Schema-as-ABI: Typed Capabilities and Ring-Transport Dispatch in capOS

Pre-evidence draft, last reviewed 2026-04-29 09:08 UTC.

> **Status: pre-evidence draft.** Sections that depend on missing artifacts (measurement numbers, persistence proof-of-concept, network-transparency proof-of-concept, completed service-object identity migration) are marked with explicit `TODO` blocks naming the gap and the entry in `docs/paper/evidence-gaps.md` that closes them. Do not cite claims that depend on those gaps until the gaps are closed.

Abstract

TODO Abstract. Write last, after the Evaluation section has numbers and the Service Object Identity Migration is complete. The current draft has enough content for a structural skeleton and load-bearing sections; the abstract will summarize the closed contributions, not the in-flight ones.

Closes when: all Tier-1 evidence in `evidence-gaps.md` is checked in.

1 Introduction

Operating-system kernels typically expose two parallel, weakly connected mechanisms for describing what userspace is allowed to do: a *syscall surface* defined by C function signatures and a fixed numeric dispatch table (e.g. Linux's `__NR_read = 0`), and an *authorization mechanism* defined by some combination of access bits, rights masks, role strings, or path-based namespaces. Capability operating systems unify the authorization side around object handles, but they typically still keep the typing layer (the wire format, the rights bits, the IPC payload contract) as separate concerns layered on top of those handles. seL4 keeps untyped IPC buffers and per-entry rights masks. Zircon types its kernel objects but the typing lives in C handle conventions and a fixed syscall numbering. Genode provides component-typed RPC, but RPC is implemented over an underlying kernel's IPC primitives that are themselves untyped.

capOS is a research operating system that pushes a single mechanism, the Cap'n Proto schema, into all four roles at once:

1. The schema *is* the kernel ABI. There is no separate C-header syscall convention; method invocation is a serialized capnp message.
2. The schema *is* the access-control mechanism. There is no parallel rights bitmask. To restrict what a caller can do, a service grants a narrower capability (a wrapper object that exposes fewer methods), not the same capability with reduced flags.
3. The schema *is* the IPC wire format. Cross-process invocation uses the same serialized form as kernel invocation.
4. The schema *is* the substrate intended to carry persistence and network-transparency over the same wire.

Roles 1 and 3 are implemented and demonstrated by the artifact released with this report. Role 2 is *partially* implemented: the schema replaces per-handle rights bitmasks (no `READ` / `WRITE` flags gate method invocation; authority is narrowed by wrapping), but caller-selected endpoint identity remains as transitional service-identity machinery in some demos, with an in-progress migration to service-minted object capabilities described in Section 5.5. Role 4 is supported by the design but not yet exercised by an end-to-end proof-of-concept. The partial and open positions are recorded in the Limitations section and in the live `evidence-gaps.md` companion to this draft.

This report makes the following contributions, structured so that each contribution maps to one or more concrete artifacts in the released codebase:

- **C1 (partial).** A capability-OS design where typed schema methods replace parallel rights bitmasks, validated by an implementation in which kernel dispatch carries no per-entry rights and authority narrowing is performed exclusively by wrapping. Section 3. The rights-bitmask half of this contribution is closed; the service-identity half remains partial while the legacy endpoint identity to service-object migration is in progress. The selected Service Object Identity Migration closes the C1 paper claim once the routing/lifecycle proof, subject/proof root-open proof, and chat/adventure/stdio migration land; the legacy naming-cleanup pass does not gate the claim. See Section 5.5 and `evidence-gaps.md` claim C1.
- **C2 (closed implementation, open evaluation).** A two-syscall kernel boundary (`exit`, `cap_enter`) in which all capability operations are shared-memory ring submissions. Implementation: complete. Empirical characterization (latency, throughput, IPC handoff cost, schema-dispatch overhead): TODO §8, see `evidence-gaps.md` claim C2. Section 4 and Section 8.

- **C3 (open hypothesis).** The wire format enables persistence: a capability whose state is serialized via its capnp schema and restored across kernel reboot. No proof-of-concept yet exists; Section 9.3 and Section 10 describe the missing artifact and the paper-scoped scope it will cover. See `evidence-gaps.md` claim C3.
- **C4 (open hypothesis).** The wire format enables network transparency: a capability invocation crossing TCP between two capOS instances (or between a capOS guest and a host capnp stub) using the same schema as the local case. No proof-of-concept yet exists; Section 9.4 and Section 10 describe the missing artifact. See `evidence-gaps.md` claim C4.
- **C5 (partial).** A pragmatic verification stack for a `no_std` kernel of this size: bounded Kani over the pure-logic library, a Loom ring model, Miri, fuzz targets at parser boundaries, and a workplan/review discipline that makes review findings durable. The implementation is in place; Tier-2 strengtheners (notably a ring-protocol Kani proof) remain open. Section 7. See `evidence-gaps.md` claim C5.

The closed implementation work in Section 5 (capability lifecycle: copy, move, transport-level release, object-epoch revocation, and exactly-once accounting rollback) is system mechanism rather than an independent paper claim; it underpins C1 and C2 and is described in Section 5 as such.

> **TODO §1 contribution list.** Reorder and tighten once C2 numbers exist and the Service Object Identity Migration is complete. Until then this list deliberately separates closed from open contributions rather than overstating coverage.
>
> **Closes when:** `evidence-gaps.md` claims C1, C2, C3, C4 are all checked in.

2 Background and Related Work

The design space for capability operating systems has been explored along five mostly-independent axes: capability table structure, IPC mechanism, memory-management capability shape, persistence model, and naming model. Most surveyed systems pick a strong position on one axis and follow convention on the others.

2.1 seL4

seL4 organizes capabilities in CNodes — power-of-two arrays joined into a tree by guard bits — and reaches them through capability paths (O(depth) lookup). Authority is qualified by a rights mask per slot (read/write/grant/grant-reply) and by a u64 *badge* set when the capability is minted. IPC is synchronous through Endpoint kernel objects; the typical fast path is a direct context switch from caller to callee with the message payload carried in message registers. Notification objects provide a lightweight bitmask signal/wait primitive separate from full IPC. seL4 is formally verified at the binary level for selected configurations. Recent work introduces Mixed-Criticality Scheduling (MCS) where scheduling contexts can be donated through IPC to prevent priority inversion.

