifact.
4. *Commit or refusal.* If the proposal passes the gate, the update may be committed to the Personal Layer. If it fails, the system must reject it cleanly rather than partially absorbing it.
5. *Reversal and user control.* The user or operator must retain meaningful control over accepted personalization state, including the ability to inspect, edit, reset, delete, or selectively forget it according to the deployment profile.

This gate matters because it is what distinguishes bounded personalization from silent drift. Without it, the architecture would gradually lose the very separation between immutable artifact and local adaptation on which its trust model depends. In privacy terms, this also keeps personalization closer to user-centric and risk-managed local state than to opaque behavioral profiling (Cavoukian, 2011; National Institute of Standards and Technology, 2020a).

Seen as a whole, bounded local memory and personalization are not exceptions to the Tasklet Cell model. They are controlled extensions of it. Session memory supports continuity within a bounded execution context. Episodic memory supports useful local recall across interactions. The Personal Layer provides a bounded local adaptation boundary. Reversible updates and rollback preserve trust, accountability, and user recourse. The learning gate ensures that personalization remains local, privacy-aware, inspectable, and under control. Together, these mechanisms allow a Cell to become more useful in context without ceasing to be a bounded, verifier-backed artifact. Figure 5 summarizes this gated update path.

Figure 5
Learning Gate Flow


User Submits Correction
Host Requests Personalization Change
Schema Parse
Valid
Invalid
Run Regression Tests
Core Verifier Pack Passes
Failure Found
Run Custom Approved Examples
Tests Pass
Failure Found
Apply to Personal Layer
Rollback / Reject Change
Host Notified: Improvement Successful

12. Observability, Replay, and Debuggability

A bounded artifact architecture is useful in practice only if its behavior can be inspected, reproduced, and debugged without collapsing back into cloud telemetry or uncontrolled host introspection. For that reason, Tasklet Cells are defined not only by bundle structure, verification, and bounded execution, but also by what the system can observe about itself locally and how that information can be used to reproduce and understand behavior after the fact.

This section defines the architecture's observability model. The central constraint is that observability must remain consistent with the same principles that govern the rest of the system: local-first operation, explicit boundaries, bounded storage, fail-closed behavior, and privacy-aware handling of sensitive data. The goal is not to turn a Tasklet Cell deployment into a miniature observability platform. It is to provide enough local evidence to support replay, debugging, audit, and regression testing without introducing hidden telemetry or a new path of ambient authority.

12.1 Schema-Stable Local Logs

The first observability primitive is the schema-stable local log. A schema-stable local log is a bounded, structured record of what occurred at the artifact boundary and at the Runner boundary during validation, execution, and refusal. The emphasis here is deliberate: the architecture does not require bit-identical logs across all environments. It requires logs that are stable enough in structure and meaning to support replay, debugging, and audit under controlled conditions.

Structured local logging is therefore treated as a first-class assurance surface. Logs should be collected, protected, and retained with explicit integrity and retention considerations, consistent with established log-management guidance (Kent & Souppaya, 2006). A log that changes shape unpredictably, depends on incidental host noise, or mixes structured state with free-form debug chatter is far less useful than a smaller but disciplined record.

At a minimum, schema-stable local logs should favor:

- stable field structure;
- explicit event classes;
- declared artifact and Runner identifiers;
- policy and enforcement context;
- canonicalized or otherwise stable representations for any structured values that will later be compared or hashed;
- and bounded payload content.

Typical loggable events may include:

- bundle ingestion and structural validation outcomes;
- integrity-check outcomes;
- Verifier Pack pass or failure results;
- execution admission or refusal;
- policy denial and resource-bound violations;
- schema validation failures;
- output acceptance or rejection;
- and personalization-gate decisions when a Personal Layer exists.

Schema-stable logging does not mean that every field must be globally immutable. It means that the architecture should prefer a stable schema over free-form text and should avoid unnecessary nondeterminism in the fields that matter for comparison, replay, and audit. In particular, logs should not rely on hidden contextual state or arbitrary formatting choices that make the same event difficult to compare across runs.

12.2 Redaction and Retention Bounds

Observability introduces a second problem: once the system records information locally, it must decide how much of that information should be stored, how long it should remain, and how sensitive content is protected. For this reason, the Tasklet Cell architecture treats redaction and retention as part of the observability model rather than as a downstream compliance afterthought.

The default posture should be redaction by default. The system should prefer to log identifiers, schemas, outcome classes, enforcement decisions, and replay references rather than raw sensitive payloads. Full content capture may be useful in some debugging profiles, but it should not be the default operating mode.

This redaction principle serves several purposes.

First, it reduces the privacy risk of local logs. A system that records raw user content, raw business documents, or unrestricted host context by default can easily undermine the architecture's privacy and boundedness claims. Privacy-by-design principles and privacy-risk-management frameworks both support the idea that systems should protect sensitive information through default settings, lifecycle thinking, and explicit control rather than through unrestricted accumulation (Cavoukian, 2011; National Institute of Standards and Technology, 2020a).

Second, it preserves replay discipline. Replay often does not require every raw intermediate value to be retained indefinitely; it requires enough structured context to reconstruct the same decision path under controlled conditions.

Third, it keeps observability bounded. If logs are allowed to become an unbounded secondary data store, the architecture quietly reintroduces the operational and privacy risks it claims to avoid.

Retention must therefore also be explicit. The architecture should support bounded retention through configurable time-to-live, size limits, scope limits, or profile-based storage policies. Stronger deployment profiles may additionally support selective deletion, log compaction, redaction upgrades after storage, and explicit protection of audit information against casual modification or overexposure. The important point is that observability data must remain privacy-aware, bounded, and locally protected.

12.3 Replay Records

The second core observability primitive is the replay record. A replay record is the structured local artifact that allows a validation decision or execution outcome to be reproduced later under the same artifact and policy boundary.

The replay record is narrower than a full execution transcript. Its role is not to preserve every possible internal detail of runtime behavior. Its role is to preserve enough structured information to answer the practical systems question: can this decision be reproduced locally under the same artifact, Runner, and policy conditions?

At a minimum, a replay record should capture:

- the Cell identity and version;
- the bundle hash or other stable artifact identifier;

- the Runner version;
- the policy profile or policy hash;
- the validated input payload or a redaction-safe reference or canonicalized hash of it;
- the execution or refusal outcome;
- any comparison or verification mode required to re-evaluate the case;
- and, where relevant, the specific Verifier Pack or enforcement-report version associated with the original event.

In some profiles, a replay record may also include references to the specific Verifier Pack version, Personal Layer state version, or enforcement-report identifier associated with the original event. The exact replay surface may vary, but the architectural goal is constant: replay should be local, structured, and bounded.

This requirement matters because debugging without replay is weak debugging. If the system cannot connect a runtime outcome back to a stable local record of what artifact was run, under what policy, and with what validated input, then many of the paper's claims about boundedness and auditability become much harder to sustain in practice.

12.4 Conformance Replay for Regression Testing

Replay is not only a debugging tool. It is also a conformance tool. The architecture therefore supports conformance replay, meaning the reuse of local replay material to test whether artifact behavior remains acceptable across controlled changes in the local environment, Runner, or artifact profile.

Conformance replay is especially important in an architecture where local validity depends on the Verifier Pack and where execution is bounded by policy. A Cell may be structurally valid in general and yet fail under a new policy profile, a changed Runner version, or a modified local comparison mode. Replay makes those changes visible.

At a practical level, conformance replay should allow the system to:

- rerun verifier material against the same artifact boundary;
- compare new outcomes against prior accepted outcomes under declared comparison rules;
- detect regressions in schema conformance, rejection behavior, or output acceptance;
- and do so without requiring cloud coordination or remote test infrastructure.

This makes replay an extension of the architecture's evidence posture. The system is not merely logging that a decision happened. It is preserving enough local structure to ask whether the same artifact and the same environment still behave in the same acceptable way. That matters both for development and for controlled post-deployment maintenance. As discussed later in Section 14.9 and Appendix C, the current manuscript also distinguishes replayable outputs from stronger bit-for-bit artifact reproducibility claims.

12.5 Offline Debugging Boundaries

The final part of the observability model is the definition of offline debugging boundaries. A Tasklet Cell system should be debuggable, but it should not become debuggable by abandoning the very boundaries that make it safe and local in the first place.

For that reason, debugging in this architecture must remain subject to the same overall rules that govern execution:

- no ambient cloud telemetry as a requirement;

- no unconstrained expansion of authority for the sake of debugging;
- bounded and policy-aware local records;
- and explicit distinction between core artifact state, Personal Layer state, and host-provided context.

This means that a debugging profile may enrich local visibility, but it should do so through explicit local controls rather than hidden side channels. For example, a stronger local debug mode may permit more verbose structured records, more detailed enforcement reports, or more complete replay material. However, it should still remain local, inspectable, and bounded, and it must not create a bypass around the same validation, policy, and capability controls that govern normal execution.

Offline debugging boundaries also matter for trust. If the only way to understand behavior is to export runtime data to a remote service, then the architecture has failed to preserve the locality it claims to value. If the only way to reproduce a decision is to grant broad host access to the debugging layer, then the architecture has undermined its own capability boundaries.

The correct design target is therefore not maximal introspection. It is sufficient local introspection under explicit control. A Tasklet Cell should be understandable enough to debug and reproduce within its deployment boundary, but not so open-ended that debugging becomes a back door for hidden execution paths or privacy leakage.

Taken together, schema-stable local logs, bounded redaction and retention, replay records, conformance replay, and explicit offline debugging boundaries define the observability model of the architecture. They ensure that bounded local capability is not only executable, but inspectable; not only verifier-backed, but reproducible in the operational sense intended here; and not only safe in principle, but understandable in practice.

13. Security and Trust Model

The Tasklet Cell architecture is not secure merely because it is local, compact, or packaged. Local execution can reduce some classes of risk, but it does not remove the need for an explicit trust model. If a host is expected to load an artifact, grant it bounded authority, and rely on its outputs inside a product workflow, then the system must specify what is trusted, what is untrusted, what is verified, and what happens when verification fails.

This section defines that model. The central claim is deliberately narrow: trust in a Tasklet Cell does not arise from origin alone, from host proximity, or from the mere existence of integrity metadata. Trust is mediated through the local Runner boundary. The artifact must satisfy integrity and conformance checks, the Verifier Pack must pass under the target policy, and all execution authority must remain bounded and explicitly granted. Anything outside that local mediation path remains untrusted until proven otherwise.

13.1 Artifact Integrity and Trust Anchors

The first layer of the trust model is artifact integrity. A Tasklet Cell must carry enough integrity metadata for the Runner to determine whether the artifact presented for execution is the artifact that was intended. At a minimum, this requires stable identifiers for the bundle and its required components, together with a way to detect tampering, mismatch, or structural ambiguity before execution begins.

In practical terms, integrity material may include:

- content hashes for required bundle components;
- canonicalized metadata used for hashing or signing;

- optional signatures or attestations, depending on profile;
- provenance records or inventory material that inform local trust decisions;
- and profile declarations that tell the Runner how integrity should be interpreted under policy.

However, integrity metadata alone is not enough. A cryptographically authentic artifact can still be invalid under the local policy profile, and a provenance-rich artifact can still be rejected if its Verifier Pack fails or if its requested capability surface is unacceptable in the target environment. This is why the architecture distinguishes between authenticity and local validity. Software-update and supply-chain frameworks such as TUF and in-toto are useful reference points here: they show why authenticity, metadata integrity, role separation, and supply-chain verification matter, but they do not eliminate the need for a local policy decision about whether a given artifact is acceptable for execution in a particular context (Samuel et al., 2010; Torres-Arias et al., 2019).

The trust anchors for artifact acceptance must therefore be explicit. Where signature verification is used, the roots of trust must come from locally trusted policy material rather than from arbitrary bundle-provided hints. Where provenance is considered, it must be evaluated as supporting trust context rather than as a replacement for local validation. The artifact may carry references, signatures, or attestations, but the Runner must still decide what those materials mean under the active local trust policy.

This posture is also consistent with broader cybersecurity supply-chain risk-management guidance, which treats supply-chain security as an organizational discipline involving policies, plans, risk assessments, and trust decisions rather than as a property that can be delegated entirely to artifact metadata (National Institute of Standards and Technology, 2024). The consequence is important: the artifact may describe itself, but it does not decide its own trust status. Trust anchors live outside the artifact's self-description and are interpreted by the local enforcement boundary.

13.2 Verifier-Mediated Validity

The second layer of the trust model is verifier-mediated validity. In this architecture, a Cell is not valid because it exists, because it is signed, or because it came from a nominally trusted source. It becomes valid only when its Verifier Pack passes under the target Runner and target policy.

That rule has several implications.

First, the Verifier Pack is a runtime trust mechanism, not merely a development-time test bundle. Its purpose is to establish that the artifact's declared behavior is locally acceptable under the profile in which it is about to run.

Second, validity is contextual. The same Cell may be valid under one Runner configuration and invalid under another, depending on policy, profile, comparison mode, capability surface, or execution constraints. Local validity is therefore not a universal property of the artifact in the abstract. It is a property of the artifact under mediation.

Third, verification failure must map to non-execution. An artifact that fails verifier-mediated validity is not partially trusted and must not be allowed to proceed into degraded or best-effort execution merely because it appears close enough.

This architecture uses verifier-mediated validity as the point at which trust becomes operational. Before that point, the artifact may be well formed, authentic, or even familiar. After that point, it is either valid for execution under the local boundary or it is not. There is no intermediate category in which unclear validity is tolerated for convenience. In systems terms, this resembles the general policy-decision and policy-enforcement separation described in Zero Trust Architecture: trust becomes meaningful only when policy is interpreted and enforced at the decision boundary, not when a resource or artifact simply presents self-descriptions or network proximity (National Institute of Standards and Technology, 2020b).

13.3 Threat Model

The threat model addressed by this paper is bounded and practical. It is not a claim to eliminate every possible security risk. It is a claim to reduce and mediate a specific set of risks at the artifact and execution boundary.

The model is designed to address threats such as:

- artifact tampering or mismatch between declared and actual contents;
- supplier, provenance, or attachment mismatch at the bundle boundary;
- structurally unsafe or ambiguous bundles;
- invalid or adversarial input that attempts to cross contract boundaries;
- unauthorized capability use, including unintended tool or host access;
- policy mismatch between artifact assumptions and local execution rules;
- excessive resource consumption or runtime expansion beyond declared limits;
- hidden drift through uncontrolled personalization or adaptation;
- and indirect trust escalation through external orchestrators or helper layers.

The architecture is not designed to solve every upstream or downstream problem. It does not claim to eliminate all model errors. It does not claim to make all local code safe simply by packaging it. It does not claim to defend against every host compromise, every supply-chain threat, or every malicious local operator. Instead, it narrows the trust surface at the point where a bounded artifact is admitted and executed.

This narrowing is achieved through several mechanisms working together: structural validation, integrity checks, verifier-mediated validity, bounded capability grants, resource controls, fail-closed behavior, and replay-oriented observability. The security posture therefore depends less on a single perfect check than on the fact that multiple boundaries align at the same local decision point. At the same time, the model does not assume that the underlying runtime substrate is flawless; the architecture narrows trust at the artifact boundary, but it does not claim to eliminate all engine, compiler, or host-environment defects. Table 3 maps the main threat classes to their mitigation layers.

Table 3
Threats and Mitigations Mapping

Threat vector	**Mitigation mechanism**	**Enforcement layer**
Artifact tampering / content mismatch	Required integrity metadata, content hashes, and pre-execution hash checking	Runner / local integrity boundary
Structurally unsafe or ambiguous bundles	Stable bundle layout; rejection of duplicate normalized paths, path traversal, unsafe decompression behavior, and archive ambiguity	Runner / bundle parser
Invalid or adversarial input crossing contract boundaries	Explicit input/output schemas and contract validation before execution	Runner (schema and contract validation)

Unauthorized capability use / unintended host or tool access	Policy-bound capability controls, explicit capability declarations, allowlists, and no ambient authority	Runner + Gateway
Policy mismatch between artifact assumptions and local rules	Validity is contextual under the target Runner and local policy; mismatch maps to non-execution	Runner
Excessive resource consumption / runtime expansion	Declared runtime limits such as wall time, steps, memory, output size, and fail-closed rejection on bound violations	Runner
Hidden drift through uncontrolled personalization	Learning-gate mediation, reversible updates, and regression-style checks before apply	Runner + Learning Gate
Indirect trust escalation through orchestrators or helper layers	External orchestrator remains untrusted; Gateway may enforce schema, allowlist, size, and rate limits but cannot bypass Runner checks	Gateway + Runner
Self-authorizing update / signature trust roots	Locally pinned trust anchors govern verification; manifest-supplied trust references are metadata only	Local trust policy + Runner

Note. Threat classes and mitigation points in the Tasklet Cell trust model. The security posture relies on layered local enforcement including integrity checks, verifier-mediated validity, capability controls, resource bounds, fail-closed behavior, and non-bypass mediation rather than on a single trust primitive.

13.4 Untrusted Orchestrator and Gateway Limits

A particularly important trust boundary in this architecture concerns external orchestrators and optional Gateway Profiles. Many real systems contain coordination logic outside the Cell itself: schedulers, workflow managers, desktop shells, or other components that decide when to invoke a capability. The architecture does not deny that such components may exist. It denies that they are automatically trusted.

In this model, an orchestrator remains untrusted until its requests are mediated through the local Runner and, where present, the Gateway Profile. This means:

- an orchestrator cannot confer validity on a Cell by requesting execution;
- a Gateway cannot widen the Cell's authority boundary through convenience;
- and no request path is allowed to bypass the same local validation, policy, and capability controls applied to direct host execution.

The Gateway Profile exists to keep those limits explicit. It may validate request schemas, apply allowlists, enforce rate or size limits, and preserve offline-first policy. What it must not do is turn external coordination into an alternate trust root. In policy terms, the Gateway can be understood as a local enforcement point, but it does not replace the Runner's final execution-validity decision (National Institute of Standards and Technology, 2020b).

This is one of the main ways in which the paper departs from many loosely bounded agent frameworks. Coordination is allowed. Unmediated trust transfer is not. The orchestrator may request work, but it does not get to define the trust boundary. The Runner still does that. As discussed later in Section 14.4 and Appendix D, the current evidence is consistent with this claim: malformed or adversarial bundles are rejected fail-closed at the local boundary rather than being treated as degraded but runnable inputs.

