DEFENSIVE PUBLICATION — TECHNICAL DISCLOSURE (TDCOMMONS DEPOSIT COPY) · 2026-06-25

Keywords: defensive-publication, prior-art, ai-agents, microkernel, boot-architecture, dependency-injection, topological-sort, fail-fast

Tiered-Boot Capability Kernel for an AI-Agent Platform

A Technical Defensive Publication

Subtitle: A microkernel-style boot assembler in which every module — operating system, AI-inference runtime, and business domain alike — shares one uniform lifecycle contract; a topological fail-fast boot loads numbered capability tiers such that Tier 0 is an operable, health-serving platform with no AI and no business logic, and the AI runtime is its own distinct tier.

Field	Value
Title	Tiered-Boot Capability Kernel for an AI-Agent Platform
Publisher / Copyright holder	Gus IT LLC (Florida, USA)
Author	Gustavo Assuncao, PhD
Publication date	2026-06-25
Version	1.0
Document type	Technical Defensive Publication (public prior art)
Classification	Public
License	AGPL-3.0-or-later (copyleft; commercial license available)
Deposit channel	To be assigned (IP.com / Zenodo / arXiv) — establishes a public, dated, citable prior-art record. No DOI is asserted at time of writing.
Field	Boot and module-lifecycle architecture for large modular AI-agent platforms.

Abstract

Large AI-agent platforms aggregate dozens to hundreds of interdependent modules spanning operating-system services, an AI-inference runtime, and business domains. A conventional monolith or undifferentiated plugin host gives no guarantee that such a platform boots deterministically, fails fast on an unmet dependency, or remains operable and manageable when AI or business logic is absent or broken. Critically, when the heavy, fast-iterating AI runtime crashes during startup, it frequently takes the entire process down with it — including the very health endpoint operators need to diagnose the failure. This publication describes a boot architecture that resolves the problem through three combined mechanisms. First, a **single uniform module-lifecycle contract** — `{id, category, requires, optional, register, bootstrap, shutdown, health, emits, listens, config}` — is applied *identically* across every module, regardless of whether it is an OS service, the AI runtime, or a business domain. Second, the assembler computes a **topological ordering** of modules from their declared required/optional dependencies and

refuses to boot (fail-fast) when any required dependency is unmet. Third, modules load in **numbered capability tiers** with AI-platform-specific semantics: a first tier (Tier 0) instantiates a kernel that exposes an aggregated health endpoint and is fully operable and manageable while *no* AI-inference runtime and *no* business-domain module is loaded; the AI runtime is loaded as its **own distinct tier** (Tier 2), gated above OS/policy services (Tier 1) and below optional business domains (Tier 3). A later-tier module that fails its declared dependency is **isolated** without preventing lower tiers from remaining operable and health-reporting, and per-domain Postgres schema isolation is bound to module identity. The platform thereby attains and reports a *verified operable health state* before, and independent of, loading any AI or domain logic; a broken later-tier module cannot take down the kernel. The document is published to establish dated, citable prior art and to seed an open-source reference application.

---

1. Executive Summary

1.1 Thesis

A modular AI-agent platform should be **operable before it is intelligent.** The boot process should *guarantee* — not merely hope — that the kernel, its health endpoint, and its management surfaces come up and report a verified-operable state *before* the expensive, fragile AI-inference runtime is even loaded, and *independent* of whether any business domain ever loads. The mechanism that delivers this guarantee is a microkernel-style assembler that combines (a) one uniform lifecycle contract across *all* module classes, (b) a topological, fail-fast dependency boot, and (c) numbered capability tiers in which the AI runtime is an explicitly distinct tier sandwiched between OS/policy services and optional business domains.

1.2 Contributions

#	Contribution	Where
C1	A **uniform lifecycle contract** used identically by OS, AI-runtime, and business-domain modules — eliminating the usual "the AI layer is special" divergence.	§6, §7
C2	**Topological fail-fast assembly**: the boot computes a dependency order and *refuses to proceed* when a required dependency is missing, rather than crashing later or booting half-formed.	§7.2, §7.4
C3	**Numbered capability tiers with AI-platform semantics**: Tier 0 is a fully operable, health-serving kernel with zero AI/domain logic.	§6.2, §7.3
C4	**AI runtime as its own distinct tier (Tier 2)**, gated above Tier 1 (OS/policy) and below Tier 3 (optional business domains).	§6.2, §7.3
C5	**Later-tier fault isolation**: a Tier-2/Tier-3 module that fails its declared dependency is quarantined; lower tiers stay up and health-reporting.	§7.5, §11
C6	**Identity-bound schema isolation**: each business-domain module's Postgres schema is bound to its module id, so domain data isolation tracks module boundaries.	§8
C7	A **clean-room reference implementation** that reduces the combination to practice in <300 lines of dependency-light JavaScript.	§9, Appendix C