What capOS borrows: the direct-switch IPC fast path on synchronous cross-process calls (Section 4); the *idea* of a badge for distinguishing callers (kept transitionally, see Section 5.5 and the service-object-capabilities migration); the principle that the kernel should be small enough to verify at all.

What capOS does not borrow: the per-entry rights mask, for the reason stated in Section 3.1 — it duplicates the typing layer the schema already provides.

2.2 Zircon (Fuchsia)

Zircon is a capability microkernel where every kernel object (channel, VMO, port, thread, process, …) is referenced by a *handle*. Handles carry rights flags chosen from a fixed enumeration. The typing of what methods exist on a handle is enforced by the C syscall surface, not by a schema; FIDL (the Fuchsia interface language) describes user-level RPC over channels, but the kernel itself does not parse FIDL. Channels are asynchronous typed message queues that can carry handles. VMOs (Virtual Memory Objects) are the bulk-data substrate; VMARs (Virtual Memory Address Regions) are the address-space-side capability.

What capOS borrows: the generation-tagged handle (CapId in capOS) for stale-reference detection; the move-by-default semantics for handle transfer through IPC; the VMO concept, present in capOS as `MemoryObject`.

What capOS replaces: the parallel rights mask. The role rights serve in Zircon — preventing a holder of a `READ`-only handle from invoking a `WRITE` syscall — is served in capOS by the schema itself, because the schema lists which methods exist on the handle. There is no `WRITE` syscall to gate.

2.3 EROS / CapROS / Coyotos

EROS introduced two ideas this report relies on. First, *transparent persistence*: the kernel periodically checkpoints the entire global state (including all process address spaces and capability tables) to disk, so that a reboot is indistinguishable from any other context switch. Second, *confinement* as a verifiable property of capability systems. The cost of transparent persistence is enormous kernel complexity and difficult debuggability.

What capOS borrows: object-epoch revocation (an EROS idea — bumping a per-object epoch invalidates all outstanding references in O(1)); the *explicit* persistence stance derived from EROS's costs (capOS plans explicit `Store`/`Namespace` capabilities rather than transparent checkpointing).

What capOS does not borrow: transparent global checkpointing.

2.4 Genode

Genode organizes systems as trees of components, each holding capabilities to *sessions* on services provided by their parent or by services routed to them by parent policy. Genode runs on top of various underlying kernels (seL4, NOVA, Linux); IPC and capability primitives come from those kernels. Genode contributes the architectural pattern that resource ownership is made explicit through *session quotas*: a client donates RAM, CPU budget, or other resources at session creation, and the server allocates from that donation rather than from its own quota.

What capOS borrows: the architectural style (services are normal components, identity comes from object capabilities they hold and mint, parent-mediated routing); session-style resource accounting at capability-creation boundaries (Section 5.3); the conviction that VFS behavior belongs in libraries and services rather than the kernel.

What capOS does not adopt: a separate parent-mediated routing layer. capOS's manifest acts as the boot-time routing input, and trusted brokers/sessions handle dynamic routing.

2.5 Plan 9 / Inferno

Plan 9 unifies system interfaces under "everything is a file": resources are exposed as filesystems served via the 9P protocol, and per-process namespaces compose those filesystems with `bind`/`mount` operations. This yields a strong uniformity property and a natural network transparency (serving a 9P filesystem over the network is the same as serving it locally).

What capOS borrows: the per-process namespace as authority shape (capOS's per-process capability table is a stronger version of the same idea). The per-process Namespace cap is influenced by Plan 9 union mounts.

What capOS does not adopt: the universal file metaphor. File I/O is one weak type; capOS prefers schema-typed interfaces where each operation carries its own contract.

2.6 Cap'n Proto and capnp-rpc

Cap'n Proto is an interface description language and serialization format designed for zero-copy access and capability-based RPC. It defines an RPC protocol (capnp-rpc) in which each side maintains four tables (questions, answers, imports, exports), capabilities are referenced by connection-local IDs, transport-level reference counting flows over a `Release` message, and method calls can be promise-pipelined: a call on the *result* of an outstanding call may be sent before the first call returns, and the server resolves both in one round trip.

capOS's design hypothesis is that the kernel-userspace boundary can be modeled as one party of a capnp-rpc connection. Each process is a vat; the per-process capability ring is its RPC connection to the kernel. This unifies the kernel ABI, in-process invocation, IPC, and (eventually) remote capability calls under one transport pattern.

2.7 io_uring

The completion-ring transport in capOS is shaped after Linux's `io_uring`: a shared submission queue and completion queue between userspace and kernel, with a single syscall (`cap_enter` in capOS, `io_uring_enter` in Linux) that drives progress. capOS extends the pattern by routing *capability invocation*, not just block I/O, through the ring; the two-syscall surface (`exit` and `cap_enter`) is a deliberate consequence of that choice.

2.8 Summary of Position

The contribution of capOS, against this background, is not a new capability-OS primitive. It is the choice of a single typed wire layer spanning kernel ABI, access control, IPC, and (planned) persistence and network transparency. Each surveyed system uses a typed schema for one of these roles; capOS attempts to use one schema for all of them, and the report evaluates the consequences.

3 Capability Model

A capability in capOS is a process-local handle to a kernel object that carries:

1. an **interface** — the Cap'n Proto schema describing which methods are callable on this handle;
2. a **referent** — the kernel object the handle is permission to act on;
3. a **wire format** — serialized capnp messages for every invocation, the same in-process and across processes.

There is no ambient authority. There is no global namespace, no "open by path" syscall, no implicit resource access, and no rights bitmask on capability slots.

3.1 The Interface IS the Permission

In Zircon and seL4, a kernel-object handle is *opaque*. The kernel decides what methods can be invoked on it by consulting a per-handle rights bitmask: a `VMO` handle with the `READ` right can be `zx_vmo_read`-ed; without it, the syscall returns `ZX_ERR_ACCESS_DENIED`. Rights are a parallel typing mechanism imposed because the handle itself is not typed.

In capOS, a capability slot exposes one Cap'n Proto interface, identified by a 64-bit interface ID. Method dispatch consults the schema, not a flag mask: a `Console` capability has `write` (method 0) and `writeLine` (method 1); there is no other method to gate.