13.5 Manual-Update Trust Correction

The final part of the trust model is the manual-update trust correction. This point is easy to get wrong in artifact systems and therefore deserves explicit treatment.

If an artifact profile supports signatures, attestations, manual replacement, or policy-managed updates, the local trust anchor must come from the local trust policy, not from the artifact's own declaration of where trust should be found. In other words, manifest-supplied key paths, certificate references, or similar self-descriptions are metadata only. They may help describe how an artifact was produced, but they must not be allowed to define the trust root that validates the artifact.

This distinction matters because an artifact that can declare its own trust anchor is dangerously close to self-authorizing. The trust boundary would collapse if the artifact could say both "verify me" and "here is the authority that should verify me" without local policy mediation. TUF is directly relevant to this point because it treats trusted root metadata and key-rotation logic as distinct, explicit trust material rather than something the target artifact can redefine on its own (Samuel et al., 2010).

The corrected rule is therefore simple: locally trusted pinned keys, or equivalent locally governed trust anchors, must drive verification decisions. Manifest-supplied trust references may be carried as metadata, but they are not binding on the Runner unless the local policy has independently decided to trust them.

This correction strengthens the entire architecture. It keeps artifact authenticity subordinate to local trust policy, reinforces the distinction between metadata and trust anchors, and prevents update or replacement workflows from becoming a subtle bypass around verifier-mediated validity. At the same time, the paper does not claim that this section by itself solves operational signing, revocation, transparency logging, or key-custody management end to end. Those remain adjacent deployment concerns in the broader supply-chain landscape represented by TUF, in-toto, SLSA, and Sigstore and are treated here as future operational hardening rather than present proof (OpenSSF, n.d.; Samuel et al., 2010; Sigstore Project, n.d.; Torres-Arias et al., 2019).

Taken together, these five elements define the security and trust posture of the Tasklet Cell architecture. Artifact integrity establishes that the bundle is what it claims to be. Verifier-mediated validity determines whether it is locally acceptable. The threat model defines which risks are intentionally addressed. Untrusted orchestrator handling prevents coordination layers from becoming alternate trust roots. The manual-update trust correction ensures that trust anchors remain local rather than artifact-defined. Together, these rules make the architecture's trust boundary explicit, bounded, and enforceable.

14. Empirical Evaluation and Benchmarks

We evaluate AGIF Tasklet Cells as an offline, verifiable artifact architecture rather than as a general AGI system. The goal of this section is to establish what the current implementation can support through executable evidence, and just as importantly, what it cannot yet support. All results in this section are grounded in the released paper-evidence bundle for the reference implementation and should be interpreted as evidence for a bounded local Runner/Verifier architecture with a concrete productization path, not as evidence of broad autonomous capability or production deployment at scale.

14.1 Experimental Setup and Evidence Methodology

The canonical evaluation anchor for the present paper state is the clean release evidence bundle generated for commit 78a1635 on 2026-03-09. The bundle was produced locally and offline from a clean detached worktree and is preserved publicly in the AGIF Tasklet Cell public repository release paper-evidence-r1-78a1635-clean, with the in-repo anchor rooted at paper_evidence/2026-03-09/78a1635. The release pack includes env.json, summary.json, summary.md, paper_table.md, reproduce.md, generated figures, and raw suite outputs. The portable bundle supports audit of the reported outputs rather than serving as a self-contained rerun environment; the public release should be read as an inspectable evidence object rather than as a full regeneration harness.

Historical baseline evidence at 533c63e remains useful as failure context, but it is no longer the canonical evaluation anchor. That distinction matters because the paper should not mix stale baseline text with the current clean release state.

Following community norms around research artifacts, the released evidence pack is treated here as the citation anchor for empirical claims. The paper also distinguishes repeatable result outputs from stronger claims of bit-for-bit reproduction of every intermediate build artifact (Association for Computing Machinery, n.d.; Reproducible Builds Project, n.d.-a).

The clean evidence run executed 30 repeated runs of five suites:

1. FINAL_RELEASE_READINESS_SWEEP
2. review_hallucination_bench_v1
3. a6_extractor_benchmark_v1
4. runner_fail_closed_bench_v1
5. reasoning_trace_bench_v1

Across the full bundle, this corresponds to 2400 executed evaluation units:

- 600 release-readiness checks (20 × 30)
- 120 hallucination-safety cases (4 × 30)
- 180 extraction-quality cases (6 × 30)
- 1140 fail-closed negative cases (38 × 30)
- 360 reasoning-trace cases (12 × 30)

The recorded clean execution environment is Apple M4 on macOS 26.3.1 arm64 with Python 3.14.3 and Rust 1.93.1. The run is explicitly offline/local, and the recorded expected runtime is approximately 71.6 minutes. Table 4 summarizes the clean-bundle evaluation coverage and outcomes.

Table 4
Aggregate Evaluation Summary

Suite	Repeats	Per-repeat coverage	Result	Core outcome
FINAL_RELEASE_READINESS_SWEEP	30	20 checks	Pass	30/30 repeats passed; all 20 checks passed in every repeat

review_hallucination_bench_v1	30	4 cases	Pass	unsafe_allow_count = 0, abstain_accuracy = 1.0, reason_accuracy = 1.0
a6_extractor_benchmark_v1	30	6 cases	Pass	routing_accuracy = 1.0, vendor_hit_rate = 1.0, currency_accuracy = 1.0, grand_total_mae = 0.0, tax_total_mae = 0.0, subtotal_mae = 0.0, numeric_grounding_rate = 1.0, abstain_fail_rate = 0.0
runner_fail_closed_bench_v1	30	38 cases	Pass	rejection_rate = 1.0, unsafe_allow_count = 0, expected_reason_match_rate = 1.0
reasoning_trace_bench_v1	30	12 cases	Pass	trace_schema_valid_rate = 1.0, trace_determinism_rate = 1.0, evidence_alignment_rate = 1.0

14.2 Release Readiness and System Integration

The broadest integration signal comes from FINAL_RELEASE_READINESS_SWEEP. Each repeat executes 20 named checks spanning the training/export/runtime chain, determinism and replay, desktop-shell integration, UI and guided demo flow, fail-closed user experience, manual update and rollback behavior, licensing and activation, public release artifacts, and an end-to-end prototype proof.

The named checks are:

- A2 Training pipeline scaffold
- A3 Export and quantization
- A4 Runtime inference
- A5 Determinism and replay
- A6 Accuracy/performance gate
- B1 Desktop shell architecture
- B2 Trigger and gateway wiring
- B3 Extraction and validator screens
- B4 Replay and learning history screens
- B5 Guided demo flow
- B6 Fail-closed UX
- C1 macOS installer signing
- C2 Windows installer signing
- C3 Manual signed updates
- C4 Version compatibility and rollback
- D1 Whitepaper-to-product claims matrix
- D2 Licensing and activation
- D3 Public release artifacts
- D4 Whitepaper alignment evidence matrix
- PHASE6 End-to-end prototype proof

The clean anchor records 30/30 passing repeats, no failed sub-check identifiers, and full sub-check coverage. The suite-level aggregate metrics are determinism_rate = 1.0, latency p50 / p95 / p99 of 130394.58 / 151537.89 / 152406.03 ms, mean peak RSS of 84.00 MB, and zero output-size variance at 12,553 bytes per repeat. For the paper, this is the strongest evidence that the reference stack is not merely a conceptual packaging model, but a product-shaped local system whose training, export, runtime, verification, packaging, and release surfaces have been exercised together under repetition.

This distinction matters because the release-readiness sweep is an integration-level result, not merely a narrow task benchmark. It does not prove field deployment, but it does show that the reference system behaves like a coherent local product stack rather than a disconnected research demo.

14.3 Hallucination Safety and Bounded Abstention

The review_hallucination_bench_v1 suite evaluates whether the system abstains appropriately when claims are insufficiently supported, when evidence is missing, or when evidence conflicts. Each repeat contains 4 cases, yielding 120 total case executions across the clean bundle.

Across all 30 repeats, the suite reports 30/30 passing repeats, unsafe_allow_count = 0, abstain_accuracy = 1.0, reason_accuracy = 1.0, and hallucination_resistance_score = 1.0. The suite-level per-repeat runtime latency p50 / p95 / p99 is 1180.74 / 1189.50 / 1190.35 ms, and mean peak RSS is 30.20 MB. In other words, the implementation does not merely abstain more often; it abstains for the expected reasons under the benchmark's low-support, missing-evidence, and conflicting-evidence cases, while correctly allowing the supported case.

The safe claim is therefore narrow but meaningful: the current implementation demonstrates bounded abstain-or-respond behavior under explicit evidence constraints. It does not support a stronger claim about general truthfulness or open-domain hallucination resistance beyond the evaluated harness.

14.4 Fail-Closed Negative Tests and Verifier Behavior

The most direct evidence for the Runner/Verifier model comes from runner_fail_closed_bench_v1. This suite executes 38 negative cases per repeat, or 1140 total negative executions across the clean evidence pack. These cases target malformed bundles, tampered bundles, invalid metadata, policy violations, rollback scenarios, runtime-limit violations, and related negative conditions at the artifact boundary.

The 38 negative cases span 12 categories: hash tamper (6), schema (7), verifier (3), archive paths (3), limits (4), licensing (2), integrity metadata (3), policy (3), rollback (1), runtime limits (4), runtime policy (1), and runtime input (1).

Across all 30 repeats, the suite reports 30/30 passing repeats, rejection_rate = 1.0, unsafe_allow_count = 0, and expected_reason_match_rate = 1.0. The suite-level per-repeat runtime latency p50 / p95 / p99 is 4728.36 / 4752.99 / 4774.74 ms, and mean peak RSS is 31.10 MB. This is the clearest evidence in the bundle for a fail-closed admission model. The Runner does not merely reject bad inputs generically; it rejects the provided malformed and adversarial cases with the expected failure classes under repeated execution.

The safe publication claim remains bounded: the Runner/Verifier stack demonstrates complete rejection on the provided malformed and adversarial bundle set. It does not prove resistance to every possible malicious input, every future attack class, or every host-compromise scenario.

14.5 Reasoning-Trace Validity and Evidence Alignment

The reasoning_trace_bench_v1 suite evaluates whether the system can emit a trace structure that is schema-valid, repeatable at the evaluated-output level, and aligned to declared evidence fields. Each repeat contains 12 cases, resulting in 360 trace-evaluation cases across the full clean bundle.

Across 30 repeats, the suite reports 30/30 passing repeats, trace_schema_valid_rate = 1.0, trace_determinism_rate = 1.0, and evidence_alignment_rate = 1.0. The suite-level per-repeat runtime latency p50 / p95 / p99 is 37.13 / 37.67 / 37.75 ms, and mean peak RSS is 23.95 MB. The reference trace structure contains four steps per case—budget_check, risk_assessment, policy_gate, and final_decision—and the canonical 12-case final-decision histogram is approve = 4, manual_review = 5, reject = 3. In the evaluated harness, the traces are not just present; they are schema-valid and aligned to the evidence fields named by each step.

This supports a specific architectural claim: a Tasklet Cell can expose a bounded local reasoning trace that is inspectable and machine-checkable without turning the system into an open-ended conversational agent. It does not support broader claims about philosophical explanation faithfulness or full causal interpretability.

14.6 Numeric Extraction Benchmark and Historical Baseline Correction

The clearest historical empirical limitation in the project was the A6 numeric extraction benchmark at baseline 533c63e, where the older bundle reported strong routing, vendor, and currency behavior but unacceptable grand-total extraction quality. In the clean anchor at 78a1635, that historical blocker is resolved.

The clean a6_extractor_benchmark_v1 reports 30/30 passing repeats with routing_accuracy = 1.0, vendor_hit_rate = 1.0, currency_accuracy = 1.0, grand_total_mae = 0.0, tax_total_mae = 0.0, subtotal_mae = 0.0, max_arithmetic_error = 0.0, numeric_grounding_rate = 1.0, and abstain_fail_rate = 0.0. The suite-level per-repeat runtime latency p50 / p95 / p99 is 3516.01 / 3542.79 / 3698.15 ms, and mean peak RSS is 75.33 MB. The raw repeat reports separately record mean per-case row latency of 30.69 ms across repeats.

That is a meaningful change in the paper’s empirical posture and should be reflected honestly. The present paper state should not preserve stale failure language as though it were still canonical.

At the same time, the passing A6 result does not license a universal claim about finance-document extraction. The safe claim remains bounded to the evaluated workload and current clean harness. The broader limitation is now scope rather than the exact historical grand-total failure itself.

14.7 Artifact Footprint and Bounded Package Size

The evidence bundle also records the footprint of the reference cell package, identified as cells/finance_doc_extractor_neural_v6.cell. The clean bundle records the following component sizes:

- bundle archive bytes: 369,595
- total component bytes before compression: 955,161
- dominant component: logic_wasm = 952,014 bytes
- manifest_json = 1,157 bytes
- hashes = 684 bytes
- model_dir = 507 bytes
- external_runtime_model_bin = 256 bytes
- schemas = 232 bytes
- verifier_pack = 191 bytes

- licenses = 120 bytes

This matters because the paper's central systems claim is not only that Cells are executable, but that they are bounded, inspectable, and packageable as local artifacts. The footprint data shows that the reference artifact is still small enough to reason about as a discrete package rather than as an opaque remote service dependency. Table 5 summarizes the footprint breakdown for the reference Cell.

Table 5
Reference Cell Footprint Breakdown

Bundle asset / region	**Measured size**	**Status in reference Cell**	**Purpose**
logic/logic.wasm	952,014 bytes	Present	Executable capability payload; dominant footprint component
manifest.json	1,157 bytes	Present	Identity, limits, contracts, logic/verifier references, integrity metadata
hashes/	684 bytes	Present	Tamper detection before execution
model/	507 bytes	Present	Optional model asset directory for richer profiles
external_runtime_model_bin	256 bytes	Present	External runtime model payload recorded in the clean bundle metadata
schemas/	232 bytes	Present	Machine-checkable input/output boundary
verifier/	191 bytes	Present	Built-in Verifier Pack used to establish local validity
licenses/	120 bytes	Present	Third-party notice material
knowledge/, provenance/, sbom/, signatures/, tools/, attestations/	Not reported for this reference Cell	Optional by format	Additional profile material for richer artifact variants; not required for minimum validity
Total component bytes before compression	955,161 bytes	Measured	Full uncompressed reference artifact content
Bundle archive bytes	369,595 bytes	Measured	Packaged .cell archive footprint

14.8 What the Current Evidence Supports

Taken together, the clean evidence bundle supports a narrower but stronger set of claims than the older baseline framing.

The evidence supports the claim that a Tasklet Cell can be packaged as a bounded local artifact, executed by a local Runner, checked by a Verifier Pack, rejected safely when malformed or policy-invalid, and instrumented with trace and abstention behavior that remains repeatable at the evaluated output level. It also supports the claim that the surrounding reference system has reached a meaningful level of integration maturity in the sense that a release-readiness sweep can be executed repeatedly with complete pass coverage.

The evidence does not support claims that AGI has been achieved, that the broader AGIF fabric has been realized, that CellPOS has been commercially validated at scale, or that the current system is universally robust across open-domain tasks. The clean A6 result also does not justify universal finance-document extraction claims; it justifies a bounded, suite-scoped claim.

This is the correct scientific posture for the paper: the current implementation establishes offline verifiable local Cell artifacts with executable evidence and a product-shaped case-study path. It does not establish general intelligence, broad market deployment, or universal task superiority.

14.9 Threats to Validity and Reporting Boundaries

Several limitations should be stated explicitly.

Single-environment execution. The current clean evidence was generated in one local execution environment: Apple M4, macOS 26.3.1 arm64, Python 3.14.3, and Rust 1.93.1. This is sufficient for a portable audit/results bundle and a bounded re-execution workflow, but not enough to support strong cross-platform performance or compatibility claims.

Small benchmark cardinality in some suites. The hallucination suite has 4 cases per repeat, the extraction suite 6, and the reasoning-trace suite 12. Repetition improves stability estimates, but it does not create benchmark breadth or replace broader task diversity.

Clean-anchor supersession of the dirty baseline. The canonical clean anchor no longer carries the dirty-working-tree caveat associated with 533c63e. Historical material tied to 533c63e should therefore be read as baseline-failure context rather than as the evaluated paper state.

Integration readiness is not deployment validation. The release-readiness result is strong evidence of an integrated local system, but it is not the same thing as longitudinal field validation, broad customer usage, or cross-environment deployment proof.

Reproducibility boundaries remain narrower than universal bit-for-bit claims. The clean bundle records no explicit known gaps, but the methodological boundary still matters: the strongest determinism evidence is at the level of evaluated outputs and suite results. It should not automatically be interpreted as proof that every intermediate file and every derived artifact is bit-identical across repeats.

Task-performance claims remain bounded. The historical A6 blocker is resolved in the clean anchor, but the claim surface remains limited to the evaluated workload and harness. Broader benchmark expansion and broader external validation remain future work.

This is also where the paper should state its methodology most explicitly: community norms increasingly treat research artifacts as part of the scientific object; therefore, the released evidence pack is treated as the citation anchor for empirical claims, and the paper distinguishes repeatable result outputs from full bit-for-bit reproduction of every intermediate build artifact (Association for Computing Machinery, n.d.; Reproducible Builds Project, n.d.-a; Reproducible Builds Project, n.d.-b).

14.10 Evaluation Conclusion

The correct conclusion is not that AGIF has been achieved or that the system is broadly production-proven. The correct conclusion is that the current implementation provides evidence-backed support for AGIF Tasklet Cells as an offline verifiable artifact architecture with a concrete private-pilot implementation path.

At the same time, the paper remains bounded rather than absolute. The clean anchor resolves the historical A6 numeric-extraction blocker, but the claim surface still stops short of universal finance-document extraction and still distinguishes repeatable output-level determinism from stronger artifact-level reproducibility claims. Broader fabric claims, broader task-quality claims, and broader deployment validation remain future work.

15. CellPOS Case Study: Embedded Offline Intelligence in POS

A systems paper about bounded local artifacts is more persuasive when it is grounded in a real product domain rather than left at the level of abstract architecture. In this paper, that role is filled by CellPOS. CellPOS is not presented as proof of broad commercial validation, nor is it the whole paper. It is presented as a private-pilot product context in which the Tasklet Cell architecture is embedded into a workflow that is operationally meaningful, product-relevant, and constrained enough to test the paper's central design claims.

CellPOS is useful here because it moves the discussion from architectural possibility to evidence tied to a product context. The build-proof record does not document only source code or planning artifacts. It documents a completed offline productization and a frozen private-pilot baseline for a packaged local product that combines CellPOS Core with an Embedded AGIF Brain (ENFSystems LLC, 2026). The reported baseline includes a packaged runtime server, packaged browser surfaces, an embedded AGIF child-process runtime, offline local license validation, an offline signed manual update flow, clean-room packaged acceptance, and a published GitHub release. Taken together, those elements make CellPOS more concrete than a generic "future application" section. They give the paper a real product boundary to point to while preserving a narrow and honest claim surface.

That matters because the paper's central claim is not merely that Tasklet Cells are conceptually attractive. It is that bounded, verifier-backed local artifacts can be embedded into product-shaped workflows in a domain where trust, timing, operational continuity, and fail-closed behavior matter. CellPOS is a useful proving ground for that claim because POS combines structured business events, operational time pressure, diagnostics, local workflow memory, and a need for bounded intelligent assistance without ambient cloud dependence.

15.1 Why POS Is a Suitable Application Domain

Point-of-sale systems are a suitable application domain because they align well with the strengths of the Tasklet Cell model.

First, POS workflows are highly structured. Orders, tickets, discounts, taxes, kitchen states, payment events, and fiscal or runtime errors are typed business objects with explicit operational meaning. That makes them well suited to contract-driven capability artifacts.

Second, POS workflows are operationally time-sensitive. Front-of-house staff, kitchen staff, and operators need assistance that is local, fast, and easy to interpret. A bounded local artifact is a better fit for that setting than a system that depends on remote latency, shifting service behavior, or broad autonomous loops.

Third, POS is compliance-adjacent. In Germany, technical security systems for electronic record-keeping are governed by the KassenSichV framework and the BSI TR-03153 technical guideline, while payment-card environments may also fall within PCI DSS scope when cardholder-data processing is relevant (Bundesamt für Sicherheit in der Informationstechnik [BSI], n.d.; KassenSichV, n.d.; PCI Security Standards Council, 2024). That increases the value of fail-closed behavior, explicit traceability, and understandable diagnostics without requiring this paper to make broader legal or certification claims than the evidence supports.

Fourth, POS is memory-sensitive without requiring cloud dependence. Remembering prior corrections, preferences, reorder patterns, and operational bottlenecks can improve usability substantially, but those benefits do not require unbounded remote profiles or hidden behavioral telemetry. They are fully compatible with bounded local memory and reversible personalization.

Fifth, POS is operationally realistic. If Tasklet Cells are to be argued as a practical software-artifact model rather than as a purely academic abstraction, it helps to show them inside a domain in which users can benefit from anomaly detection, bottleneck insight, diagnostics, and bounded recall inside real product workflows. POS is one such domain.

For these reasons, CellPOS is not just a convenient example. It is a domain in which the paper's architectural claims can be exercised under meaningful product constraints.

15.2 Integration Architecture

In the CellPOS case study, Tasklet Cells are embedded as local capability units inside the POS application rather than as external assistants. The host application remains the main workflow surface. Cells do not replace the POS system's transaction model, state machine, browser surfaces, or regulatory logic. Instead, they sit beside those systems as bounded capability artifacts invoked under explicit conditions.

The documented product package includes:

- a packaged runtime server,
- packaged browser surfaces,
- a packaged embedded AGIF child-process runtime,
- offline local license validation,
- an offline signed manual update flow,
- admin visibility for version, AGIF state, license state, and update state,
- and fail-closed behavior for invalid license, invalid update, and invalid runtime conditions (ENFSystems LLC, 2026).

The main browser surfaces are:

- /pos
- /kds
- /order
- /admin

The CellPOS repository record documents a concrete packaged pilot artifact. The frozen pilot baseline is anchored to repository CellPOS, tag cellpos-private-pilot-v0.1.0, commit

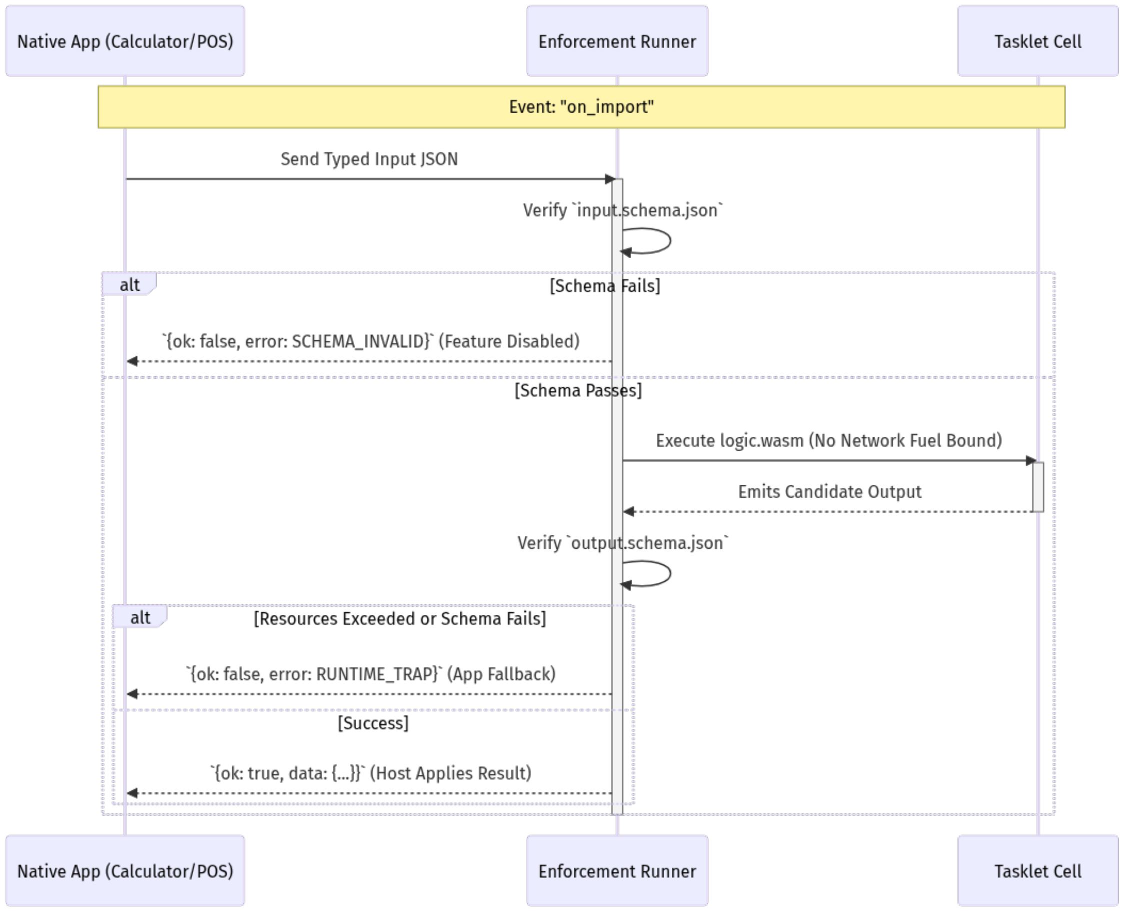
904de056207eebdb24fc820d8f409c2e4323c0eb, and artifact CellPOS-0.1.0-prod007.0.tar.gz with recorded SHA-256 80f31c72d7054f34a7e802f3ed324bb9196bcc5e1202be0148bca121e06402dd and recorded size 28,359,248 bytes (ENFSystems LLC, 2026). That repository-recorded evidence supports the claim that CellPOS was built into a runnable offline package rather than left as a source-only codebase, while independent inspection of the packaged artifact itself depends on access to that tarball.

Architecturally, the integration can be understood in five layers.

1. *Host POS application.* This remains the system of record for orders, tickets, payment events, operator actions, kitchen state, and regulatory workflow.
2. *Trigger and request layer.* Host-side events such as discount application, ticket updates, import operations, kitchen-delay states, or runtime and TSE failures are converted into typed local requests containing only the minimum context required by the Cell contract.
3. *Runner and local policy boundary.* Every Cell request passes through the same local Runner boundary defined earlier in the paper. Structural validation, integrity checks, verifier execution, policy binding, and bounded execution all occur here.
4. *Cell layer.* Cells provide narrow functions such as anomaly detection, bottleneck insight, bounded explanation, ranking, or recall. They remain contract-driven and verifier-backed.
5. *Local state layer.* Where memory is needed, CellPOS uses bounded local stores for session state, episodic state, and personalization state. In the intended profile, this is implemented through explicit local storage such as SQLite rather than through remote profile services.

This integration pattern preserves the separation between business-workflow authority and artifact-level assistance. The POS system remains the owner of transactions, order state, and regulatory workflow. Cells remain bounded helpers. As documented in Appendices F and G, the packaged product starts from the installed layout, renders its browser surfaces, starts the embedded AGIF child process, performs handshake and health checks, and exposes degraded and fail-closed states when runtime context is intentionally reduced. Figure 6 summarizes this embedding sequence.

Figure 6
Native Embedding Sequence


Native App (Calculator/POS)
Enforcement Runner
Tasklet Cell
Event: "on_import"
Send Typed Input JSON
Verify `input.schema.json`
alt
[Schema Fails]
`{ok: false, error: SCHEMA_INVALID}` (Feature Disabled)
[Schema Passes]
Execute logic.wasm (No Network Fuel Bound)
Emits Candidate Output
Verify `output.schema.json`
alt
[Resources Exceeded or Schema Fails]
`{ok: false, error: RUNTIME_TRAP}` (App Fallback)
[Success]
`{ok: true, data: {...}}` (Host Applies Result)
Native App (Calculator/POS)
Enforcement Runner
Tasklet Cell

15.3 Bounded Local Memory and Transparent Reasoning Trace

CellPOS is useful as a case study because it needs both memory and traceability, but only in bounded forms.

On the memory side, the product benefits from retaining local context such as prior accepted corrections, preferred wording, reorder tendencies, approved aliases, and operational histories. The proof material explicitly documents persistent local SQLite memory as part of the packaged AGIF product layer (ENFSystems LLC, 2026). That is exactly the kind of local store that fits the bounded-memory model described earlier in the paper: useful enough to support continuity, but still local, inspectable, and bounded.

Within the paper's architectural terms, the CellPOS memory model can be read as three local layers:

- *session memory* for continuity inside a single operational sequence,
- *episodic memory* for bounded recall of prior relevant events or accepted corrections,
- and *Personal Layer state* for reversible, policy-gated adaptation such as alias maps, preferences, or threshold calibration.

On the traceability side, the proof material also documents local reasoning traces and AGIF operational visibility in /admin (ENFSystems LLC, 2026). That matters because workflows in this domain are more trustworthy if the operator can see why something was flagged, why a runtime path was refused, or why the system is surfacing a particular diagnostic. In the CellPOS setting, the reasoning trace is not meant to be an open-ended chain-of-thought surrogate. It is meant to be a structured, user-facing explanation surface derived from local artifact behavior.

That combination of bounded local memory and transparent trace is one reason this case study belongs in the paper. It shows that local intelligence can be useful not only because it predicts or scores something, but because it can remember enough to help and explain enough to be trusted.

15.4 Product-Visible Workflows

The value of the CellPOS case study becomes clearest when the artifact model is mapped onto concrete product-visible workflows. The documented AGIF layer inside the product exposes four workflows:

- fraud and anomaly detection,
- kitchen bottleneck insight,
- plain-language diagnostics,
- and reorder and preference recall.

These are not generic aspirations pasted onto the case study. They are the product-visible workflows the packaged AGIF layer is documented to support (ENFSystems LLC, 2026).

15.4.1 Fraud and Discount-Abuse Detection

The first workflow is fraud and discount-abuse detection. POS systems contain repeated patterns of operator behavior around discounts, overrides, voids, and price adjustments. Not all unusual behavior is malicious, but some patterns are operationally suspicious enough to justify local review.

A Tasklet Cell is a good fit here because the workflow is structured, the signals are local, and the decision surface can remain narrow. Deterministic checks can cover hard constraints such as impossible discount structures or explicit rule violations. A compact local knowledge layer can provide approved discount types, normal ranges, or role-specific policy hints. An optional micro-model can score suspicious patterns among otherwise valid transactions. The result is a local anomaly indication rather than an opaque remote fraud decision.

The key point is that the system remains fail-closed and review-oriented. The Cell may flag or rank suspicious events, but the host system still controls what action is taken next. The artifact supports bounded decision support, not autonomous financial adjudication.

15.4.2 Kitchen Bottleneck Insight

The second workflow is kitchen bottleneck insight. Restaurants and kitchens generate repeated operational patterns such as ticket accumulation, prep delays, sequencing conflicts, and service slowdowns. Much of this information is local and transient, yet still structured enough to support bounded local inference.

A Cell can help by detecting patterns that are not captured by a single hard-coded rule alone. Deterministic logic can still enforce obvious thresholds and ticket-state rules. A compact knowledge layer can contribute station mappings, prep categories, or operational labels. An optional micro-model can support ranking or classification over likely bottleneck states. The result is an operational hint surfaced locally to staff or managers, not an autonomous scheduling system.

This workflow is important because it shows that Tasklet Cells are not limited to static document-like tasks. They can also assist in dynamic operational environments, provided the capability boundary remains narrow and the outputs remain advisory or otherwise bounded.

15.4.3 Plain-Language Diagnostics for TSE and Runtime Issues

The third workflow is plain-language diagnostics for TSE and runtime issues. This is a fitting product-facing use of the architecture because it combines structured local state, strict operational boundaries, and a real user need for understandable explanation.

In many POS contexts, operators face failures that are technically specific but operationally opaque: TSE connection issues, invalid fiscal states, printer or TSE timing mismatches, runtime refusal conditions, or environment-specific device problems. A bounded local artifact can translate those states into structured diagnostics and plain-language guidance without relying on cloud troubleshooting loops.

This workflow is especially compatible with the Tasklet Cell model because the diagnostic path can remain heavily deterministic. The host system and runtime already know much of the relevant state. The Cell's role is not to invent explanations from nowhere, but to classify, rank, and explain the most likely local problem state in a way that is transparent and user-legible.

This is also where the reasoning trace becomes useful. A plain-language diagnostic is more trustworthy if it can be tied back to explicit local evidence such as a refusal code, device state, or bounded runtime condition. In the case-study framing, the Cell is best understood as an explanation and triage layer over local system evidence rather than as a replacement for the underlying fiscal or runtime subsystem.

15.4.4 Reorder and Preference Recall

The fourth workflow is reorder and preference recall. POS systems often benefit from remembering local, user-approved patterns such as repeat items, operator-specific preferences, customer-facing defaults, or prior accepted corrections. This is a natural fit for bounded local memory because the value comes from continuity rather than from open-ended prediction.

A Cell can support reorder and preference recall by combining local episodic memory with bounded ranking or recall logic. The key architectural point is that this memory remains local, explicit, and reversible. The system is not trying to build an unrestricted behavioral profile. It is trying to remember enough useful local context to reduce friction in repeated workflows.

This workflow matters because it makes the personalization story concrete. It shows why bounded local memory is not an academic side note, but a real product requirement.

15.5 Private-Pilot Status and Scope Limits

A key aspect of the proof material is that it turns CellPOS from a plausible concept into a documented product baseline. The repository evidence shows:

- productization marked 8/8 complete,
- final classification Ready for private pilot,
- a real packaged artifact,
- offline licensing,
- offline signed manual updates,

- packaged acceptance from an isolated install layout,
- and a published GitHub baseline release (ENFSystems LLC, 2026).

The frozen pilot baseline is documented concretely as:

- Private GitHub repository record: Ahsadin/CellPOS
- Private GitHub release record: cellpos-private-pilot-v0.1.0
- tag: cellpos-private-pilot-v0.1.0
- baseline commit: 904de056207eebdb24fc820d8f409c2e4323c0eb
- packaged artifact: CellPOS-0.1.0-prod007.0.tar.gz
- repository-recorded artifact path omitted from publication copy
- artifact SHA-256: 80f31c72d7054f34a7e802f3ed324bb9196bcc5e1202be0148bca121e06402dd
- recorded size: 28,359,248 bytes

The packaged proof also documents the installable layout and lifecycle surface, including:

- packaged layout directories such as app/, bin/, config/, product/, and site-data/,
- packaged lifecycle scripts including bin/start-cellpos.sh, bin/import-license.sh, and bin/apply-update.sh,
- offline license behavior based on a local signed JSON license artifact,
- local-only validation by bundled public trust material,
- signed update import with signature and payload-hash verification before replacement,
- and security-hygiene checks confirming that the shipped distributable excludes private signing keys, carries only the public trust keys required for local verification, and keeps private signing material outside the distributable (ENFSystems LLC, 2026).

The proof material also includes more operationally specific evidence than a generic release note. It documents a packaged acceptance sweep from an isolated clean-room install outside the repository tree, packaged startup from bin/start-cellpos.sh, browser rendering for /pos, /kds, /order, and /admin, embedded AGIF child-process startup and health checks, manager workflows operating from the packaged runtime, fail-closed handling of invalid or missing license state, fail-closed rejection of invalid signed updates, restart persistence of activated state, recorded verification commands, and explicit pass tokens for the productization and clean-room acceptance steps (ENFSystems LLC, 2026).

That evidence is substantial, but the case-study claim must still be read with the right level of modesty. CellPOS is a private-pilot case study, not a claim of broad commercial rollout, universal market validation, or complete production maturity.

That distinction is important for three reasons.

First, the role of the case study is architectural demonstration. It shows that Tasklet Cells can be integrated into a real product domain with meaningful workflows, not that all deployment questions have already been resolved.

Second, the evidence boundary remains artifact-layer and workflow-level rather than full market-scale validation. The broader evaluation in the paper supports release readiness, fail-closed behavior, reasoning-trace validity, and bounded runtime properties for the local artifact architecture. It does not, by itself, prove mature deployment outcomes across all POS environments or broad empirical superiority for each workflow.

Third, the proof material—especially Appendices E through H and the build-proof record—states the remaining limits clearly. It does not claim production-secure signing infrastructure yet. It does not claim second-machine or fresh-OS proof yet in the repository. It also states that later commits exist after the frozen pilot baseline, but that the pilot proof point remains the tagged release commit. In other words, the case study is strong because it is real, but also because it does not pretend to be more finished than it is.