To attenuate authority, the granting code creates a *different* `CapObject` implementation that wraps the original and exposes only a subset (or a transformed subset) of methods. From the kernel's perspective, this is just a different object in a slot. Examples:

- `Store` (read/write) wrapped to a `ReadOnlyStore` that forwards `read` and rejects `write`;
- `Fetch` (full HTTP) wrapped to a `HttpEndpoint` scoped to one origin;
- `Namespace` (full) wrapped to a `Namespace` scoped to a prefix.

Each wrapper is a normal capability: same dispatch path, same lifecycle, same wire format, no kernel awareness of the attenuation.

This decision is reversible: meta-rights for the *capability reference itself* (`TRANSFER`, `DUPLICATE`) may eventually be added, because they are about reference handling, not about the methods of the referent. But there is no rights mask gating method invocation, by construction.

3.2 Capability Table Structure

Each process owns a flat capability table mapping a `CapId` to an `Arc<dyn CapObject>`. `CapId` is encoded as `[generation:8 | index:24]`: the upper byte increments each time a slot is freed, so a stale handle held across a slot reuse is rejected with `CapError::StaleGeneration` rather than silently aliasing a new object.

A flat table with O(1) lookup is sufficient for capOS's use cases. seL4's CNode tree is a richer structure that pays cost in lookup depth for delegation patterns capOS does not currently need; if those patterns emerge, the table can be replaced without disturbing the dispatch path (the `CapTable` interface lives in the host-testable `capos-lib/src/cap_table.rs` and is exercised under bounded Kani).

3.3 Bootstrap: the CapSet Page

A child process is born with no implicit authority and no `argv`-style inheritance of kernel state. Instead, the kernel maps a single read-only 4 KiB page (the *CapSet page*) at a fixed virtual address. The page starts with `CapSetHeader { magic, version, count }` and is followed by `CapSetEntry { cap_id, name_len, interface_id, name: [u8; 32] }` records in manifest declaration order. The runtime locates a capability by manifest name, not by numeric index, so the manifest can be reordered without breaking clients. The page is mapped without write permission so the process cannot mutate its own bootstrap authority map.

3.4 The Schema as Contract

Capability interfaces live in `schema/capos.capnp` and are versioned through the same review and generated-code-check pipeline as the kernel source. The current schema includes interfaces for `Console`, `TerminalSession`, `FrameAllocator`, `MemoryObject`, `VirtualMemory`, `Endpoint`, `ProcessSpawner`, `ProcessHandle`, `BootPackage`, `Timer`, `ThreadControl`, and a management-only `CapabilityManager`, alongside the identity/policy interfaces (`UserSession`, `CredentialStore`, `SessionManager`, `AuthorityBroker`, `RestrictedShellLauncher`, `AuditLog`, `EntropySource`) and the SSH-related contracts described in Section 6.

The schema is the *contract*. Generated Rust bindings are checked into the repository and validated by `make generated-code-check`, which compares the checked-in baseline against output of the pinned Cap'n Proto compiler. This makes schema drift review-visible.

A subtle but load-bearing consequence: a capability slot exposes exactly one public interface. If the same backing state must be reachable through two interfaces, the granting code mints two separate capabilities, each wrapping the same state with one of the interfaces. Encoding "this slot accepts multiple interface IDs" was deliberately rejected because it makes hidden authority easy to miss during review. (See `REVIEW.md` and `docs/security/trust-boundaries.md` for the security-review consequences.)

4 Ring Transport

All capability invocations cross the kernel boundary through a shared-memory submission queue / completion queue (SQ/CQ) ring, one per process, mapped into the process address space at a fixed virtual address. The kernel exposes only two syscalls: `exit` (terminate the process) and `cap_enter` (process pending submissions and wait for completions). New operations are SQE opcodes, not new syscalls.

4.1 Submission and Completion

Each non-idle process is assigned one 4 KiB ring page containing a volatile header, a 16-entry SQ, and a 32-entry CQ. The SQE is a fixed 64-byte ABI record. The userspace path to invoke a capability is:

1. Build a Cap'n Proto params message and serialize it into a user-owned buffer.
2. Write a `CapSqe` into the SQ describing opcode (`CAP_OP_CALL`), capability handle, method ID, params buffer pointer/length, result buffer pointer/length, and (for transfer-bearing calls) transfer descriptor count.
3. Advance the SQ tail (a normal user memory write).
4. Issue `cap_enter(min_complete, timeout_ns)`.

Steps 1–3 are syscall-free. The kernel reads the SQ during `cap_enter`, validates SQE fields, validates and locks the user buffers behind the process `AddressSpace` mutex, dispatches to the capability object, copies the serialized result into the caller's result buffer, and writes a `CapCqe` into the CQ. `cap_enter` returns the available completion count or blocks (with the supplied timeout) until enough completions exist.

Successful completions return non-negative byte counts. Transport failures return negative `CAP_ERR_*` codes. Application-level errors serialize as `CapException` payloads delivered with `CAP_ERR_APPLICATION_EXCEPTION`. Distinguishing transport failures from application failures is part of the ABI: malformed ring metadata, bad user buffers, lookup failures, and rollback errors stay in the transport namespace; any failure originating inside a successfully-dispatched capability method is application-typed.

4.2 Opcode Surface

The implemented opcode set is intentionally minimal:

- `CAP_OP_CALL` — invoke a method on a process-local capability.
- `CAP_OP_RECV` — accept a queued call on an `Endpoint` server side.
- `CAP_OP_RETURN` — complete a previously received call (with ordinary results, with application exception, or with a transferred result capability).
- `CAP_OP_RELEASE` — release a process-local capability slot (transport-level reference drop; not an application method).
- `CAP_OP_NOP` — measurement-only, exercises the ring path with no capability work.
- `CAP_OP_PARK` / `CAP_OP_UNPARK` — compact capability-authorized park operations on a `ParkSpace` for the in-process threading runtime.
- `CAP_OP_FINISH` — ABI-reserved for the future system capnp transport layer; the current kernel rejects it as `CAP_ERR_UNSUPPORTED_OPCODE`, so it cannot be silently confused with a malformed opcode.