This is the correct scope for the case study. CellPOS is not offered as evidence that the broader AGIF agenda has been realized, nor as proof that every intelligent POS feature is now solved. It is offered as evidence that bounded, verifier-backed local artifacts can be embedded into a product-shaped workflow in a commercially meaningful domain. That is enough to make the paper's central claim concrete while still preserving an honest boundary between architectural evidence and broader deployment claims. The broader unresolved items—external validation, second-machine and fresh-OS proof, hardware and device-matrix validation, production-secure signing infrastructure, and the wider AGIF fabric agenda—are treated as future work rather than as completed proof.

16. Limitations and Future Work

The Tasklet Cell architecture is intentionally narrow. That narrowness is one of its strengths, but it also creates real limitations that should be stated plainly. This section does not treat those limitations as rhetorical obstacles to be dismissed. It treats them as part of the paper's contribution boundary. The architecture is useful precisely because it does not attempt to solve every AI deployment problem at once.

Several limits follow directly from the design choices developed throughout the paper. A verifier-backed artifact model is easier to reason about than an unconstrained agent runtime, but it cannot support the same range of open-ended behavior. A bounded local pipeline is easier to embed safely than a cloud-centric assistant, but it inherits strict limits on authority, scope, and adaptability. A fail-closed system is easier to trust in product workflows, but it may reject more often when local validity conditions are not met. These are not incidental trade-offs. They are structural consequences of the architecture.

16.1 Not for Open-Ended Creative or General-Purpose Agent Tasks

Tasklet Cells are not designed for open-ended creative tasks, broad conversational assistance, or general-purpose agent behavior. The architecture assumes a bounded task surface, explicit contracts, a narrow capability role, and a local validity gate. Those assumptions are directly at odds with many workloads that depend on open-ended exploration, improvisational generation, indefinite planning, or unconstrained tool chaining.

This means that the architecture is a poor fit for tasks such as:

- unrestricted creative writing,
- broad conversational assistance across many domains,
- open-ended research agents,
- indefinite multi-step planning with evolving goals,
- and autonomous systems that are expected to expand their own task scope dynamically.

That limitation is not a defect in the sense intended by this paper. It is part of the design center. The model is built for bounded local capability, not for the largest possible behavioral envelope. Future work could investigate how multiple bounded artifacts might be composed into richer systems, but that is different from claiming that a single Tasklet Cell should already behave like a general-purpose assistant.

16.2 Verification Limits

The architecture places verification at the center of local validity, but that does not mean that every useful behavior is equally easy to verify. Some tasks admit strong schemas, clear comparison rules, and straightforward golden cases. Others do not.

In practice, verification becomes harder when:

- outputs are intrinsically fuzzy or subjective,
- correctness depends on long implicit context,
- ranking quality depends on human preference rather than crisp task labels,
- or task success depends on latent reasoning quality that is only partially observable at the output boundary.

The Verifier Pack model helps by making local validity explicit, but it does not magically convert every difficult AI evaluation problem into a simple pass/fail rule. Some Cells will inevitably have stronger verifier surfaces than others. In those cases, the architecture can still be useful, but the verifier boundary may need to rely on narrower acceptance criteria, human-review paths, or stronger abstention and rejection behavior.

This means that the paper's verification story should not be read as a universal claim about all AI tasks. It is a claim about a bounded class of artifact-friendly tasks for which local conformance and local validity are tractable enough to matter.

16.3 Sandbox and Runtime Parity Limits

The architecture assumes that a Runner can enforce bounded execution and explicit capability limits. In practice, however, runtime substrates and deployment environments are not perfectly uniform. Different operating systems, runtime implementations, packaging modes, or local system policies may vary in how strongly they can enforce memory limits, process isolation, signal handling, file-access control, or debug-mode restrictions.

This creates a practical limit on the portability of guarantees. The paper can define a clean enforcement model, but different environments may only approximate parts of it. Some deployment profiles will be able to provide stronger local isolation than others. Some runtime behaviors will be easier to replay or compare on one platform than on another. Some security or observability guarantees will depend on the quality of the underlying host environment rather than on the artifact model alone. This limitation also applies to determinism: when numerical behavior depends on floating-point execution, compiler behavior, or parallel runtime paths, reproducibility across platforms can become weaker than the architecture would ideally prefer.

For that reason, the architecture should be understood as defining a target enforcement posture, not as claiming that every runtime-and-operating-system pair implements that posture identically. This point should be read strongly: WebAssembly- and Runner-based isolation can be very useful, but the sandbox model remains scoped and constrained, and real runtime implementations can still have security defects, miscompilations, or correctness-affecting bugs. Classical security design principles remain relevant here. Least privilege, fail-safe defaults, and complete mediation help explain the architecture's posture, but they do not by themselves prove that a particular sandbox or runtime implementation is secure in every environment (Saltzer & Schroeder, 1975). Later engineering work may therefore need to define profile tiers, platform-specific hardening guidance, explicit capability matrices, and patch-management expectations to make those differences visible.

A related limitation is that the architecture depends on correct handling of untrusted artifact bundles at the ingestion boundary. Archive extraction, path normalization, decompression defense, and bundle parsing remain practical risk surfaces if implemented incorrectly. The bundle and Runner model reduce those risks only if the corresponding archive-safety rules are implemented robustly.

16.4 A6 Numeric Extraction Limitation

The clearest historical empirical limitation in the project was the A6 numeric extraction benchmark at baseline 533c63e. In the clean anchor at 78a1635, that specific failure is resolved: the strengthened A6 benchmark now passes with grounded grand-total, tax-total, and subtotal extraction, along with arithmetic consistency and numeric evidence grounding. The remaining limitation is therefore no longer the exact historical grand-total failure itself, but the bounded evaluation scope. The present paper claim remains limited to invoices and receipts in en-US, de-DE, and es-ES, with broader public-benchmark expansion and broader private holdout coverage still outside the current claim boundary.

This limitation matters because it is not merely an abstract warning. It is a concrete reminder that evaluated success still has scope conditions. The architecture can support the claim that bounded local artifact execution, fail-closed behavior, reasoning-trace validity, and structural extraction behaviors are real. It still cannot justify a broad claim that the extraction workflow is solved at full production-grade breadth across arbitrary financial-document distributions.

The paper should treat this as a real limitation rather than as a wording inconvenience. It also sharpens a second point: in parts of the current evidence base, output-level determinism is established more strongly than full artifact-level reproducibility. That distinction matters, and the paper should preserve it rather than compressing all reproducibility claims into one undifferentiated label.

16.5 External Validation Still Pending

The current evidence base is strong at the artifact layer, but broader external validation is still pending. The release-readiness and safety evidence show that the architecture behaves in disciplined ways under repeated local evaluation. The CellPOS build proof shows that a packaged offline product baseline exists and is ready for private pilot (ENFSystems LLC, 2026). Those are meaningful results, but they are not the same thing as broad external validation.

What remains pending includes, at a minimum:

- broader user-facing pilot evaluation,
- field validation across more operating environments,
- comparative operator studies,
- longitudinal evidence on personalization quality and rollback behavior,
- stronger fresh-machine or fresh-OS validation beyond the current repository-backed proof,
- hardware and device-matrix validation in broader POS deployment conditions,
- and larger real-world evidence that the workflows remain useful under normal commercial variation.

This is especially important for the case-study interpretation. A private-pilot baseline is not a multi-site deployment study. A packaged proof is not a market-scale validation record. Current empirical support remains strongest for repeated artifact-layer evaluation in a limited environment rather than for broad cross-environment deployment validation. The architecture is therefore further along than a concept note, but not yet at the point where broad external generalization claims would be justified.

For personalization specifically, this also means that longer-term update behavior remains under-evaluated. Continual-learning research shows that new updates can interfere with previously useful behavior, a failure mode widely described as catastrophic forgetting (Kirkpatrick et al., 2017). The Tasklet Cell architecture addresses that risk through bounded personalization, rollback, and learning gates, but stronger longitudinal validation is still future work.

16.6 Production-Secure Signing Infrastructure as Adjacent Work

The paper's core contribution is the artifact model, local enforcement boundary, and evidence-backed execution posture. A production-secure signing and update infrastructure is closely related to that contribution, but it is not identical to it.

The current materials already show the importance of trust anchors, signature validation, and offline signed manual update flows (ENFSystems LLC, 2026). However, a fully hardened signing infrastructure involves additional concerns: key custody, rotation policy, revocation procedures, production signing workflows, trust-store distribution, secure release governance, and operational handling of compromised or deprecated trust material.

Those concerns are essential for product maturity, but they belong partly to release engineering and supply-chain operations rather than only to the core artifact model. For that reason, the paper should describe production-secure signing infrastructure as adjacent work: highly relevant, already partially reflected in the architecture, but not itself the whole research contribution.

This distinction also helps preserve the scope boundary. The paper can argue convincingly that trust anchors must remain local and that manifest-supplied trust references are metadata only. It does not need to claim that the full operational signing lifecycle is already complete in all production-hardening dimensions.

16.7 Broader AGIF Fabric as Future Work

Finally, the broader AGIF fabric remains future work. This paper defines and evaluates the artifact layer, the Runner boundary, bounded local memory, replayability, and a concrete product-shaped case study. It does not establish fabric-level coordination, descriptor gossip, world-model aggregation, open-ended multi-Cell adaptation, or broad distributed cognition at scale (Khan, 2026).

Those questions remain important. In fact, they are part of what makes AGIF a broader research agenda rather than a single paper claim. If bounded local artifacts are to participate in richer future systems, then future work will need to address issues such as:

- safe composition of multiple Cells,
- coordination across bounded artifact boundaries,
- policy-aware descriptor exchange,
- multi-Cell memory and replay semantics,
- and stronger distributed trust and governance models.

The point of this paper is not to solve those problems prematurely. It is to define a disciplined local unit that could plausibly participate in such systems later. In that sense, the paper should be read as contributing one layer of the broader AGIF research stack, not as claiming completion of the stack itself.

Taken together, these limitations are not signs of architectural failure. They are the explicit edges of the paper's contribution. The Tasklet Cell model is strong precisely because it is willing to stay narrow: not a universal agent architecture, not a proof of full fabric cognition, not a final answer to every deployment question, but a bounded and evidence-backed local artifact model with a clear path for future work.

17. Conclusion

This paper has argued for a narrower and more disciplined way of delivering useful AI capability. Rather than treating intelligence as a remote service or an open-ended agent loop, it defines the Tasklet Cell as a bounded, contract-driven software artifact that can be validated locally before execution and constrained locally while it runs. A Tasklet Cell is packaged with explicit contracts, integrity metadata, and a Verifier Pack, and it becomes executable only under a local Runner that enforces offline-first execution, bounded authority, resource limits, and fail-closed behavior.

That architecture is intentionally limited, and that limitation is one of its strengths. By keeping the unit of deployment narrow, verifier-backed, and replayable, the model becomes more amenable to assurance, easier to embed safely, and easier to reason about than broader cloud-centric agent systems. The paper's argument is therefore not that all AI should be reduced to tiny local modules, nor that remote services have no role. It is that a large and commercially important subset of AI functionality can be delivered in a stricter and more defensible form: as offline, verifiable software artifacts.

The contributions established in the paper sit at five connected layers. At the artifact layer, the paper defines the Tasklet Cell as a bounded capability unit with explicit contracts, integrity metadata, and a Verifier Pack. At the enforcement layer, it defines the Runner as the local mediation boundary that validates, constrains, and rejects execution under explicit policy. At the capability layer, it shows how deterministic tools, compact local knowledge, and optional micro-models can be composed into useful narrow functionality without collapsing into open-ended autonomy. At the systems layer around that core, it defines bounded local memory, replay-oriented observability, and an explicit trust model in which orchestrators remain untrusted until mediated locally and trust anchors remain local rather than artifact-defined. At the evidence layer, it shows that these claims can be grounded in executable proof rather than left as architecture diagrams and design rhetoric.

The empirical results support that claim surface at the level the paper now states. The clean evidence anchor shows a fully passing release-readiness sweep, zero unsafe-allow events in the scoped hallucination-safety benchmark, complete rejection on the provided fail-closed negative suite, and perfect reasoning-trace schema validity and evidence alignment under repeated local execution. The strengthened A6 benchmark also passes on the clean anchor, but the paper keeps that result properly scoped to the evaluated invoice and receipt workload rather than inflating it into a universal finance-document claim. That is the right scientific posture for the current state of the work: strong evidence for a bounded local artifact architecture, not a grand claim of general intelligence or universal task robustness.

The CellPOS case study makes that boundary concrete. It shows that Tasklet Cells can be embedded into a real product domain with structured business workflows, bounded local memory, transparent reasoning traces, offline licensing, offline signed manual updates, and product-visible workflows such as fraud and discount-abuse detection, kitchen bottleneck insight, plain-language diagnostics, and reorder and preference recall. The proof record is strong enough to support a ready-for-private-pilot claim. It is not, and should not be read as, a claim of broad commercial rollout, universal market validation, or fully hardened production maturity.

The broader AGIF vision remains future work. This paper does not establish fabric-level coordination, descriptor gossip, world-model aggregation, open-ended multi-Cell adaptation, or distributed cognition at scale. What it does establish is narrower and, for that reason, more defensible: if a broader AGIF fabric is ever to become trustworthy and product-embeddable, it will need local units that can be validated, bounded, replayed, and rejected safely. Tasklet Cells are proposed as one such unit at the software-artifact level. The paper should therefore be read as contributing one disciplined layer of the broader AGIF research stack rather than as claiming completion of that stack.

The conclusion is therefore straightforward. The current implementation provides evidence-backed support for AGIF Tasklet Cells as an offline, verifiable artifact architecture with a concrete private-pilot implementation path. It does not solve every AI deployment problem, and it does not claim completion of the broader AGIF agenda. What it does provide is a disciplined artifact model, a local enforcement model, a bounded capability pipeline, and an evidence-backed product path that make local intelligence a realistic and commercially meaningful design direction when the design target is not maximal openness, but controlled usefulness under explicit constraints.

References

Anderson, J. P. (1972, October). ***Computer security technology planning study*** (Vol. I, Technical Report ESD-TR-73-51). HQ Electronic Systems Division (AFSC). https://csrc.nist.gov/files/pubs/conference/1998/10/08/proceedings-of-the-21st-nissc-1998/final/docs/early-cs-papers/ande72a.pdf

Association for Computing Machinery. (n.d.). ***Artifact review and badging***. https://reviewers.acm.org/training-course/artifact-review-and-badging

Bundesamt für Sicherheit in der Informationstechnik. (n.d.). ***BSI TR-03153: Technical security systems for electronic record-keeping systems***. https://www.bsi.bund.de/EN/Themen/Unternehmen-und-Organisationen/Standards-und-Zertifizierung/Technische-Richtlinien/TR-nach-Thema-sortiert/tr03153/tr03153_node.html

Cavoukian, A. (2011). ***Privacy by design: The 7 foundational principles***. Information and Privacy Commissioner of Ontario. https://www.ipc.on.ca/sites/default/files/legacy/2018/01/pbd-1.pdf

Cybersecurity and Infrastructure Security Agency. (n.d.). ***Software Bill of Materials (SBOM)***. https://www.cisa.gov/sbom

ENFSystems LLC. (2026). ***63_CELLPOS_BUILD_PROOF_REPORT*** [Internal build proof report].

Jansen, W., & Grance, T. (2011). ***Guidelines on security and privacy in public cloud computing*** (NIST Special Publication 800-144). National Institute of Standards and Technology. https://doi.org/10.6028/NIST.SP.800-144

Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A., Adam, H., & Kalenichenko, D. (2018). Quantization and training of neural networks for efficient integer-arithmetic-only inference. In ***Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*** (pp. 2704–2713). https://doi.org/10.1109/CVPR.2018.00286

JSON Schema. (2022). JSON Schema Draft 2020-12. https://json-schema.org/draft/2020-12

KassenSichV. (n.d.). ***Ordinance establishing the technical requirements for electronic record-keeping and security systems in business transactions***. https://www.gesetze-im-internet.de/englisch_kassensichv/index.html

Kent, K., & Souppaya, M. (2006). Guide to computer security log management (NIST Special Publication 800-92). National Institute of Standards and Technology. https://doi.org/10.6028/NIST.SP.800-92

Khan, D. Z. (2025a). ***Embedded Neural Firmware (ENF): A deterministic, offline "Neural BIOS" for batteryless devices***. https://doi.org/10.5281/zenodo.17298403

Khan, D. Z. (2025b). ***ENF Technical Note 01: Embedded Neural Firmware—The embedded AI crisis and the case for sealed neural firmware*** (Report No. 1). ENFSystems LLC. https://doi.org/10.13140/RG.2.2.13715.34084

Khan, D. Z. (2025c). ***ENF Technical Note 02: Related work: TinyML toolchains, secure boot, energy harvesting, and formal verification for sealed neural firmware*** (Report No. 2). ENFSystems LLC. https://doi.org/10.13140/RG.2.2.12037.61925

Khan, D. Z. (2025d). ***ENF Technical Note 04: The ENF framework and meta-vision***. ENFSystems LLC. https://doi.org/10.13140/RG.2.2.16163.52002

Khan, D. Z. (2026). ***Adaptive General Intelligence Fabric (AGIF): A publish-ready concept paper*** [Internal concept paper]. ENFSystems LLC.