Ordinary `CAP_OP_CALL` SQEs are processed only at `cap_enter`, not from the timer interrupt, even though the kernel polls the ring on every preemption tick. The reason is execution-context safety: SQE submission is just a shared-memory write, but executing a capability method may allocate, lock, mutate page tables, parse capnp, spawn processes, or perform IPC side effects, none of which is safe from interrupt context. Timer polling is restricted to non-CALL ring work and explicitly interrupt-safe CALL targets that opt in.

4.3 Opcode Admission Policy

The opcode set above is small by design. capOS treats the ring opcode table as kernel ABI, parallel to but distinct from the syscall trap surface, and applies an explicit decision graph (`docs/architecture/capability-ring.md`, "Choosing a Capability Method, Ring Opcode, or Syscall") to any new kernel-visible operation. The default is a typed capability method dispatched through `CAP_OP_CALL`. A new compact opcode is admitted only when generic Cap'n Proto framing is materially wrong for the operation's hot path *and* the operation needs ring- or scheduler-specific behavior such as thread ownership, reserved completion credit, CQ ordering or backpressure, or interaction with the process ring head. A new syscall has a higher bar still: it is admitted only when the operation is about entering or leaving the kernel execution context itself and cannot be authorized by a capability the process already holds. The current `exit` and `cap_enter` syscalls meet that bar; ordinary resource operations are not eligible to become syscalls just because they are common.

The compact park opcodes (`CAP_OP_PARK` and `CAP_OP_UNPARK`) qualify under this policy because blocking wait mutates scheduler state, must be thread-owned on the process ring, reserves completion credit for later wake or timeout delivery, and needs compact capability-authorized hot-path framing. They are not a precedent for moving ordinary object methods into the opcode table for convenience. Diagnostic opcodes such as `CAP_OP_NOP` are kernel ABI but exempt from the authority-bearing opcode review gate; they must stay side-effect-free and review-visible.

The relevant property for the schema-as-ABI thesis is structural: the typed capability method is the default; opcode and syscall surfaces are deliberately narrow exceptions with reviewed gating criteria; and the kernel ABI does not silently grow. Each surface (capability method, ring opcode, syscall) is reviewed through the same change- control discipline as the rest of the kernel, with the bar tightening toward the trap surface.

4.4 IPC and Direct-Switch Handoff

Cross-process invocation goes through `Endpoint` capabilities: a server holds an endpoint with a queue of pending calls and posts `CAP_OP_RECV` SQEs to accept them; a client invokes through a *client facet* of the same endpoint with `CAP_OP_CALL`. When the client posts a CALL and the server is already blocked in a RECV waiter, the kernel performs a *direct-switch handoff*: instead of returning to the round-robin scheduler, the next runnable thread is set to the server, so the CALL runs before any unrelated work. This eliminates one full scheduler round-trip from the synchronous IPC fast path.

> **TODO §4.3 measurement.** The latency saving from direct-switch vs. the ordinary scheduler path is one of the headline numbers the Evaluation section needs. Currently asserted by construction, not measured.
>
> **Closes when:** `evidence-gaps.md` claim C2 measurement harness lands the IPC handoff figure.

4.5 The Ring Is Not the Final Multi-Thread ABI

The current ring is process-wide. With sibling threads in one process running on different CPUs, a shared CQ would force userspace to serialize completion consumption (or the kernel to invent specific-wait state on top of a circular buffer). The accepted direction is per-thread ring ownership; until then, a runtime *reactor* in `capos-rt` consumes the process CQ and demultiplexes completions by `user_data`, waking waiting threads through a capability-authorized `ParkSpace`. The reactor is a compatibility bridge, not the target ABI.

> **TODO §4.4.** Describe per-thread ring v2 design once it is in code, citing `docs/proposals/ring-v2-smp-proposal.md`. Currently a directional commitment, not a contribution.
>
> **Closes when:** Stage 7 SMP work lands per-thread completion routing.

4.6 Bounded Loom Model

Producer-consumer invariants of the SQ/CQ ring are checked under a bounded Loom model in `capos-config/tests/ring_loom.rs`, exercised by `cargo test-ring-loom`. The model covers SQ/CQ capacity, FIFO order, CQ overflow/drop behavior, and corrupted-SQ recovery. It does not yet cover the full state space of the kernel-side dispatcher, which remains out of scope for Loom. Section 7 describes how Kani complements Loom.

> **TODO §4.5.** Promote ring protocol from Loom-only to Kani when the Tier-2 proof obligation is written. Cite the eventual Kani harness.
>
> **Closes when:** Tier-2 ring Kani proof in `evidence-gaps.md` C5.

5 Capability Lifecycle

Capabilities have a non-trivial lifecycle: they are created, copied or moved into other capability tables, possibly attenuated by wrapping, released through the transport, and revoked when their referent goes away. Each step interacts with resource accounting and with the kernel's failure-rollback discipline.

5.1 Copy, Move, and Transfer Descriptors

Capability transfer accompanies CALL and RETURN SQEs through *transfer descriptors* packed after the params/result payload. Each descriptor specifies a sender-side `cap_id` and a `transfer_mode` of either `COPY` or `MOVE`. Copy duplicates the slot in the receiver's table while preserving the sender's slot; move atomically transfers the slot. The kernel validates descriptor alignment and bounds, looks up source caps, and either commits the entire transfer set atomically or rolls all of it back; partially-applied transfers are not a possible kernel state.

Result-cap insertion is the dual: a `RETURN` payload may include transferred result capabilities. The completion advertises this through `CAP_CQE_TRANSFER_RESULT_CAPS` and a `cap_count`; the transferred capability IDs and interface IDs are appended to the result buffer as `CapTransferResult` records. The runtime client validates the kernel-supplied interface ID against the expected client interface before producing an owned typed handle.

5.2 Transport-Level Release

Cap'n Proto applications do not encode capability lifetime as an application method on every interface; the *transport* owns reference bookkeeping. capOS follows the same discipline. The runtime client owns typed local handles; when the last local handle drops (Rust `Drop`, later language runtimes' GC/finalizer), the runtime queues a `CAP_OP_RELEASE` for that capability ID, and an explicit runtime flush or the next ring-client boundary submits it through the ring. `CapabilityManager` exposes only management operations (`list`, child-scoped `revoke`); ordinary local release is *not* a method on `CapabilityManager`, because a method call would need to dispatch through the same table the release is mutating.