Kirkpatrick, J., Pascanu, R., Rabinowitz, N., Veness, J., Desjardins, G., Rusu, A. A., Milan, K., Quan, J., Ramalho, T., Grabska-Barwinska, A., Hassabis, D., Clopath, C., Kumaran, D., & Hadsell, R. (2017). Overcoming catastrophic forgetting in neural networks. ***Proceedings of the National Academy of Sciences of the United States of America, 114***(13), 3521–3526. https://doi.org/10.1073/pnas.1611835114

Liu, Y., Deng, G., Li, Y., Wang, K., Wang, Z., Wang, X., Zhang, T., Liu, Y., Wang, H., Zheng, Y., Zhang, L. Y., & Liu, Y. (2023). ***Prompt injection attack against LLM-integrated applications***. arXiv. https://arxiv.org/abs/2306.05499

MITRE. (n.d.-a). ***CWE-22: Improper limitation of a pathname to a restricted directory ('Path Traversal')***. https://cwe.mitre.org/data/definitions/22.html

MITRE. (n.d.-b). ***CWE-409: Improper handling of highly compressed data (data amplification)***. https://cwe.mitre.org/data/definitions/409.html

National Institute of Standards and Technology. (n.d.). ***Software Bill of Materials (SBOM)***. https://csrc.nist.gov/glossary/term/software_bill_of_materials

National Institute of Standards and Technology. (2015). ***Secure Hash Standard (SHS)*** (FIPS PUB 180-4). https://doi.org/10.6028/NIST.FIPS.180-4

National Institute of Standards and Technology. (2020a). NIST Privacy Framework: A Tool for Improving Privacy through Enterprise Risk Management (Version 1.0) (NIST CSWP 01162020). https://doi.org/10.6028/NIST.CSWP.01162020

National Institute of Standards and Technology. (2020b). Zero Trust Architecture (Special Publication 800-207). https://doi.org/10.6028/NIST.SP.800-207

National Institute of Standards and Technology. (2023). ***Artificial intelligence risk management framework (AI RMF 1.0)*** (NIST AI 100-1). https://doi.org/10.6028/NIST.AI.100-1

National Institute of Standards and Technology. (2024). ***Cybersecurity supply chain risk management practices for systems and organizations*** (Special Publication 800-161 Rev. 1, Update 1). https://doi.org/10.6028/NIST.SP.800-161r1-upd1

OpenSSF. (n.d.). ***SLSA specification*** (Version 1.2). https://slsa.dev/spec/v1.2/

OWASP Foundation. (n.d.). ***CycloneDX bill of materials standard***. https://cyclonedx.org/

OWASP Foundation. (2024). ***OWASP Top 10 for LLM Applications 2025***. https://genai.owasp.org/resource/owasp-top-10-for-llm-applications-2025/

PCI Security Standards Council. (2024). ***Payment Card Industry Data Security Standard (PCI DSS) v4.0.1***. https://www.pcisecuritystandards.org/document_library/

Reproducible Builds Project. (n.d.-a). ***Definitions***. https://reproducible-builds.org/docs/definition/

Reproducible Builds Project. (n.d.-b). ***Deterministic build systems***. https://reproducible-builds.org/docs/deterministic-build-systems/

Rundgren, A., Jordan, B., & Erdtman, S. (2020). ***JSON Canonicalization Scheme (JCS)*** (RFC 8785). RFC Editor. https://www.rfc-editor.org/rfc/rfc8785

Saltzer, J. H., & Schroeder, M. D. (1975). The protection of information in computer systems. ***Proceedings of the IEEE, 63***(9), 1278–1308. https://doi.org/10.1109/PROC.1975.9939

Samuel, J., Mathewson, N., Cappos, J., & Dingledine, R. (2010). Survivable key compromise in software update systems. In ***Proceedings of the 17th ACM Conference on Computer and Communications Security*** (pp. 61–72). https://doi.org/10.1145/1866307.1866315

Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Hambro, E., Zettlemoyer, L., Cancedda, N., & Scialom, T. (2023). Toolformer: Language models can teach themselves to use tools. ***Advances in Neural Information Processing Systems, 36***. https://openreview.net/forum?id=Yacmpz84TH

Sigstore Project. (n.d.). ***Overview***. https://docs.sigstore.dev/about/overview/

Souppaya, M., Scarfone, K., & Dodson, D. (2022). ***Secure Software Development Framework (SSDF) Version 1.1: Recommendations for Mitigating the Risk of Software Vulnerabilities*** (NIST Special Publication 800-218). National Institute of Standards and Technology. https://doi.org/10.6028/NIST.SP.800-218

SPDX Workgroup. (2024). ***SPDX specification*** (Version 3.0.1). https://spdx.github.io/spdx-spec/v3.0.1/

Torres-Arias, S., Afzali, H., Kuppusamy, T. K., Curtmola, R., Cappos, J., & Cakmak, M. (2019). in-toto: Providing farm-to-table guarantees for bits and bytes. In ***Proceedings of the 28th USENIX Security Symposium*** (pp. 1393–1410). https://www.usenix.org/conference/usenixsecurity19/presentation/torres-arias

Wasmtime Project. (n.d.). ***Interrupting Wasm execution***. https://docs.wasmtime.dev/examples-interrupting-wasm.html

Watson, R. N. M., Anderson, J., Laurie, B., & Kennaway, K. (2010). Capsicum: Practical capabilities for UNIX. In ***Proceedings of the 19th USENIX Security Symposium*** (pp. 29–46). USENIX Association. https://www.usenix.org/event/sec10/tech/full_papers/Watson.pdf

WebAssembly/WASI. (n.d.). ***WASI design principles***. https://github.com/WebAssembly/WASI/blob/main/docs/DesignPrinciples.md

World Wide Web Consortium. (2026, March 10). WebAssembly core specification. https://www.w3.org/TR/wasm-core-2/

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y. (2022). ***ReAct: Synergizing reasoning and acting in language models***. arXiv. https://arxiv.org/abs/2210.03629

Appendix A. Bundle and Manifest Example

This appendix provides a concrete example of the Tasklet Cell artifact format. Its purpose is to make the paper's artifact model explicit at the bundle level rather than leaving it as an abstract architectural claim. It should be read as an example of the bounded software artifact described in the main text: a single-task, contract-driven, verifier-backed local module that is validated by a Runner before execution. It is not intended to define every future extension of the format. Instead, it documents the canonical structure needed to understand the current Tasklet Cell model.

A.1 Canonical `.cell` Artifact Structure

In the current model, a Tasklet Cell is packaged as a .cell artifact. Conceptually, this is a structured bundle containing the executable logic, input and output contracts, integrity metadata, verifier material, and licensing notices needed to treat the Cell as a bounded local artifact rather than as an opaque plugin.

The canonical minimum bundle layout is as follows:

```
<cell>.cell
  manifest.json
  hashes/sha256.json
  schemas/input.schema.json
  schemas/output.schema.json
  verifier/tests.json
  logic/logic.wasm
  licenses/THIRD_PARTY_NOTICES.txt
```

This structure matters because each path serves a specific role in the validity rule. The Runner does not trust the bundle simply because it exists or because it was produced by a known build pipeline. Instead, it must inspect the manifest, check integrity metadata, validate schemas, locate the logic payload, and run the Verifier Pack before the Cell is treated as valid.

A.2 Required Bundle Elements

The required bundle elements serve distinct purposes:

- manifest.json defines the identity, compatibility, task role, limits, contracts, logic references, verifier references, and integrity references for the Cell.
- hashes/sha256.json records the expected hashes for bundle contents so that tampering can be detected before execution.
- schemas/input.schema.json defines the machine-checkable input boundary.
- schemas/output.schema.json defines the machine-checkable output boundary.
- verifier/tests.json contains the built-in Verifier Pack used to establish local validity.
- logic/logic.wasm contains the executable Cell logic in the preferred sandboxed form.
- licenses/THIRD_PARTY_NOTICES.txt carries required third-party notices.

The architecture also allows optional directories such as:

- knowledge/
- model/
- provenance/
- sbom/

- signatures/
- tools/
- attestations/

These optional paths support richer artifact profiles, but they do not replace the minimum validity surface. They may enrich integrity, provenance, inventory, or deployment-policy evaluation, but they do not by themselves make an otherwise invalid Cell executable.

A.3 Example Bundle Layout

A more concrete example of a Tasklet Cell bundle is shown below. This example is illustrative: it shows a richer profile than the canonical minimum and is intended to make the artifact posture easier to visualize.

```
invoice_completeness_validator_v1.cell
  manifest.json
  hashes/sha256.json
  schemas/input.schema.json
  schemas/output.schema.json
  verifier/tests.json
  verifier/cases/accept_complete/input.json
  verifier/cases/accept_complete/output.json
  verifier/cases/reject_missing_invoice_number/input.json
  verifier/cases/reject_missing_invoice_number/output.json
  logic/logic.wasm
  knowledge/rules.json
  provenance/build_info.json
  sbom/spdx.json
  licenses/THIRD_PARTY_NOTICES.txt
```

This example reflects the intended artifact posture of the paper. The Cell is narrow in scope, explicitly structured, and sufficiently self-describing to be validated locally. It is not an open-ended runtime package with implicit authority.

A.4 Example Manifest

A representative manifest is shown below. The exact values are illustrative, but the field classes reflect the current model.

```
{
  "cell_format_version": "1.0.0",
  "id": "invoice_completeness_validator_v1",
  "name": "Invoice Completeness Validator",
  "version": "1.0.0",
  "task_role": "invoice_completeness_validation",
  "runner_compat": {
    "runner_api_version": "1.0.0",
    "execution_kind": "wasi"
  },
  "logic": {
    "module_path": "logic/logic.wasm",
    "entrypoint": "_start",
    "abi": "wasi_snapshot_preview1"
  },
  "determinism": {
    "mode": "strict",
    "seed": 0
  },
  "limits": {
    "max_wall_ms": 500,
    "max_steps": 1000000,
    "max_memory_mb": 64,
    "max_output_bytes": 32768,
    "max_tool_calls": 0
  },
  "contracts": {
    "input_schema_path": "schemas/input.schema.json",
    "output_schema_path": "schemas/output.schema.json"
  },
  "verifier": {
    "tests_path": "verifier/tests.json",
    "comparison": {
      "mode": "canonical_json"
    }
  },
  "capabilities": {
    "network": false,
    "filesystem": {
      "preopens": []
    },
    "env": []
  },
  "licensing": {
    "spdx": [
      "Apache-2.0"
    ],
    "notices_path": "licenses/THIRD_PARTY_NOTICES.txt"
  },
  "integrity": {
    "hash_algorithm": "sha256",
    "hash_manifest_path": "hashes/sha256.json"
  },
  "optional_attachments": {
    "knowledge_path": "knowledge/rules.json",
    "provenance_path": "provenance/build_info.json",
    "sbom_path": "sbom/spdx.json"
  }
}
```

This manifest matters for two reasons. First, it defines the execution boundary explicitly: contracts, limits, determinism, capabilities, and logic references are declared rather than assumed. Second, it gives the Runner the information needed to decide whether the artifact is even eligible for validation and execution.

A.5 Example Verifier Pack

The Verifier Pack is central to the Tasklet Cell model. A Cell is not valid merely because its bundle is well formed. It becomes valid only when the embedded verifier material passes under the target Runner and policy.

A representative verifier/tests.json may look like this:

```
{
  "version": "1.0.0",
  "comparison": {
    "mode": "canonical_json"
  },
  "cases": [
    {
      "name": "accepts_complete_invoice",
      "input_path": "verifier/cases/accept_complete/input.json",
      "expected_output_path": "verifier/cases/accept_complete/output.json"
    },
    {
      "name": "rejects_missing_invoice_number",
      "input_path": "verifier/cases/reject_missing_invoice_number/input.json",
      "expected_output_path": "verifier/cases/reject_missing_invoice_number/output.json"
    }
  ],
  "structural_requirements": [
    "manifest.json",
    "schemas/input.schema.json",
    "schemas/output.schema.json",
    "logic/logic.wasm",
    "hashes/sha256.json"
  ]
}
```

The exact internal layout of the Verifier Pack may vary by profile, but its architectural role does not. It is part of runtime validity, not merely part of development-time testing. A conventional package may ship with tests; a Tasklet Cell ships with verifier material that directly participates in the decision to allow or deny execution.

A.6 Runner Validation and Validity Decision

The bundle example in this appendix should be read together with the Runner's staged validity decision. In the current model, the Runner is expected to apply the following sequence:

1. *Bundle ingestion and structural validation.* Open the .cell artifact, normalize paths, reject ambiguous archive semantics, and confirm that required structural components are present and well formed.
2. *Manifest and contract validation.* Parse the manifest and verify that declared contracts, profile fields, and artifact references are coherent.
3. *Integrity and provenance checks.* Verify the content hashes for required bundle components and evaluate any profile-relevant provenance or inventory material.
4. *Verifier Pack execution.* Execute the Verifier Pack under the enforced runtime boundary. This is the decisive validity stage.
5. *Policy binding and execution preparation.* Bind the Cell to the local policy profile, including offline posture, capability restrictions, runtime bounds, and output rules.
6. *Execution enablement.* Only after the preceding checks pass may the logic payload execute against a schema-valid request.

This sequence matters because local validity is stricter than structural correctness. A syntactically correct bundle is not yet valid. An authentic or signed bundle is not yet valid. Even a structurally well-formed bundle with correct hashes is not yet valid. The Cell becomes executable only after it passes the full verification path under the target Runner and policy.

A.7 Integrity, Provenance, and Archive-Safety Expectations

The bundle format also carries archive-safety and integrity expectations. At a minimum, the Runner should reject bundles that show evidence of tampering, path escape, duplicate normalized paths, unsafe decompression behavior, or ambiguity in the interpretation of required artifact regions. In practice, this means the artifact model assumes:

- hash checking before execution;
- path normalization and zip-slip defense;
- refusal of ambiguous, shadowed, or duplicate archive paths;
- decompression limits and file-count limits;
- fail-closed handling when integrity or structure checks do not pass.

The model also allows optional provenance, signature, attestation, SBOM, and license material. These attachments can enrich deployment and audit decisions, but they do not bypass the local validity gate. A Cell may be authentic yet still invalid under local verification or local policy.

A.8 Interpretation

This appendix should be read as a concrete artifact example for the main paper's architectural claim. The important point is not that every future Tasklet Cell must use exactly the same optional paths or manifest extensions. The important point is that the Cell is packaged as a bounded, inspectable, verifier-backed local software artifact with explicit contracts, explicit logic references, explicit integrity metadata, and an execution decision mediated by the Runner.

In that sense, the bundle is the software embodiment of the paper's thesis. It is the unit that allows useful local intelligence to ship not as a remote service call or a broad agent loop, but as a verifiable offline artifact executed under a local enforcement boundary.

Appendix B. Gateway and Orchestrator Profile

This appendix defines the Gateway Profile and orchestrator posture used in the Tasklet Cell model. Its purpose is to make explicit how external coordination can be permitted without collapsing the paper's local trust boundary. The core architectural rule is strict: an external orchestrator is not trusted by default. Any coordination path that involves external request routing must remain mediated first by a strict local Gateway and then, beyond that Gateway, by the Runner as the final enforcement wall.

B.1 Purpose

The paper's main argument depends on bounded local validity and bounded local execution. The Gateway Profile is therefore optional and is needed only to the extent that coordination with external request sources can occur without undermining those constraints. The Gateway is not an alternate execution engine. It is a policy-enforcing mediator. Its function is to apply schema checks, allowlist checks, limit checks, and no-bypass controls before any Runner call is allowed.

B.2 Trust Model

The trust model is intentionally strict:

- the external orchestrator is untrusted;
- the Gateway is a deterministic mediator with allowlist and limit enforcement;
- the Runner remains the final enforcement wall for verifier checks, offline posture, resource bounds, capability controls, and any learning-gate rules.

This means that even an apparently well-formed request is not sufficient. The request must still pass Gateway policy, and Gateway approval does not remove Runner enforcement. No request path is allowed to bypass the same local validation, policy, and capability controls that apply to direct host execution. The orchestrator cannot promote a Cell to valid status, expand its authority boundary, or disable local safeguards. Trust therefore remains local, mediated, and fail-closed.

B.3 Request and Response Envelopes

All Gateway traffic is expected to use schema-validated envelopes.

A request envelope includes fields such as:

- request_id
- cell_id
- operation
- request_schema_id
- response_schema_id
- payload_size_bytes
- rate_token
- policy_hash
- input_json
- redacted_input_json
- consent_flag

A response envelope includes fields such as:

- request_id
- decision
- decision_reason
- applied_policy_hash
- runner_invoked
- result_ref

These envelope rules matter because they prevent the Gateway from becoming an informal message bridge. Instead, it becomes a deterministic policy surface with explicit, inspectable request and response structure. The Gateway should accept only the minimum context required by the Cell's declared contract, preserve offline-first policy, and retain any redacted form needed for replay, debugging, or audit.

B.4 Example Request and Response Envelopes

Representative request and response envelopes are shown below. The exact values are illustrative, but the field structure reflects the current model.

```
{
 "request_id": "req_20260227_0001",
 "cell_id": "invoice_completeness_validator_v1.cell",
 "operation": "run",
 "request_schema_id": "invoice_input_v1",
 "response_schema_id": "invoice_output_v1",
 "payload_size_bytes": 1842,
 "rate_token": "finance_desk_default",
 "policy_hash": "sha256:6d5f5f7a0c2f4fd8f9733f40fdac9e3a06b967f8fcb2c5a8c0d604af6a0438dd",
 "input_json": {
   "invoice_number": "INV-2041",
   "vendor_name": "Example GmbH",
   "invoice_date": "2026-02-27",
   "currency": "EUR",
   "total_amount": 1299.50
 },
 "redacted_input_json": {
   "invoice_number": "INV-2041",
   "vendor_name": "Example GmbH",
   "invoice_date": "2026-02-27",
   "currency": "EUR",
   "total_amount": 1299.50
 },
 "consent_flag": true
}
{
 "request_id": "req_20260227_0001",
 "decision": "allow",
 "decision_reason": "policy_and_limits_ok",
 "applied_policy_hash": "sha256:6d5f5f7a0c2f4fd8f9733f40fdac9e3a06b967f8fcb2c5a8c0d604af6a0438dd",
 "runner_invoked": true,
 "result_ref": "runner://exec/20260227/0001"
}
```

The important point is not the literal field values. The important point is that the Gateway Profile makes request identity, policy identity, decision state, and Runner invocation explicit rather than implicit.

B.5 Allowlist Policy Format

The Gateway must enforce a machine-readable allowlist policy before any Runner call occurs. The policy includes required fields such as:

- policy_version
- policy_hash
- allowed_cells
- allowed_ops
- limits

A representative example is shown below:

```
{
 "policy_version": "1.0.0",
 "policy_hash": "sha256:6d5f5f7a0c2f4fd8f9733f40fdac9e3a06b967f8fcb2c5a8c0d604af6a0438dd",
 "allowed_cells": [
   "finance_doc_extractor_neural_v1.cell",
   "invoice_completeness_validator_v1.cell",
   "vat_math_checker_v1.cell",
   "duplicate_detector_v1.cell"
 ],
 "allowed_ops": {
   "finance_doc_extractor_neural_v1.cell": ["validate", "test", "run"],
   "invoice_completeness_validator_v1.cell": ["validate", "test", "run"],
   "vat_math_checker_v1.cell": ["validate", "test", "run"],
   "duplicate_detector_v1.cell": ["validate", "test", "run"]
 },
 "limits": {
   "max_payload_bytes": 262144,
   "max_requests_per_minute": 60,
   "max_parallel_requests": 4
 }
}
```

The purpose of this structure is not merely operational control. It is also a form of architectural discipline. The Gateway should admit only known Cells, known operations, and known limit profiles. Anything outside that policy surface should fail closed.

B.6 Policy Hash and Deterministic Policy Identity

The Gateway Profile requires that policy_hash be computed over canonical JSON bytes. This ensures that policy identity is deterministic and resistant to superficial formatting differences. Any mismatch between the claimed policy_hash and the active local policy must cause the request to be rejected before Runner invocation.

This matters because a policy system is meaningful only if the policy being claimed is the policy actually being enforced. It also means that request-supplied policy claims are not a trust root. The authoritative policy remains the local policy selected by the deployment, and the Gateway uses the hash only to confirm that the request is aligned with that active policy.

B.7 Transport Profile

The default transport profile is local IPC, such as a Unix-domain socket or an equivalent local-only channel. Remote transport is not forbidden in principle, but it must preserve the same envelope and enforcement semantics. The paper's core trust boundary does not depend on whether a message originated remotely or locally. It depends on whether the request passes through the same bounded mediation path.

In practical terms, origin does not confer trust. A local caller is not automatically trusted merely because it resides on the same machine, and a remote caller does not gain a different execution path. The same policy, envelope, limit, and Runner requirements must apply in all cases.

B.8 Enforcement Order

The Gateway's enforcement order is a required part of the profile:

1. parse the request envelope;
2. validate the envelope schema;
3. validate policy_hash against the active local policy;
4. enforce allowlist checks for allowed_cells and allowed_ops;
5. enforce size, rate, and parallel limits;
6. forward only an admissible request to the Runner;
7. require the Runner to apply the same local validity path, including verifier checks, policy binding, capability controls, and runtime bounds.

This order matters because it prevents the Gateway from becoming a thin forwarder. The Gateway must act as a pre-Runner control surface rather than as a convenience pass-through, and it must not create an alternate fast path around local validation.

B.9 Non-Bypass Invariants

The Gateway Profile includes explicit no-bypass invariants:

- the Gateway cannot invoke execution paths that skip Runner verification;
- the Gateway cannot disable Runner offline or resource-bound controls;
- the Gateway cannot bypass learning-gate checks where such checks apply;
- the Gateway cannot elevate capabilities beyond the active allowlist policy;
- the Gateway cannot introduce unconstrained tool, shell, or host-surface access beyond explicitly allowed interfaces;
- the Gateway cannot treat a request as trusted merely because it originates from an orchestrator, scheduler, or host-side coordinator.

These invariants are essential because they preserve the paper's central claim that validity and execution remain local, enforced, and bounded. Without no-bypass guarantees, the Gateway would become a loophole through which the architecture could silently drift back toward ambient authority.

B.10 Finance Desk Example and Conformance Expectations

The project materials include a Finance Desk example in which the allowlisted Cells are limited to a small named set and only the operations validate, test, and run are allowed. The corresponding no-bypass conformance expectations include deterministic deny cases such as:

- a non-allowlisted cell_id;
- an allowlisted Cell paired with a disallowed operation;
- a request attempting to skip verifier or conformance state;
- a request attempting to disable offline or limit metadata;
- a malformed envelope with a forged policy_hash;
- a request that exceeds size, rate, or parallelism limits.

These examples matter because they show that the Gateway Profile is not merely a design aspiration. It is tied to explicit negative-case expectations that preserve fail-closed mediation behavior.

B.11 Interpretation

This appendix should be read as the coordination-layer complement to the paper's local Runner model. The Gateway Profile is optional, but when present it does not weaken the artifact boundary; it exists to preserve that boundary in the presence of external request sources. The orchestrator remains untrusted, the Gateway remains a deterministic mediator, and the Runner remains the final enforcement wall.

In that sense, the Gateway Profile helps the paper make a stronger claim: bounded local intelligence can participate in larger software systems without surrendering its validity rule, offline posture, or fail-closed execution model.

Appendix C. Reproducibility and Artifact Index

This appendix defines the reproducibility boundary and artifact index for the Tasklet Cells paper-evidence baseline. Its purpose is to tie the paper's empirical claims to a concrete repository state, a concrete evidence bundle, concrete reproduction entry points, and a bounded set of output artifacts that can be inspected directly.

A key methodological point should be stated at the outset. Historical appendix material tied to 533c63e should be read as baseline-failure context, not as the canonical current evaluation anchor. The canonical clean anchor for the present paper state is commit 78a1635 with evidence rooted at paper_evidence/2026-03-09/78a1635. For external review, the same clean evidence object is archived publicly at the GitHub release paper-evidence-r1-78a1635-clean in the Ahsadin/agif-tasklet-cell-public repository.

C.1 Purpose

The paper makes executable-evidence claims rather than relying only on narrative description. For that reason, the empirical sections need a clear artifact index that answers the following questions:

- which repository baseline was used;
- which evidence bundle should be treated as authoritative;
- which commands reproduce the evaluation entry points;
- which output files are the main citation-oriented artifacts;
- which suites and metrics define the empirical claim surface;
- and which limits or known gaps remain visible in the evidence itself.

This appendix provides that index.

C.2 Evaluated Baseline and Source of Truth

The Tasklet Cells evidence baseline is tied to the following recorded identity:

- clean detached worktree root recorded in env.json: omitted from publication copy; recorded as a clean detached worktree in env.json
- branch: HEAD
- baseline commit SHA: 78a163503ff570407c8c34065a5500c0e34e50d0
- short SHA: 78a1635
- recorded capture time in env.json: 2026-03-09T16:11:13Z
- recorded summary generation time in summary.md and summary.json: 2026-03-09T17:22:49Z

This baseline matters because the paper's empirical claims are not intended to float across arbitrary repository states. They are tied to a specific evaluation bundle and a specific run identity.

At the same time, the recorded environment explicitly reports:

- dirty = false
- dirty_file_count = 0

This matters because the nominal commit hash and the evaluated artifact state now align cleanly for the canonical paper anchor. The older 533c63e baseline remains relevant only as historical failure context, not as the current source of truth.

C.3 Recorded Execution Environment and Runtime Identity

The recorded execution environment for the clean evidence bundle is as follows:

- platform: macOS-26.3.1-arm64-arm-64bit-Mach-O
- CPU: Apple M4
- RAM: 16 GB
- Python: Python 3.14.3
- Rust: rustc 1.93.1 (01f6ddf75 2026-02-11)
- Cargo: cargo 1.93.1 (083ac5135 2025-12-15)
- Runner Cell SHA-256: 3168964af8d9fcf32947606cfede8c38ebdb1e73c5e820ccfb03fe5895c19bd5

The reproducibility notes also record the following:

- expected runtime for the captured run: approximately 71.6 minutes
- recorded aggregate runtime in summary.json: 4295.756256332999 seconds
- internet required: no (all suites run offline and locally)

This environment record matters because the paper's architectural posture depends on local, offline evaluation rather than on cloud-dependent testing or opaque hosted benchmarks.

C.4 Reproduction Entry Points and Repetition Boundary

The public citation and inspection entry points for the evidence bundle are:

```
https://github.com/Ahsadin/agif-tasklet-cell-public/releases/tag/paper-
evidence-r1-78a1635-clean
# or
https://github.com/Ahsadin/agif-tasklet-cell-public
```

The released clean anchor records 30 repetitions. That repeat count is captured explicitly in summary.md and summary.json and is part of the public evidence record.

These public locators matter because they define the stable entry surface that external reviewers can inspect directly. They should be read as the public citation and audit surface for the evidence object, not as proof that the portable ZIP is a self-contained rerun environment.

C.5 Evidence-Pack Schema and Canonical Path

The released evidence bundle records the following package identity in summary.json:

- schema: agif_paper_evidence_pack_v1
- output directory: paper_evidence/2026-03-09/78a1635
- repetitions: 30

The canonical evidence path recorded by the bundle is therefore:

paper_evidence/2026-03-09/78a1635
Public archival release page:
https://github.com/Ahsadin/agif-tasklet-cell-public/releases/tag/paper-evidence-r1-78a1635-clean
Public portable archive asset name:

agif-paper-evidence-r1-78a1635-clean-n30-portable.zip
For external review and citation, the public release page above should be treated as the stable public locator, and the named portable ZIP should be treated as the canonical downloadable archive rather than relying on the repo-local path alone.

The released archive contains a simplified top-level directory rooted at:

78a1635/

This matters because the archive root, the recorded output directory, and the schema marker together define the paper-facing evidence object more precisely than the commit hash alone.

C.6 Top-Level Artifacts and Raw Layout

Within the released archive directory, the principal top-level artifacts are:

- env.json
- summary.json
- summary.md
- paper_table.md
- reproduce.md
- figures/
- raw/

The figures/ directory includes the following paper-oriented graphics:

- determinism_stability.png
- size_breakdown.png
- latency_distribution.png

These files serve different roles:

- env.json records the captured execution environment and repository state;
- summary.json records the machine-readable aggregate results and package metadata;
- summary.md records the human-readable outcome summary;
- paper_table.md provides citation-oriented tabular results for the paper;
- reproduce.md records the reproduction notes and verification commands;
- figures/ contains paper-oriented visual outputs;
- and raw/ preserves per-suite and per-repeat artifacts for audit and inspection.

The raw tree is not merely a placeholder directory. It contains concrete repeat-level artifacts such as report.json, stdout.json, stderr.log, and suite-specific working material. For example, the release-readiness sweep includes repeat-level readiness reports, the fail-closed benchmark preserves repeat reports and logs, and the A6 benchmark preserves exported model state, quantization metadata, snapshot manifests, and temporary rollback or conformance state under its work/ and tmp/ directories.

C.7 Captured Bundle Metadata

The released summary.json also captures bundle-level metadata for the evaluated Cell artifact:

- bundle path: cells/finance_doc_extractor_neural_v6.cell

- bundle archive size: 369,595 bytes
- total component bytes: 955,161 bytes

The recorded component breakdown includes:

- logic_wasm = 952,014 bytes
- manifest_json = 1,157 bytes
- hashes = 684 bytes
- model_dir = 507 bytes
- external_runtime_model_bin = 256 bytes
- schemas = 232 bytes
- verifier_pack = 191 bytes
- licenses = 120 bytes
- other = 0 bytes

This metadata matters because it makes the appendix more than a run log. It also records the evaluated artifact surface of the Tasklet Cell bundle itself.

C.8 Recorded Suites and Evaluation Volume

The evidence bundle records five principal evaluation suites:

- FINAL_RELEASE_READINESS_SWEEP
- review_hallucination_bench_v1
- a6_extractor_benchmark_v1
- runner_fail_closed_bench_v1
- reasoning_trace_bench_v1

The released run executed 30 repeats of these suites. In the paper's evaluation methodology, this corresponds to 2400 executed evaluation units in total:

- 600 release-readiness checks (20 × 30)
- 120 hallucination-safety cases (4 × 30)
- 180 extraction-quality cases (6 × 30)
- 1140 fail-closed negative cases (38 × 30)
- 360 reasoning-trace cases (12 × 30)

This matters because the paper is not claiming a vague benchmark posture. It is claiming evidence from a specific, repeated suite set with a clear workload surface.

C.9 Recorded Outcome Summary

The released clean anchor records the following top-level suite outcomes:

- FINAL_RELEASE_READINESS_SWEEP = pass
- review_hallucination_bench_v1 = pass
- runner_fail_closed_bench_v1 = pass
- reasoning_trace_bench_v1 = pass
- a6_extractor_benchmark_v1 = pass

The corresponding recorded summary metrics include:

- FINAL_RELEASE_READINESS_SWEEP: 30/30 passed; determinism_rate = 1.0; target_check_count = 20; final_repeat_passed_count = 20
- review_hallucination_bench_v1: 30/30 passed; unsafe_allow_count = 0; abstain_accuracy = 1.0; reason_accuracy = 1.0; hallucination_resistance_score = 1.0
- runner_fail_closed_bench_v1: 30/30 passed; rejection_rate = 1.0; unsafe_allow_count = 0; expected_reason_match_rate = 1.0
- reasoning_trace_bench_v1: 30/30 passed; trace_schema_valid_rate = 1.0; trace_determinism_rate = 1.0; evidence_alignment_rate = 1.0
- a6_extractor_benchmark_v1: 30/30 passed; routing_accuracy = 1.0; vendor_hit_rate = 1.0; currency_accuracy = 1.0; grand_total_mae = 0.0; tax_total_mae = 0.0; subtotal_mae = 0.0; max_arithmetic_error = 0.0; numeric_grounding_rate = 1.0; abstain_fail_rate = 0.0

This summary is important because it preserves the paper's bounded-evidence posture without relying on stale baseline text. The clean anchor supports strong claims in several areas, and it keeps the passing A6 status visible together with the still-bounded claim surface.

C.10 Known Gaps and Interpretation Boundary

The clean evidence bundle records no explicit known gaps:

- known_gaps = []
- all tracked suites passed across all repeats in the clean N30 anchor.

This no-gap result is visible in both summary.json and summary.md. That visibility matters because the paper's methodology is designed to expose the current state directly rather than preserve outdated failure text once the clean anchor has replaced the baseline.

A second methodological boundary is also important. The recorded determinism metrics in the bundle are strongest at the level of evaluated outputs and suite results. They should not automatically be interpreted as proof that every derived intermediate artifact inside every working directory is bit-identical across repeats. The paper should therefore describe the current evidence as strongest on repeated output-level determinism rather than on universal artifact-level reproducibility.

C.11 Verify Output Hashes

The release bundle also records a direct hash-check procedure for the main evidence outputs:

```
cd paper_evidence/2026-03-09/78a1635
python3 - <<'PY'
import hashlib, json
from pathlib import Path
for rel in ['env.json','summary.json','summary.md','paper_table.md','reproduce.md']:
   p = Path(rel)
   print(rel, hashlib.sha256(p.read_bytes()).hexdigest())
PY
```

This step matters because it provides a lightweight integrity check for the citation-oriented outputs. It does not, by itself, prove full supply-chain reproducibility, but it does support stable output inspection, comparison, and citation.

C.12 Interpretation

This appendix should be read as the paper’s reproducibility map for the released clean evidence anchor.

In that sense, the appendix strengthens the paper’s methodological claim. The evidence is not merely described. It is indexed, inspectable, repeatable within a stated boundary, and explicit about both what now passes cleanly and what remains bounded by scope.

Appendix D. Threats and Mitigations Table

This appendix summarizes the principal threat classes assumed by the Tasklet Cell model and the corresponding mitigation mechanisms built into the artifact, Gateway, and Runner boundary. Its purpose is not to restate the full security discussion from the main text, but to provide a compact mapping from threat vectors to concrete defensive controls.

D.1 Purpose

The Tasklet Cell architecture makes a narrow trust claim: useful local AI capabilities can be embedded as bounded software artifacts if validity, authority, integrity, and execution are all mediated by explicit local controls. A threat-to-mitigation table is useful because it makes that claim concrete. It shows which classes of failure or attack the architecture is designed to resist, where those defenses live in the system, and which parts of that posture are supported by the released evidence bundle. Table 6 provides the appendix-level mapping.

D.2 Threat-to-Mitigation Mapping

Table 6

Threat-to-Mitigation Mapping

Threat vector	Mitigation mechanism	Primary enforcement layer
Bundle tampering or substitution	Required SHA-256 hash-manifest verification before load; fail-closed rejection on mismatch	Runner
Provenance, attachment, or inventory mismatch	Provenance and attachment checks under local policy; invalid or mismatched bundle metadata does not confer trust	Runner / local policy
Structurally unsafe or ambiguous bundles	Structural validation, path normalization, and rejection of malformed, duplicate, or ambiguous archive paths	Runner
Archive path escape (zip-slip)	Path normalization and refusal of paths that escape the bundle root	Runner
Zip bombs or decompression abuse	Maximum total uncompressed size, per-file size, file-count limits, and rejection based on suspicious compression ratios	Runner
Schema-invalid input	Input validation before execution; deterministic refusal on contract mismatch	Runner
Schema-invalid output	Output validation after execution; deterministic refusal on contract mismatch	Runner
Verifier Pack failure	Conformance execution of built-in verifier cases; the artifact remains invalid unless all required checks pass	Runner

Licensing metadata or notice failure	Manifest and notice validation; licensing-invalid bundles fail closed rather than executing under degraded trust assumptions	Runner
Policy mismatch between artifact assumptions and local rules	Local policy binding after structural, integrity, and verifier checks; execution is denied when the profile does not match	Runner / local policy
SBOM or attestation policy violation	Local policy may require inventory or attestation material and reject bundles that do not satisfy the active profile	Runner / local policy
Rollback or downgrade-state abuse	Rollback-state validation and strict-mode rejection of invalid rollback fixtures or state transitions	Runner / local policy
Unbounded runtime behavior	Hard limits on wall-clock time, fuel or steps, memory, output size, and tool calls	Runner
Invalid runtime policy preset	Deterministic rejection of unsupported or malformed runtime-policy selections	Runner
Invalid runtime input payload	Pre-execution parsing and input validation before the logic payload is allowed to run	Runner
Data exfiltration or phishing	Offline-by-default execution; no ambient network capability and no undeclared communication surface	Runner sandbox / local policy
Prompt injection or jailbreak pressure	Strict input and output contracts, bounded logic, and refusal of unconstrained execution behavior	Runner plus Cell contract boundary
Unauthorized tool or host access	Minimal capability exposure; no ambient host authority and no undeclared file, process, or network access	Runner
Tool amplification or ambient shell abuse	Allow-listed explicit tools only; no unconstrained shell or host authority	Gateway plus Runner policy
Untrusted orchestrator requests	Strict Gateway mediation, allowlist enforcement, and no-bypass invariants before Runner invocation	Gateway
Non-allowlisted Cell or operation	Deterministic deny decision before Runner invocation	Gateway
Policy spoofing or forged policy identity	Canonical policy_hash checking against the active Gateway policy	Gateway
Attempt to bypass verifier, offline, or capability controls	No alternate fast path; the same local validation and policy path must apply regardless of request origin	Gateway plus Runner

Silent trust-anchor substitution	Local trust anchors remain authoritative rather than artifact-supplied trust declarations	Local policy / Runner
Manifest-supplied trust-reference abuse	Manifest key paths, certificate references, and similar fields are metadata only unless independently trusted by local policy	Runner / local trust policy
Hidden personalization drift	The Personal Layer remains local, bounded, inspectable, and separate from the immutable core artifact	Learning Gate plus Runner
Learning-layer regression or catastrophic forgetting	Personal-Layer changes must pass regression checks and remain reversible through rollback	Learning Gate plus Runner

D.3 Threat Scope

The architecture is designed around a bounded threat surface. The assumed threats include, at a minimum:

- bundle tampering or mismatch between declared and actual contents;
- provenance, supplier, or attachment mismatch at the bundle boundary;
- structurally unsafe or ambiguous bundles;
- invalid or adversarial input that attempts to cross contract boundaries;
- unauthorized capability use, including unintended tool or host access;
- policy mismatch between artifact assumptions and local execution rules;
- licensing, attestation, or inventory-policy violations under stricter local profiles;
- excessive resource consumption or runtime expansion beyond declared limits;
- hidden drift through uncontrolled personalization or adaptation;
- and indirect trust escalation through external orchestrators or helper layers.