`CAP_OP_FINISH` is reserved in the same opcode namespace for a future application-level "end of work" signal that the transport must deliver reliably; because it is reserved, a malformed-opcode submission cannot be accidentally treated as a finish.

5.3 Resource Accounting and Rollback

Every authority transfer in capOS goes through one of a small number of accounting paths. Frame grants from `FrameAllocator` and `MemoryObject`-backed pages charge a per-process `ResourceLedger::frame_grant_pages` counter; `VirtualMemory` reservations charge the same ledger. Capability-table slot consumption is bounded by a per-process slot quota. `ProcessSpawner` charges the parent for child-table slots and frame grants taken on the child's behalf.

Failure-rollback is the load-bearing invariant. A failed `spawn`, copy, move, or release must leave the relevant ledger unchanged, the relevant table slots unchanged, and the relevant frame ownership unchanged. This invariant is asserted by host tests, exercised under Kani in `capos-lib`, and proven against hostile spawn inputs in `make run-spawn`. Recent kernel hardening (commits `e7857b9`, `41969af`) made the `VirtualMemory` reservation release transactional and hardened the rollback paths

after a review finding showed a non-transactional release could double-decrement on certain failure sequences; both are recorded in `REVIEW_FINDINGS.md`.

5.4 Revocation: Object Epochs

Revocation propagates through *object epochs*. Each capability backing state carries an epoch counter; `CapabilityManager.revoke` increments the epoch, after which any ring use of a stale handle for that object either fails closed (in the transport namespace) or, where a result buffer exists, completes with a typed `Disconnected` application exception. The `make run-revocable-read` smoke proves this for a revocable read capability: a child holds a read handle, the parent revokes the underlying object, the next read fails closed with `Disconnected`, and process-exit cleanup releases ledger charges correctly.

5.5 Service Object Capabilities (in progress)

The current implementation also keeps a transitional `u64` *badge* field on endpoint client facets, derived from seL4-style mint-time labeling. Earlier shared-service demos (chat, adventure, stdio) used the badge as the per-client identity that the server multiplexed on. This is the single largest open authority shape remaining in the codebase, and an explicit migration to *service object capabilities* is in progress: each per-client identity becomes its own service-minted object capability (`ChatParticipant`, `AdventurePlayer`, …), the endpoint becomes a transport queue not a service-identity field, and ordinary delegated clients can no longer relabel their badge to impersonate another identity.

The selected migration is tracked in `docs/backlog/service-object-identity-migration.md`; the Stage 6 capability semantics backlog keeps historical containment/substrate detail:

- **Containment and transitional substrate (landed).** Ordinary delegated client grants must preserve source identity, normal shell help no longer exposes `badge N` syntax, hostile spawn coverage proves nonzero relabels and legacy badge-zero encodings fail closed, and current endpoint-backed service facets preserve held receiver metadata across ordinary copy and move.
- **Big Chunk 1 (open).** Add explicit service-object routing/lifecycle semantics and prove them with a synthetic QEMU service before migrating chat, adventure, or stdio.
- **Big Chunk 2 (open).** Validate local subject/proof authority before object mint, aligning with external-subject admission into local or pseudonymous principals.
- **Big Chunk 3 (open).** Convert chat, adventure, and `StdIO` away from caller-selected endpoint identity to service-minted object capabilities.
- **Big Chunk 4 (open).** Retire compatibility state: rename internal fields where the behavior is now receiver-selector-only and remove legacy syntax from manifests and smoke harnesses.

> **TODO §5.5.** When the Service Object Identity Migration closes, this section should describe the *post-migration* model in present tense and move the migration narrative to a historical note. The current text intentionally describes the in-progress state, because the C1 contribution claim is partial until shared-service migration closes. Legacy internal field renaming is a follow-up cleanup pass and does not block the C1 claim.
>
> **Closes when:** `evidence-gaps.md` claim C1 migration items are checked in.

6 System Composition

The capability primitives in Section 3 through Section 5 are the substrate. To demonstrate they support a working system, capOS composes them into a small but realistic boot-to-shell pipeline driven entirely by an explicit boot manifest.

6.1 Manifest-Driven Boot

The boot manifest is a Cap'n Proto `SystemManifest` produced by the host-side `mkmanifest` tool from a CUE source (e.g., `system.cue`), embedded as a single Limine boot module. It declares the kernel parameters, the binaries available in the image, the `init` process, and (in non-init-led manifests) the service graph that init spawns. The kernel validates only the boot-boundary parts it owns (`schemaVersion`, `binaries`, `initConfig.init`, `kernelParams`); the service graph is validated by init through `capos-config` before spawning children. Manifests serve as the boot-time routing input, analogous to Genode's parent-mediated routing but evaluated once at boot instead of dynamically.

6.2 Identity and Authorization at the Edge

Default boot launches `capos-shell` directly as init, with the following caps from the kernel CapSet: a session-scoped `TerminalSession`, a `CredentialStore` for password verifiers, a `SessionManager` for minting `UserSession` capabilities, an `AuditLog`, and an `AuthorityBroker` that mints scoped *shell bundles*.

The shell mints an *anonymous* `UserSession` on boot, asks the broker for an `anonymous` shell bundle (which has an empty launcher allowlist), and enters its REPL. The user can then `login` (verify against `CredentialStore` to upgrade to an *operator* session) or, if no credentials are configured, `setup` (provision a first verifier and chain into login). Failed authentications use bounded backoff and redact secrets from audit records; the bootstrap `EntropySource` authority is held only by `CredentialStore` and `SessionManager`, not ambient. The boundary is exercised by `make run-login`, `make run-login-setup`, and `make run-local-users`.

This is a small-scale but complete instance of the design rule that *identity is policy metadata, not authority*. The kernel never sees "users", "roles", "UIDs", or "principals". It sees only typed capability handles. Identity flows through the broker, which uses it as input when minting (and only when minting); after the bundle is issued, the bundle is the authority.