This threat scope is deliberately narrower than the threat model for a general-purpose autonomous agent system. The architecture's defensive strategy depends on keeping the artifact boundary narrow and the authority surface explicit.

D.4 Runner-Centric Security Posture

The Runner is the core enforcement wall for several reasons:

- it validates the artifact before execution;
- it enforces offline and capability restrictions;
- it enforces resource bounds;
- it runs the Verifier Pack;
- it validates input and output contracts;
- and it fails closed on structural, integrity, policy, or runtime violations.

This means the Runner is not merely a launcher. It is the main mechanism by which the artifact remains a bounded and inspectable software component rather than an opaque process with ambient authority.

D.5 Gateway-Specific Security Role

The Gateway exists to mediate external coordination without undermining the local trust boundary. Its security role is narrower than the Runner's but still essential. The Gateway enforces:

- schema-validated request and response envelopes;
- allow-listed Cells and operations;
- rate, size, and parallelism limits;
- policy-hash consistency;
- and no-bypass control over Runner invocation.

The Gateway therefore protects the system from unsafe orchestration patterns, but it does not replace the Runner's local validity checks. The orchestrator may request work, but it does not define the trust boundary.

D.6 Evidence-Backed Negative-Case Coverage

The clean evidence bundle anchored at 78a1635 gives this appendix a concrete empirical boundary. The runner_fail_closed_bench_v1 suite executes 38 negative cases per repeat across 12 categories: hash tampering, schema, verifier, archive paths, limits, licensing, integrity metadata, policy, rollback, runtime limits, runtime policy, and runtime input. Across 30 repeats, this corresponds to 1,140 negative executions.

The recorded aggregate outcome is strong but should still be interpreted narrowly:

- pass rate: 100.00% (30/30 repeats)
- rejection rate: 1.0
- unsafe allow count: 0
- expected reason-match rate: 1.0

The suite therefore supports the claim that the Runner rejects malformed or adversarial bundles with explicit failure modes rather than allowing degraded or best-effort execution. The evidence is especially relevant because it covers not only simple hash tampering, but also verifier failure, archive-path abuse, bundle-size and compression abuse, policy violations, rollback rejection, invalid runtime-policy presets, and malformed runtime input.

This evidence should still be framed carefully. As discussed in Section 14.4, it supports complete rejection on the provided malformed and adversarial bundle set under repeated execution. It does not prove resistance against every possible malicious input, every future attack class, or every host-compromise scenario.

D.7 Local Trust Anchors and Trust-Reference Correction

The architecture also treats trust-anchor handling as a security issue in its own right. If a deployment profile supports signatures, attestations, manual replacement, or policy-managed updates, the local trust anchor must come from local trust policy rather than from the artifact's own declaration of where trust should be found.

The intended mitigation posture is therefore:

- locally trusted pinned keys, or equivalent locally governed trust anchors, remain authoritative;
- manifest-supplied trust references may be carried as metadata, but they do not define the trust root;
- artifact metadata may assist inspection, but it cannot self-authorize the bundle;
- and update or replacement workflows must not create a bypass around verifier-mediated validity.

This distinction matters because an artifact that can declare its own trust anchor is uncomfortably close to self-authorization.

D.8 Learning and Personalization Risks

The architecture also recognizes that a local personalization layer can become a risk if it changes behavior without control. For that reason, the paper's model treats personalization as gated, local, and reversible rather than as unrestricted self-modification.

The intended mitigation posture is:

- the core Cell remains immutable;
- Personal-Layer changes do not silently rewrite the core logic;
- regression checks are required before accepted changes remain active;
- and rollback remains possible.

This matters because the architecture's boundedness claim would weaken significantly if local learning were allowed to erase the verifier-backed boundary between fixed artifact logic and user-specific adaptation.

D.9 Interpretation

This appendix should be read as a compact trust map for the Tasklet Cell architecture, synthesized from the main-text security model and the clean negative-case evidence. It does not claim that every implementation defect is already solved, nor that every runtime environment provides identical security guarantees. Rather, it shows that the architecture is built around recognizable defensive controls: integrity verification, structural validation, verifier-mediated validity, bounded execution, offline enforcement, allowlist mediation, no-bypass policy, local trust-anchor control, and reversible personalization.

In that sense, the table supports one of the paper's central arguments: bounded local AI can be treated more like serious software when its threat surface is explicit and its defenses are tied to concrete enforcement layers rather than to informal, best-effort conventions.

Appendix E. CellPOS Case-Study Baseline Repository Record

This appendix records the repository baseline used as the primary repository-level evidence that CellPOS was built into an offline product package. Its purpose is not to restate the full case-study narrative from the main paper, but to anchor the CellPOS claims to a concrete repository, a frozen release baseline, a packaged artifact record, a verification surface, and a bounded set of repository documents that function as source-of-truth evidence.

E.1 Repository Identity

The baseline repository is as follows:

- *Repository name:* CellPOS
- *Private GitHub repository record: Ahsadin/CellPOS*
- *Local repository path used in the recorded build environment: omitted from publication copy; use the private GitHub repository record below*
- *Active branch at proof time:* main
- *Current repository head at report time:* 4cadacf5c9f0db5ca097fc1baabd1f5408f8ccb7

This repository record matters because the CellPOS case study sits inside the paper's evidence boundary. It is not presented as a hypothetical product sketch. Instead, it is tied to a specific repository, a specific productization state, and a specific released pilot baseline.

E.2 Frozen Private-Pilot Baseline and Commit Map

The frozen private-pilot baseline is the principal repository-level proof point for the completed build.

- *Baseline tag:* cellpos-private-pilot-v0.1.0
- *Baseline commit:* 904de056207eebdb24fc820d8f409c2e4323c0eb
- *Private GitHub release record: cellpos-private-pilot-v0.1.0*

This baseline matters because it ties the CellPOS private-pilot claim to a fixed release marker rather than to an open-ended moving branch. In other words, the case study is anchored to a frozen repository state rather than merely to the current contents of main.

The recorded milestone commits also clarify the proof path:

- de1a4ab2fb1fdc2195e1f9837a7dd3702b8bf7a5 marks PROD-008 clean-room acceptance completion.
- 904de056207eebdb24fc820d8f409c2e4323c0eb freezes the private-pilot baseline tied to the release tag.
- bbc247f746164962de7c81209eb6868a890fa3de adds the GitHub-facing root README.md baseline overview.
- 4cadacf5c9f0db5ca097fc1baabd1f5408f8ccb7 records later offline trust-audit and verification hardening.

The proof record therefore preserves an important provenance note: later commits exist after the frozen pilot baseline, but they do not replace the pilot proof point. They improve presentation, trust-audit clarity, and verification hygiene while leaving the tagged pilot baseline intact.

E.3 Productization Status at Baseline

At the frozen baseline, the repository recorded the following final productization state:

- *Productization status:* 8/8 complete
- *Final readiness classification:* Ready for private pilot

This means that the repository evidence records completion of the full productization sequence labeled PROD-001 through PROD-008, including:

- packaged product build generation;
- offline licensing;
- offline signed manual update support;
- clean-room packaged acceptance; and
- private-pilot handoff readiness.

The supporting tracker material also records that no productization issues remained in PROD-001..PROD-008, and that the broader dashboard verdict was HISTORICAL FOUNDATION COMPLETE + SELLABLE PRODUCT READY FOR PRIVATE PILOT.

This status should be interpreted carefully. It is strong enough to support a *Ready for private pilot* claim. It is not, by itself, a claim of broad commercial rollout, universal field validation, or complete production hardening in every operational dimension.

E.4 GitHub Publication Proof

The GitHub-facing baseline was not left as a local-only repository state. The proof record shows that the pilot baseline was published as a released GitHub artifact.

- *Release title:* CellPOS Private Pilot v0.1.0
- *Release status:* published
- *Release asset attached:* CellPOS-0.1.0-prod007.0.tar.gz

This matters because it shows that the packaged artifact was tied to a frozen release rather than existing only as an internal output file.

E.5 Packaged Artifact Record

The baseline repository does not merely claim that a product package exists. It records a concrete packaged artifact in the repository outputs:

- *Repository-recorded artifact path omitted from publication copy*
- *Artifact name:* CellPOS-0.1.0-prod007.0.tar.gz
- *Artifact SHA-256:* 80f31c72d7054f34a7e802f3ed324bb9196bcc5e1202be0148bca121e06402dd
- *Recorded size:* 28,359,248 bytes

The packaged artifact represents one offline product containing both local subsystems:

- CellPOS Core
- Embedded AGIF Brain

The documented packaged layout includes top-level directories such as:

- app/
- bin/
- config/
- product/
- site-data/

The documented packaged lifecycle scripts include:

- bin/start-cellpos.sh
- bin/import-license.sh
- bin/apply-update.sh

These details matter because they show that the proof baseline is tied to a real, versioned distributable artifact rather than to source code alone.

E.6 Repository Proof Surface and Verification Record

The repository proof surface does not stop at the existence of the packaged archive. The recorded proof materials tie the frozen baseline to all of the following:

- packaged offline product output;
- offline local license validation;
- offline signed manual update support;
- packaged startup from the distributable rather than from the development tree;
- browser rendering for /pos, /kds, /order, and /admin;
- embedded AGIF child-process startup and health checks;
- clean-room packaged acceptance from an isolated install layout; and
- security-hygiene checks confirming that shipped bundles exclude private signing keys.

The repository also records an explicit executable verification surface. The core final-stage commands are:

```
npm run test:prod-007-packaging-license-updates
npm run ci:productization-prod-007
npm run test:prod-008-clean-machine-acceptance
npm run ci:productization-prod-008
```

Additional supporting commands preserved in the proof record include:

```
npm run test:productization-prod-006
npm run test:ui-agif-final-closure-consistency
npm run test:prod-003-real-runner
npm run ci:productization-prod-001
```

The final lifecycle and acceptance stages also record explicit pass tokens:

- PROD-007_PACKAGING_LICENSE_UPDATES_PASS
- PROD-008_CLEAN_MACHINE_ACCEPTANCE_PASS

Taken together, this gives the repository baseline more weight than a plain release note or a source-code snapshot. It anchors the case study to a concrete product state with a documented packaged lifecycle and an executable proof surface.

E.7 Source-of-Truth Repository Documents

The primary repository documents supporting the baseline are:

- README.md
- docs/50_AGIF_CELLPOS_PRODUCTIZATION.md
- docs/60_PRIVATE_PILOT_HANDOFF.md
- docs/61_PRIVATE_PILOT_RELEASE_NOTES.md
- docs/62_PRIVATE_PILOT_OPEN_ITEMS.md
- docs/63_CELLPOS_BUILD_PROOF_REPORT.md
- docs/SNAPSHOTS/SNAPSHOT_2026-03-06_PROD-007.md
- docs/SNAPSHOTS/SNAPSHOT_2026-03-06_PROD-008.md
- docs/ACCEPTANCE/PROD_008_PILOT_WALKTHROUGH.md
- private productization project record: PROJECT_README.md
- private productization project record: CHANGELOG.md
- private productization project record: DECISIONS.md
- docs/00_MASTER_COMPLETION_DASHBOARD.md
- docs/ISSUES_INDEX.md

These documents matter because they provide the paper's repository-level evidence trail for productization, packaging, offline licensing, signed updates, acceptance, handoff status, tracker consistency, and remaining open items. They should therefore be treated as the core source documents for interpreting the CellPOS private-pilot baseline.

E.8 Evidentiary Interpretation, Known Limits, and Post-Pilot Backlog

This repository record shows that CellPOS was not left at the planning stage. A tagged and published baseline exists; that baseline is tied to concrete productization evidence, a packaged build output, acceptance documentation, tracker closure, verification commands, pass tokens, and a published GitHub release record. The repository record is therefore strong enough to support the paper's use of CellPOS as a private-pilot case study grounded in an actual product baseline.

At the same time, this appendix should be read within the paper's broader scope discipline. A repository baseline is evidence of a real packaged product state. It is not, by itself, evidence of broad production deployment or large-scale field validation. The same proof record explicitly notes that:

- no production-secure signing infrastructure is claimed yet;
- no second physical-machine proof is yet recorded in the repository;
- no fresh-OS proof is yet recorded in the repository; and
- the final state is Ready for private pilot, not broad commercial release.

The proof record also preserves a post-productization backlog that does not reopen PROD-001..PROD-008. Instead, it defines the next operational steps after the private-pilot baseline:

- production signing and key-management hardening;
- second-machine or fresh-OS validation;
- pilot install and support SOP creation;

- hardware and device-matrix validation; and
- broader commercial-release hardening.

Its role is therefore narrower and more precise: it provides the repository-level anchor that makes the CellPOS case study auditable, citable, and reproducible within the boundaries claimed by the paper.

Appendix F. CellPOS Packaged Artifact and Lifecycle Record

This appendix records the packaged product artifact and packaged lifecycle surface produced for the CellPOS private-pilot baseline. Its purpose is to document the repository-level artifact record used as evidence that CellPOS was built into an offline product package rather than remaining a source-only project. Independent inspection of the packaged archive itself depends on access to the recorded tarball.

F.1 Artifact Identity

The packaged artifact is identified as follows:

- *Artifact name:* CellPOS-0.1.0-prod007.0.tar.gz
- *Artifact version:* 0.1.0-prod007.0
- *Repository-recorded artifact path omitted from publication copy*
- *SHA-256:* 80f31c72d7054f34a7e802f3ed324bb9196bcc5e1202be0148bca121e06402dd
- *Recorded size:* 28,359,248 bytes

These fields matter because they tie the CellPOS case study to a concrete, versioned, and hash-identified offline product artifact.

F.2 Frozen Baseline and Distribution Linkage

The packaged artifact is tied to the frozen private-pilot baseline rather than to a moving repository state.

- *Private GitHub repository record: Ahsadin/CellPOS*
- *Baseline tag:* cellpos-private-pilot-v0.1.0
- *Baseline commit:* 904de056207eebdb24fc820d8f409c2e4323c0eb
- *Private GitHub release record: cellpos-private-pilot-v0.1.0*
- *Release title:* CellPOS Private Pilot v0.1.0
- *Release asset attached:* CellPOS-0.1.0-prod007.0.tar.gz

This linkage matters because it shows that the packaged artifact was not merely built locally. It was tied to a frozen pilot baseline, attached to a published GitHub release record, and used as the repository-level anchor for the private-pilot claim.

F.3 Product Composition

The packaged artifact represents a single offline product package containing both local subsystems:

- CellPOS Core
- Embedded AGIF Brain

The package was designed as a runnable offline bundle rather than as a source-only handoff. In that sense, it is evidence of productization at the artifact level, not only at the repository-documentation level.

F.4 Documented Package Layout

The recorded package layout includes the following top-level paths:

- app/
- bin/
- config/
- product/
- site-data/

These paths provide the packaged runtime, configuration, product metadata, startup scripts, and local data areas needed for offline execution.

F.5 Documented Lifecycle Scripts

The packaged bundle includes lifecycle scripts for startup and local operations:

- bin/start-cellpos.sh
- bin/import-license.sh
- bin/apply-update.sh

These scripts provide packaged startup, offline local license import, and offline signed update application.

F.6 Offline License Lifecycle

The packaged lifecycle includes a local signed-license flow.

The repository evidence records that:

- the license artifact is local and signed;
- validation is local only;
- bundled public trust material is used;
- no cloud dependency is required for license checking;
- missing, invalid, or expired license state is fail-closed for protected packaged use; and
- /admin remains available to show truthful license state and diagnostics.

The documented packaged import path is:

```
<INSTALL_ROOT>/bin/import-license.sh <license-file>
```

This matters because licensing is not treated as an external service dependency. It is part of the product's offline packaged lifecycle.

F.7 Offline Signed Update Lifecycle

The packaged lifecycle also includes a local signed-update flow.

The repository evidence records that:

- update packages include version metadata;
- signature verification occurs before application;
- payload-hash verification occurs before replacement;
- invalid update packages are rejected before partial application; and
- update results remain visible in /admin.

The documented packaged apply path is:

<INSTALL_ROOT>/bin/apply-update.sh <update-package>

This matters because the update path is designed as an offline, operator-controlled lifecycle action rather than as a cloud-driven background mechanism.

F.8 Admin Lifecycle Visibility

The packaged browser surface at /admin was extended to show the packaged lifecycle truthfully in HTML. The recorded lifecycle visibility includes:

- product version;
- bundle identity;
- embedded AGIF runtime identity;
- offline license state;
- license summary;
- update state;
- last update operation; and
- fail-closed reasons when activation or update verification is blocked.

This matters because the packaged lifecycle is not hidden behind scripts alone. It remains operator-visible through the product's own admin surface.

F.9 Clean-Room Install and Acceptance Walkthrough

The proof record goes beyond stating that the archive exists. It also documents a packaged clean-room lifecycle from isolated install targets outside the repository tree.

Two paths appear in the proof materials for two closely related purposes:

- *Manual walkthrough install target:* <isolated-install-root>
- *Automated clean-room acceptance root: <automated-clean-room-root>*

The first path belongs to the documented human walkthrough. The second path belongs to the recorded automated acceptance sweep. Both matter because they show that the product was exercised from installed packaged layouts rather than from the source tree itself.

The documented clean-room lifecycle consists of:

1. installing the bundle into an isolated target path;
2. starting the product from <INSTALL_ROOT>/bin/start-cellpos.sh;
3. opening the packaged browser surfaces at /pos, /kds, /order, and /admin;
4. activating the product through the packaged license-import script;
5. applying a signed update through the packaged update script; and
6. restarting the installed product and confirming that activation state, update state, and /admin truthfulness persist.

This matters because it moves the proof boundary beyond "the package exists" to "the package works from a clean installed location under its packaged lifecycle."

F.10 Verification Surface and Recorded Pass Tokens

The packaged lifecycle is also tied to an explicit verification surface rather than to prose alone. The core final-stage verification commands recorded in the repository are:

```
npm run test:prod-007-packaging-license-updates
npm run ci:productization-prod-007
npm run test:prod-008-clean-machine-acceptance
npm run ci:productization-prod-008
```

Additional supporting commands recorded in the proof surface include:

```
npm run test:productization-prod-006
npm run test:ui-agif-final-closure-consistency
npm run test:prod-003-real-runner
npm run ci:productization-prod-001
```

The final lifecycle and acceptance stages also record explicit pass tokens:

- PROD-007_PACKAGING_LICENSE_UPDATES_PASS
- PROD-008_CLEAN_MACHINE_ACCEPTANCE_PASS

These details matter because they show that the packaged artifact and lifecycle were exercised through recorded verification commands rather than accepted only on the basis of static documentation.

F.11 Security Hygiene and Trust Material

The packaged lifecycle record also includes a security-hygiene claim with a carefully bounded scope.

The evidence states that:

- the shipped bundle includes only the public trust material required for local verification;
- packaged trust material is clearly labeled non-production where applicable;
- repository-local private signing keys remain outside the distributable;
- the shipped bundle includes only config/trusted-keys/non-production-license-public.pem and config/trusted-keys/non-production-update-public.pem; and
- the repository does not claim production-secure signing infrastructure.

The later trust-audit record also states that packaged verification resolves trust from local trusted public-key paths, does not accept attacker-supplied signing material merely by reusing expected key identifiers, and preserves private-key exclusion checks for shipped bundles.

This means the lifecycle claim is strong enough to support a private-pilot offline product baseline while still preserving honest limits around production hardening.

F.12 Packaging Significance and Acceptance Use

The existence of this artifact is important because it demonstrates that CellPOS was turned into a distributable product bundle. The proof is not limited to repository code or documentation. A real versioned archive was created, recorded, published, and then used during later verification and acceptance phases.

The build-proof record explicitly ties this artifact to:

- packaged offline product output;
- clean-room packaged acceptance from an isolated install layout;
- startup from the packaged artifact rather than from a development tree;
- browser rendering for /pos, /kds, /order, and /admin;
- embedded AGIF child-process startup and health checks;
- offline local license activation;
- offline signed manual update behavior;
- restart persistence of activated state and update state; and
- private-pilot release publication.

F.13 Supporting Evidence

The artifact and lifecycle record are supported by the following repository evidence:

- README.md
- docs/60_PRIVATE_PILOT_HANDOFF.md
- docs/61_PRIVATE_PILOT_RELEASE_NOTES.md
- docs/63_CELLPOS_BUILD_PROOF_REPORT.md
- docs/SNAPSHOTS/SNAPSHOT_2026-03-06_PROD-007.md
- docs/SNAPSHOTS/SNAPSHOT_2026-03-06_PROD-008.md
- docs/ACCEPTANCE/PROD_008_PILOT_WALKTHROUGH.md
- private productization project record: PROJECT_README.md
- private productization project record: CHANGELOG.md

These documents matter because they tie the packaged artifact and lifecycle claims to concrete repository records rather than to narrative description alone.

F.14 Interpretation and Scope Limits

This artifact record shows that CellPOS reached a packageable offline product state. The product was built into a versioned archive with a defined structure, runnable scripts, release linkage, verifiable metadata, a documented clean-room lifecycle, and an executable verification surface suitable for private-pilot distribution.

That claim should still be read within the scope limits stated elsewhere in the paper. A packaged artifact and lifecycle surface are evidence of a real offline product baseline. They are not, by themselves, evidence of broad field deployment or universal production maturity. Their role is narrower and more precise: they provide the packaged-artifact and packaged-lifecycle anchor that makes the CellPOS case study concrete, auditable, citable, and reproducible within the paper's stated boundary.

Appendix G. CellPOS Verification and Acceptance Record

This appendix records the principal verification commands and clean-room acceptance results used to document that CellPOS was built as an offline product and that its packaged lifecycle behaves as documented. Its purpose is to preserve the executable and acceptance-level proof surface that supports the CellPOS private-pilot case study.

G.1 Purpose

The CellPOS case-study claim depends on more than repository documents and a packaged artifact. It also depends on executable proof that the product can be built, installed, started, exercised, and checked for fail-closed lifecycle behavior. This appendix therefore records both the verification-command surface and the acceptance results that connect the private-pilot baseline to repeatable repository evidence.

G.2 Core Productization Verification

The main productization verification commands are:

```
npm run test:prod-007-packaging-license-updates
npm run ci:productization-prod-007
npm run test:prod-008-clean-machine-acceptance
npm run ci:productization-prod-008
```

These commands form the core executable proof surface for the final packaging, licensing, update, and clean-room acceptance stages.

G.3 Acceptance Walkthrough Commands

The proof materials also record a packaged clean-room walkthrough used during isolated acceptance testing. In the repository evidence, this walkthrough begins by building or refreshing the packaged bundle and then installing it into an isolated target path outside the repository tree.

```
npm run build:prod-007-package

NODE_OPTIONS=--disable-warning=ExperimentalWarning node ./node_modules/tsx/dist/cli.mjs scripts/install-prod-007-package.ts \
  --bundle-file <released-cellpos-artifact> \
  --install-root <isolated-install-root>

<isolated-install-root>/bin/start-cellpos.sh
<isolated-install-root>/bin/import-license.sh <license-file>
<isolated-install-root>/bin/apply-update.sh <update-package>
```

These commands matter because they show that packaged acceptance was not expressed only as a CI-style test name. It was also documented as an explicit build → install → start → activate → update lifecycle from an isolated path outside the repository tree.

G.4 What PROD-007 Verification Proved

The PROD-007 verification commands were used to demonstrate that:

- a real versioned offline bundle is created;
- the required package layout exists;
- packaged installation and startup work;
- valid local license artifacts are accepted;
- missing or invalid license artifacts are rejected fail-closed;
- valid signed update packages are accepted;
- invalid signed update packages are rejected fail-closed;
- /admin shows version, runtime, license, and update state; and
- packaged /pos, /kds, /order, and /admin still render correctly.

This stage is important because it ties the product claim to a real packaged artifact with functioning local lifecycle operations rather than to a development-tree-only workflow.

G.5 What PROD-008 Verification Proved

The PROD-008 verification commands were used to demonstrate that:

- the product can be installed into an isolated target path outside the repository;
- packaged startup works from the installed layout;
- the required browser surfaces are reachable;
- the embedded AGIF child process starts from the packaged layout;
- manager workflows remain reachable from the packaged runtime;
- license and update behavior work from the packaged layout;
- restart persistence of activated state is maintained;
- the shipped package excludes private signing keys; and
- the final readiness classification is Ready for private pilot.

This stage is especially important because it moves the proof boundary beyond "the package exists" to "the package works from a clean installed location under its packaged lifecycle."

G.6 Regression and Historical Verification

Additional verification commands recorded in the repository evidence include:

```
npm run test:productization-prod-006
npm run test:ui-agif-final-closure-consistency
npm run test:prod-003-real-runner
npm run ci:productization-prod-001
```

These commands provide supporting evidence that:

- manager workflow behavior remained intact;
- tracker and closure consistency remained aligned;
- the embedded AGIF child-process runtime remained real and functional; and
- the initial productization bootstrap and tracker setup remained valid.

Their importance is historical as well as technical. They show that the final private-pilot baseline was not assembled in isolation, but instead sits at the end of a recorded productization path with earlier checks and closure logic still intact.

G.7 Recorded Pass Tokens

The repository records the following pass tokens for the final lifecycle and acceptance phases:

- PROD-007_PACKAGING_LICENSE_UPDATES_PASS
- PROD-008_CLEAN_MACHINE_ACCEPTANCE_PASS

These pass tokens matter because they provide explicit machine-readable closure markers rather than leaving the final readiness state to prose interpretation alone.

G.8 Acceptance Environment Used

The acceptance evidence records that CellPOS was installed into isolated paths outside the repository tree.

Two paths appear in the proof materials for two closely related purposes:

- *Manual walkthrough install target:* <isolated-install-root>
- *Automated clean-room acceptance root: <automated-clean-room-root>*

The first path belongs to the documented human walkthrough. The second path belongs to the recorded automated acceptance sweep. Both matter because they show that the product was exercised from installed packaged layouts rather than from the source tree itself.

The recorded packaged artifact tested was:

- recorded release asset: CellPOS-0.1.0-prod007.0.tar.gz

G.9 Environment Limitation

The repository states this limitation clearly:

- the acceptance run used the cleanest available isolated local install environment;
- it was not a second physical machine; and
- it was not a freshly imaged operating system.

This means the evidence is valid for private-pilot readiness, but it does not support a stronger claim of broader commercial deployment or full cross-environment validation.

G.10 Install and Startup Results

The acceptance evidence records successful proof of:

- install-path creation under the isolated packaged layout;
- packaged file placement into the install root;
- startup from bin/start-cellpos.sh;
- local site-data path initialization; and
- packaged runtime execution without depending on source-tree-only runtime paths.

These results matter because they show that the product can start from its packaged installation boundary rather than from a development-only environment.

G.11 Browser Surface Results

The acceptance evidence records that the following packaged browser surfaces were served successfully:

- http://127.0.0.1:4011/pos
- http://127.0.0.1:4011/kds
- http://127.0.0.1:4011/order
- http://127.0.0.1:4011/admin

The recorded visible checks include:

- /pos showed printer status, TSE/compliance state, and AGIF diagnostics or reorder visibility;
- /kds showed queue health and kitchen bottleneck visibility;
- /order showed reorder and preference visibility; and
- /admin showed AGIF Ops together with product version, license state, and update state.

This matters because it shows that the packaged product was exercised through its real browser surfaces rather than through mock or source-only UI paths.

G.12 Embedded AGIF Runtime Results

The acceptance evidence records successful proof that:

- the embedded AGIF child process started from the packaged layout;
- packaged handshake and health checks remained valid;
- packaged AGIF requests ran through the embedded runtime; and
- runtime state was reflected truthfully in /admin.

The evidence also records fail-closed packaged behavior when:

- the embedded runtime was removed; or
- the embedded runtime manifest was invalid.

These checks matter because they show that the AGIF layer was not merely present in the package as a passive payload. It was exercised as a live embedded runtime under the packaged acceptance boundary.

G.13 Manager Workflow Results

The acceptance evidence records successful packaged proof of all four manager workflows:

- fraud and anomaly detection;
- kitchen bottleneck insight;
- plain-language diagnostics; and
- reorder and preference recall.

For these workflows, the recorded proof includes:

- workflow execution from the packaged runtime;
- trace linkage; and
- truthful degraded or reduced-context visibility where required.

This matters because it shows that the packaged AGIF layer was exercised through real product-facing workflows rather than through isolated technical smoke tests alone.

G.14 License and Update Results

The acceptance evidence records successful proof that:

- a valid local license was accepted;
- a missing or invalid license remained fail-closed;
- license validation remained local rather than cloud-dependent;
- /admin reflected the current license state truthfully;
- a valid signed update package was accepted;
- an invalid signed update package was rejected fail-closed;
- signature verification occurred before application;
- payload-hash verification occurred before replacement; and
- /admin reflected the current update state truthfully.

The evidence also records that the installed packaged version advanced from:

- 0.1.0-prod007.0

to:

- 0.1.0-prod007.1

during valid update acceptance.

These results are significant because they show that the packaged lifecycle includes real local activation and update behavior rather than a static one-shot package proof.

G.15 Security Hygiene Results

The acceptance sweep included a required package-content security check.

The evidence records that:

- the shipped bundle contained no private signing keys;
- only the public trust material required for local verification was included in the packaged distributable;
- repository-local non-production private keys remained outside the shipped bundle;
- the shipped bundle included only config/trusted-keys/non-production-license-public.pem and config/trusted-keys/non-production-update-public.pem; and
- non-production trust material was clearly labeled as non-production.

The proof record also identifies the repository-side location of the non-production verification material as:

- private productization asset directory (repository-relative path omitted from publication copy)

A later trust audit further records that packaged verification resolves trust from local trusted public-key paths, does not accept attacker-supplied signing material merely by reusing expected key identifiers, and preserves private-key exclusion checks for shipped bundles.

This means the shipped package was checked not only for startup and rendering behavior, but also for trust-boundary hygiene at the package-content level.

G.16 Final Acceptance Result

The final recorded acceptance outcome was:

- *Pass token:* PROD-008_CLEAN_MACHINE_ACCEPTANCE_PASS
- *Final readiness classification:* Ready for private pilot

The broader tracker material remains consistent with this result. The repository records PROD-001..PROD-008 as complete and preserves the dashboard verdict HISTORICAL FOUNDATION COMPLETE + SELLABLE PRODUCT READY FOR PRIVATE PILOT.

This is the key acceptance conclusion drawn from the clean-room sweep.

G.17 Interpretation

This verification and acceptance record matters because it shows that CellPOS was not treated as complete on the basis of documentation or source-code presence alone. The project recorded executable proof for packaging, licensing, updates, packaged startup, acceptance behavior, and security hygiene using specific repeatable commands and an isolated installed layout.

At the same time, this appendix should still be read within the scope limits stated elsewhere in the paper. Verification commands and packaged acceptance are strong repository-level evidence, but they are not, by themselves, equivalent to broad field validation. Their role is narrower and more precise: they provide the executable and acceptance-layer proof that makes the CellPOS private-pilot baseline concrete, auditable, and evidence-backed.

Appendix H. CellPOS Scope Limits and Deferred Items

This appendix records the limits and deferred items that remained explicitly unresolved at the end of the CellPOS private-pilot baseline. Its purpose is to preserve the boundary between what the CellPOS case study actually proves and what remains future operational or release work.

H.1 Purpose

The CellPOS case-study evidence is strong because it records a real packaged offline product baseline, a clean-room packaged acceptance sweep, and a published GitHub release record with a concrete artifact. That evidence would become weaker, not stronger, if it blurred the line between what was proven and what was not. This appendix therefore records the scope limits that remain active after the private-pilot baseline.

H.2 Main Recorded Limits

The repository records the known limits explicitly. The documented limits are:

- no production-secure signing infrastructure yet;
- no second physical-machine proof yet;
- no fresh-OS proof yet in this repository; and
- the final state is Ready for private pilot, not controlled commercial beta.

The proof record also preserves an important provenance limit:

- later commits exist after the frozen pilot baseline, but they do not invalidate it;
- later commits add cleanup, README presentation, and trust-verification improvements; and
- the public proof point remains the tagged pilot baseline.

These limits matter because they prevent the case study from being overstated.

H.3 Signing-Infrastructure Limit

The repository proves that offline licensing and offline signed manual updates work locally. It also proves that the shipped bundle excludes private signing keys and includes only the public trust material required for local verification.

What the repository does not claim is a production-secure signing program.

The remaining unproven areas include:

- hardened production key custody;
- production signing-workflow governance;
- key rotation and revocation policy;
- broader operational trust-store distribution; and
- compromised-key response processes.

This distinction matters. The paper can validly claim that local trust anchors, local signature checking, and fail-closed packaged updates are real. It should not claim that full production-grade release-signing operations are already complete.

H.4 Install-Proof and Environment Limit

The acceptance evidence proves that CellPOS was installed and run from an isolated packaged layout outside the repository tree. That is strong private-pilot evidence, but it is not the same as broader deployment proof.

What is proven:

- packaged installation works from an isolated path;
- packaged startup works;
- browser surfaces work;
- the embedded AGIF runtime works;
- the packaged license and update lifecycle work; and
- security-hygiene checks pass.

What is not yet proven:

- repeat validation on a second physical machine;
- repeat validation on a freshly imaged operating system;
- wider cross-environment packaging proof; and
- broader hardware-variability proof.

This distinction matters because the clean-room acceptance sweep was strong enough for a private pilot, but it was not framed as broad deployment validation.

H.5 Why the Final Classification Stops at Private Pilot

The repository explicitly records the final classification as:

- Ready for private pilot

It also records a broader dashboard verdict:

- HISTORICAL FOUNDATION COMPLETE + SELLABLE PRODUCT READY FOR PRIVATE PILOT

It does not record:

- Ready for controlled commercial beta
- Ready for broad commercial release

This is not a weakness in the proof record. It is part of its honesty. The packaged install, startup, and usage evidence are strong enough for a private pilot, but broader operational hardening work is still pending.

H.6 Deferred Post-Productization Items

The repository records the following open operational backlog after productization. Table 7 lists these deferred items.

Table 7
Deferred Post-Productization Items

ID	Category	Status	Purpose
PILOT-001	production signing / key management	OPEN	Replace non-production local verification keys with a production-secure signing and key-handling workflow.
PILOT-002	second-machine / fresh-OS validation	OPEN	Re-run packaged install, startup, and usage on a second physical machine or freshly imaged OS and archive the evidence.
PILOT-003	pilot support + install SOP	OPEN	Create the operator and installation support SOP for pilot setup, restart, update import, and issue escalation.
PILOT-004	hardware / device matrix validation	OPEN	Validate target printer, cash drawer, TSE, and display or device combinations against the pilot hardware matrix.
PILOT-005	commercial release hardening	OPEN	Capture the remaining non-feature commercial hardening work needed after private pilot before broader release.

These items are recorded in the handoff and build-proof materials as future work after the productization initiative was completed.

H.7 Meaning of Deferred Items

The repository is explicit about the scope rule for these deferred items. They:

- do not reopen PROD-001..PROD-008;
- are operational and release backlog only;
- are not, by themselves, authorization for new product feature work; and
- do not mean that productization was incomplete.

This matters because it keeps the CellPOS case study disciplined. The case study can be used as real private-pilot evidence without pretending that every post-pilot release concern has already been resolved.

H.8 Interpretation

This appendix should be read as a boundary-preservation appendix. It records the difference between a strong private-pilot case-study proof and a broader commercial-readiness claim. The CellPOS repository proves that bounded local intelligence can be embedded into a real product-shaped workflow and packaged as an offline product baseline. It does not claim that all deployment, signing, hardware, or cross-environment validation work is complete.

That disciplined boundary is part of what makes the CellPOS evidence useful to the broader Tasklet Cells paper. The case study is strong because it is real, packaged, verified, and published, but also because it does not pretend to be more complete than the repository evidence supports.