6.3 Demo Services

Three demo services are wired through this composition:

- **First Chat.** A resident chat service spawned by the shell, with shell-spawned clients each receiving a `ChatParticipant` (post-migration) or a badged `Chat` client facet (current state). Round-trip verified by `make run-chat`.
- **Local Adventure.** A richer multi-process demo with separate chat-only NPC processes (`Centurion Varro`, `Magister Livia`, etc.) each receiving only `console` and a manifest-badged `chat` grant, while `adventure-server` holds the adventure state authority. Drives `make run-adventure`.
- **MemoryObject sharing / Revocable Read.** Proofs of cross-process shared-memory and of object-epoch revocation, respectively (`make run-memoryobject-shared`, `make run-revocable-read`).

These demos are deliberately small. Section 9 treats them as illustrative rather than evaluative.

6.4 Telnet and SSH Shell Gateways

The paused *SSH Shell Gateway* milestone is an SSH-backed `TerminalSession` reaching the same `capos-shell` flow used on the local console, through a host-local QEMU port forward, with public-key authentication, denied unsupported SSH features (no `exec`, no SFTP, no port/agent/X11 forwarding, no env import, no multiple session channels), and with the spawned shell receiving no raw network, key, or SSH-transport authority. The Telnet Shell Demo predecessor is complete and merged (`2834bfc`, `make qemu-telnet-harness`); it is bound to host loopback and is explicitly demo-only, not a production remote-access boundary.

The SSH gateway is built from a stack of narrow capabilities: `SshGateway`, sign-only `SshHostKey` (a development host-key fixture for now, with production host keys blocked on persistent storage and key-management), `AuthorizedKeyStore` (manifest-seeded principals mapped to allowed shell profiles), `TcpListenAuthority` (scoped to the SSH development port), `SshTerminalFactory` (consumes a connected TCP socket into a `TerminalSession`), and `RestrictedShellLauncher` (launches only `capos-shell` with the supplied terminal/session caps; no binary selector, no spawn authority surface). Five of the six implementation slices in `docs/backlog/runtime-network-shell.md` are closed and proven by focused QEMU smokes. The end-to-end OpenSSH-protocol slice is open.

> **TODO §6.4.** When `make run-ssh-shell` proves an OpenSSH host client opens a session channel, runs one shell command, and disconnects cleanly, this section should describe SSH as the external-facing artifact end-to-end. Until then it is a partial contribution.
>
> **Closes when:** SSH Shell Gateway milestone is resumed and achieved.

6.5 What System Composition Demonstrates

What this composition is intended to support, *and what it does not yet support*, is summarized in Section 9. The relevant positive claim is: a Cap'n Proto schema layered over a two-syscall ring transport is sufficient to implement a non-trivial layered authorization system (anonymous to operator to broker to bundle to shell), with scoped network ingress, with clean process-exit cleanup of capability state, and with auditability that does not leak secrets through the wire format.

7 Verification and Engineering Process

This section describes the engineering process the codebase actually uses, not an aspirational target. The relevant question for a research OS at this size is not "is the kernel formally verified" (it is not, and that is not a goal of this report) but "what proof and review machinery makes a kernel of this size tractable to develop and audit."

7.1 Verification Layering

Tool	Scope	Mandatory gate
Bounded Kani	Pure logic in `capos-lib`: frame bitmap allocation, cap-table generation invariants, frame-grant and cap-slot fail-closed accounting, transfer-origin fail-closed	`make kani-lib`
Loom	SQ/CQ producer-consumer model: capacity, FIFO, overflow/drop, corrupted-SQ recovery	`cargo test-ring-loom`
Miri	Pure-logic crate undefined-behavior detection	`cargo miri-lib`
Property tests / fuzz	Manifest capnp parser, exported JSON conversion, ELF parser	`fuzz/` corpus
Host tests	Config, manifest, capset layout, shared-ring helpers, ELF, frame bitmap, frame ledger, cap table, mkmanifest behavior	`cargo test-config`, `cargo test-lib`, `cargo test-mkmanifest`
Generated-code drift	Cap'n Proto compiler version pinning, schema binding equality, no_std patch anchors, content baseline drift	`make generated-code-check`
Dependency policy	cargo-deny / cargo-audit across workspace + standalone lockfiles	`make dependency-policy-check`
QEMU smokes	End-to-end behavioral proofs over `make run-*` targets, one per behavior boundary	the relevant `make run-*`
Trusted-input pinning	Limine binary SHA-256 pinning, Cap'n Proto compiler pinning, generated-code baseline	`Makefile` + `docs/trusted-build-inputs.md`

Table 1: The capOS verification stack at the time of writing.

The verification stack is deliberately *layered*: each tool covers properties the others cannot. Kani exhaustively explores bounded state spaces over pure logic. Loom searches concurrent interleavings of the ring producer/consumer model. Miri detects undefined behavior. Property tests and fuzzers explore parser boundaries with arbitrary bytes. QEMU smokes prove end-to-end behavior in a real kernel boot. None of these alone is sufficient; together they cover the trust boundaries listed in `docs/security/trust-boundaries.md`.

7.2 What Verification Does *Not* Cover

Honest framing matters here. This stack does not formally verify the kernel. In particular:

- The Loom model covers the SQ/CQ protocol, not the kernel's full scheduler-level state.
- Kani is bounded over pure logic. The kernel proper (interrupt handlers, page table edits, syscall MSR setup, AP startup) is not under proof.
- QEMU smokes are *behavioral* proofs: a smoke that boots, drives a scenario, and prints expected output proves the scenario, not the full state machine.
- Several panic surfaces are tracked by classification in `docs/panic-surface-inventory.md` rather than eliminated; those are open hardening work.

These limits are why Section 9 is explicit about what cannot yet be claimed.

7.3 Workplan and Review Discipline

The codebase is governed by three live files: `docs/roadmap.md` (the long-term direction), `WORKPLAN.md` (the actionable next-steps queue), and `REVIEW_FINDINGS.md` (the durable open-issue tracker for review findings that have not yet been resolved). Together with `REVIEW.md` (the review checklist) they form the operational loop.

Two rules in this loop are load-bearing for the rest of the report:

1. **Every file change happens in a dedicated branch in a separate worktree, including documentation-only updates.** The main worktree is read-only for source and documentation. This rule is the reason a single-author repository can sustain code-review discipline at all — there is always a branch to comment on and to gate.
2. **Review findings are durable, not transient.** Issues raised in a review go to `REVIEW_FINDINGS.md` if not resolved in-task. This prevents the same issue from being raised twice and makes it visible when a feature task starts that depends on a stale finding.

Stage and milestone closeouts include mandatory documentation updates (`docs/roadmap.md` for stage status, `WORKPLAN.md` for next-step ordering, the relevant proposal docs for accuracy, and timestamps in status updates with minute precision and timezone). This is not optional — a stage-closing commit that does not update the docs is considered incomplete.

7.4 Collaboration Methodology

This section describes the collaboration workflow used for the report, including dedicated worktrees, verification gates, and durable review tracking.

8 Evaluation

TODO §8 (entire section). Do not draft prose for this section until the measurement harness has produced reproducible numbers. The required numbers and the harness scope are listed in `docs/paper/evidence-gaps.md` C2:

1. `cap_enter` round-trip latency, warm and cold ring, single CPU.
2. SQE dispatch throughput in ops/sec, single-process baseline.
3. Endpoint IPC: ordinary scheduler path vs. direct-switch handoff latency.
4. Schema dispatch overhead vs. the existing `NullCap` baseline (compiled in under `--features measure`).
5. Memory cost per capability slot.
6. Optionally: comparison numbers from published seL4 / Zircon literature, where the comparison is fair (same architecture, same workload shape, same caveats stated).

Each figure must be reproducible from a single `make` target and output to a tracked path so the paper can cite a specific commit. Until this section exists, claims of the form "the ring is fast enough" or "schema dispatch is acceptably cheap" must not appear in the introduction or abstract.

Closes when: measurement harness lands and produces a reviewed numbers run.

9 Limitations

This report describes a research operating system. The artifact is useful for evaluating the *design* claims in Section 3 through Section 7; it is not a production system. The honest limitations are:

9.1 Single Architecture

x86_64 only. The aarch64 port is planned (the ring transport, capability model, and userspace runtime are deliberately arch-independent above `arch/`), but no aarch64 boot has been demonstrated. Consequently, claims about the design are claims about how it behaves on x86_64, not on every architecture a capability OS might target.

9.2 Partial SMP

Phase A (BSP per-CPU data) and Phase B (AP startup through Limine MP) are complete. Phase C currently demonstrates a *single* AP running scheduler-owned user contexts (`d88bca7`) while the BSP remains in kernel idle behind a scheduler-owner latch. Full concurrent multi-CPU scheduling — per-CPU run queues, reschedule IPIs, and per-thread ring routing — is open work. Until that lands, all performance claims are single-CPU claims, and the "In-Process Threading Scalability" milestone has not been demonstrated.

9.3 No Persistence

Capabilities exist only at runtime. The Cap'n Proto wire format is designed to support persistence, but no `Store`/`Namespace`-backed capability has been serialized and restored across kernel reboot. This is the single largest gap between the report's design hypothesis (the wire format unifies persistence with IPC) and the artifact.

> **TODO §9.3.** Replace this paragraph with a description of the persistence proof-of-concept once it lands. The PoC's scope is defined in `evidence-gaps.md` C3: a RAM-backed `Store`, one non-trivial capability serialized via its capnp schema, dump, kernel reboot, restore, method call returns expected result. The PoC is paper-evidence, not the production storage subsystem; it is deliberately narrower than `docs/proposals/storage-and-naming-proposal.md`.
>
> **Closes when:** `evidence-gaps.md` C3 lands.

9.4 No Network Transparency

The same gap, in mirrored form. The wire format is designed so that a remote capability call is structurally identical to a local one, but no capability call has crossed a TCP boundary between two capOS instances. The Telnet and SSH gateways prove that capOS can host a remote *terminal* session over TCP; they do not prove that capOS can host a remote *capability invocation* over TCP.

> **TODO §9.4.** Replace with a description of the network-transparency PoC once it lands. The PoC's scope is defined in `evidence-gaps.md` C4: a `RemoteCap` shim forwarding one capnp method call over an established TCP capability between two capOS QEMU guests, or between a guest and a host capnp stub. Paper-evidence, not the production networking subsystem.
>
> **Closes when:** `evidence-gaps.md` C4 lands.

9.5 No Promise Pipelining

Promise pipelining is reserved in the SQE ABI but not implemented. The `pipeline_dep` and `pipeline_field` fields exist precisely so the kernel will not need an ABI break to add it later, but capnp-rpc's characteristic property — a call on a *result* of an outstanding call may be sent before that call returns — is not yet exercised. Until it is, capOS's claim to be "becoming a capnp-rpc router" should be read as a directional commitment, not as a present-tense fact.

9.6 Service-Object Migration In Progress

Section 5.5. The "interface IS the permission" claim is undercut so long as caller-selected endpoint identity still encodes service identity in the existing chat and adventure demos. Delegated relabel containment and the transitional representation substrate have landed; the selected Service Object Identity Migration's routing/lifecycle proof, subject/proof root-open proof, shared-service migration, and legacy cleanup are open.

9.7 Verification Scope

The verification stack is layered and pragmatic, not exhaustive. Multiple kernel subsystems (interrupt handlers, page table edits, MSR setup, AP startup, scheduler-context switch) are not under formal proof. Several panic surfaces remain classified rather than eliminated.

9.8 No Hardware Path

The artifact runs in QEMU. It boots through Limine, exercises virtio devices through QEMU's device models, and its DMA isolation is provided by kernel-owned bounce buffers (see `docs/dma-isolation-design.md`) rather than by IOMMU-isolated typed `DMAPool` capabilities. Real hardware would expose paths the kernel has not yet hardened. This is deliberate: the report makes design claims, not deployment claims.

10 Future Work

The Future Work outlined here is the subset of `docs/roadmap.md` relevant to closing the gaps in Section 9 and to extending the design in directions the research itself motivates. Items here are *directionally* in scope; specific sequencing lives in `WORKPLAN.md` and `docs/paper/plan.md`.

10.1 Persistence Proof-of-Concept

A minimal RAM-backed `Store` capability that exercises the wire-format-as-persistence claim end-to-end (Section 9.3). Not the full storage subsystem, by design.

10.2 Network Transparency Proof-of-Concept

A `RemoteCap` shim forwarding one capnp method call over TCP between two capOS instances (Section 9.4). Naturally rides on the SSH/Telnet networking work.

10.3 Promise Pipelining or Notifications

At least one of {capnp-rpc-style promise pipelining, seL4-style notification objects}. Promise pipelining is closer to the schema-as-ABI thesis; notifications are closer to interrupt delivery.

10.4 Service-Object Migration Completion

The routing/lifecycle proof, subject/proof root-open proof, and shared-service migration in Section 5.5 close the C1 contribution claim's "partial" status. The legacy naming cleanup tidies the codebase but does not change paper claims.

10.5 Full Concurrent SMP

Per-CPU run queues, reschedule IPIs, per-thread capability rings. Replaces the runtime-reactor compatibility bridge with a per-thread ring v2 ABI. Proves a sibling-thread scaling milestone.

10.6 aarch64 Port

After x86_64 hardware abstraction stabilizes. The shared crates (schema, ring, runtime, process/capability abstractions above `arch/`) are designed to carry over; arch-specific work is EL0/EL1 syscall entry, GICv3, ARM generic timer, PL011 UART, TTBR0/TTBR1 MMU, TPIDR_EL1 per-CPU.

10.7 Out-of-Scope For First Paper

The roadmap also includes GPU/CUDA capabilities, live upgrade, cloud metadata, formal MAC/MIC modeling, Go runtime, WASI, and production volume encryption. These are explicit non-goals for this report and are mentioned only to delimit scope. See `docs/roadmap.md` "Future Tracks".

11 Collaboration Methodology

This section describes a review-driven, branch-based process with explicit verification and durable issue tracking.

The relevant practices are:

11.1 Per-Task Worktrees

Every change, including documentation-only changes, lives on a dedicated git branch checked out into a separate git worktree. The main worktree is treated as read-only. This rule has two effects:

- Every change is auditable as a branch-shaped review unit. The human reviewer can read the diff, request changes, and merge — the same loop they would use for a human collaborator on a remote branch.
- Working-directory misunderstandings and partial edits cannot leak into main. Misunderstood instructions, partial edits, or context drift live on a branch and either get fixed or get dropped.

11.2 Mandatory Verification Gates Before Merge

Section 7′s verification stack (`make fmt-check`, `make generated-code-check`, `cargo test-config`, `cargo test-lib`, `cargo test-ring-loom`, `make kani-lib`, the relevant `make run-*` QEMU smoke) is mandatory at task close, not at the human reviewer's discretion. Drafts without these gates can produce plausible proposals that fail to compile or fail to boot; with them, the gate output is a deterministic signal that the change at least did not break what it claimed to leave alone.

11.3 Durable Open Findings

`REVIEW_FINDINGS.md` is the live, durable list of unresolved review issues. The discipline is that any review observation that is not fixed in the same task gets a `REVIEW_FINDINGS.md` entry with context and remediation pointer. This survives session boundaries: the next session, started cold, reads `REVIEW_FINDINGS.md` and treats those findings as live. Without this, session memory of past review feedback is unreliable across sessions.

11.4 Workplan and Roadmap Sync

`WORKPLAN.md` and `docs/roadmap.md` are read at session start and updated as part of the same task that changes priorities. The rule is explicit: "a stale plan is worse than no plan." For this workflow, process continuity at session N+1 follows the documents, not the conversation log of session N, so any priority change made conversationally must be written into the documents in the same task or it does not survive.

11.5 Timestamps with Minute Precision

Status updates in `WORKPLAN.md`, `REVIEW_FINDINGS.md`, and roadmap files use minute-precision timestamps with timezone abbreviations (e.g. `2026-04-25 11:31 UTC`). Bare dates correlate poorly with commit history when multiple updates land the same day. The rule exists because stale context tends to drift; an explicit `date '+%Y-%m-%d %H:%M %Z'` invocation produces the actual time and makes the log auditable.

11.6 What This Methodology Does *Not* Solve

Honest framing: process scaffolding is not equivalent to having more engineers. Process scaffolding does not independently judge whether a design decision is wise on a multi-year timescale, and it does not push back against its own previous output as reliably as a human reviewer. It is particularly weak at noticing absence — a missing gate, a missing test, a quietly downgraded task. The discipline above is structured so that those weaknesses manifest as visible artifacts (a missing `REVIEW_FINDINGS.md` entry, a stale `WORKPLAN.md`, a docs/code drift flagged by `make generated-code-check`) rather than as silent correctness gaps.

> **TODO §11.6.** A retrospective on which classes of error this methodology actually caught (vs. which slipped through and how) is worth a short subsection. Source material lives in the git log and `REVIEW_FINDINGS.md` history. Write after at least one Tier-1 evidence cycle has run end-to-end, so the retrospective covers the same workflow the rest of the report describes.
>
> **Closes when:** at least one of `evidence-gaps.md` C2 / C3 / C4 has been delivered through this workflow and post-merge review is complete.

12 Conclusion

TODO §12. Write last, after Section 8 and §9.6 close. The conclusion should restate the schema-as-ABI thesis (§1), connect it to the partially-closed contributions (C1 once the service-object migration lands, C2 once the measurement section lands, C5 once the Tier-2 strengtheners land), acknowledge the open hypotheses (C3 persistence, C4 network transparency) and the future work that would close them, and state honestly which design hypotheses the artifact has tested and which it has not. No new claims.

Closes when: all Tier-1 evidence in `evidence-gaps.md` has landed.

Appendix A. Schema Excerpt

TODO Appendix A. Pick one representative capability and present its full schema with a method-by-method ABI commentary. Candidate: `MemoryObject` (illustrates bulk-data semantics) or `TerminalSession` (illustrates request/response with both visible and hidden echo). Source: `schema/capos.capnp`.

Appendix B. Reproducibility Kit

TODO Appendix B. Exact commands, pinned commits, expected outputs. Source: `docs/build-run-test.md`, `docs/trusted-build-inputs.md`, the Makefile. Should fit on one printed page.

Appendix C. Build and Boot Flow

TODO Appendix C. Diagram of the boot path: Limine to kernel `kmain` to manifest validation to init/shell launch to CapSet page mapping to ring page mapping to first user CALL. Source: `docs/architecture/boot-flow.md`, `docs/architecture/manifest-startup.md`.

Appendix D. Process Model

TODO Appendix D. Diagram of process state, capability table, ring page, and address space. Source: `docs/architecture/process-model.md`.

References