1.3 Headline claim

> *A boot method for a multi-module AI-agent platform in which a topological, fail-fast assembler loads modules in numbered capability tiers such that a Tier-0 kernel attains and reports a verified-*

> *operable, aggregated-health state while no AI-inference runtime and no business-domain module is loaded, the AI runtime being loaded as its own distinct subsequent tier, and a later-tier module's failure being isolated from the operability of lower tiers.*

1.4 Scope — what this is and is NOT

It IS: a description of a boot/lifecycle architecture; the combination of a uniform contract + topological fail-fast boot + AI-runtime-as-its-own-tier + Tier-0 operability-before-AI + later-tier isolation; a clean-room reference assembler.

It is NOT: a claim over topological sorting, dependency injection, plugin hosts, health-check endpoints, or staged init in the abstract — all of which are well-trodden prior art (OSGi, Spring Boot, systemd, ABP). It is NOT a runtime scheduler, a container orchestrator, a service mesh, or an AI model. It does NOT claim "AIOS / LLM as the operating system"; on the contrary, it deliberately makes the OS *operable without* the LLM. It is NOT a claim of certified compliance with any standard (§12 states *semantic alignment* only).

---

2. Introduction & Motivation

2.1 The concrete problem

Consider a platform composed of ~170 modules: kernel primitives (config, registry, event bus, scheduler), shared services (database, auth, audit, cache, sessions, secrets), policy and AI services (access gate, token/LLM routing, tool executor, persona runtime), runtime/management surfaces (app registry, route proxy, health monitor, admin UI), infrastructure providers (Postgres, object storage, email, messaging), and a tail of optional business domains (CRM, comms, documents, billing).

In a naïve startup, all of these are wired together in a hand-ordered initialization script or a single dependency-injection container that constructs *everything* before serving traffic. Two failure shapes follow:

1. **The AI tax on availability.** The AI-inference runtime is the heaviest, most frequently changed, and most failure-prone subsystem: it loads model clients, warms caches, validates provider credentials, and opens long-lived connections. When *it* throws during boot — a bad model id, an expired key, a provider outage — a monolithic container aborts the *entire* boot. The platform never reaches a state where operators can log in, read `/health`, see *why* it failed, or flip a feature flag. The thing most likely to break is wired in series ahead of the thing you need to diagnose it.
2. **The silent half-boot.** An undifferentiated plugin host that swallows module errors can boot "successfully" with a required dependency missing — e.g., the audit module loaded but the database it depends on did not. The platform appears up but is subtly broken, and the defect surfaces later as data loss or an unhandled exception under load.

2.2 The "tax" being paid

Symptom	Underlying cause	Cost
Whole platform down during an AI provider outage	AI runtime constructed in series with the kernel	Lost availability + blind operators (no health endpoint)
"It booted but auth is broken"	Required dependency silently unmet	Latent defects, data-integrity risk
Slow, all-or-nothing restarts	No tier boundary to bring up management first	Long MTTR, panicked rollbacks
Cross-domain data bleed	Domain isolation not bound to module identity	Compliance / tenancy risk
"The AI layer is special" divergence	AI runtime uses a different lifecycle than the rest	Duplicate health/shutdown logic, drift

2.3 Why existing approaches fall short

OSGi/Equinox, Spring Boot, ABP, and systemd each solve *parts* of this (see §4), but none combines, for an AI-agent platform, (a) one contract spanning OS + AI + domains, (b) a fail-fast topological boot, (c) an explicit **Tier-0-operable-before-AI** guarantee, and (d) **AI-runtime-as-its-own-tier** with later-tier fault isolation. The novelty here is the *combination and the AI-platform-specific tier semantics*, not any single ingredient.

2.4 Why now

In 2024–2026, agent platforms ballooned from a handful of services into large module graphs with a dominant, fast-changing AI runtime. The operational reality — provider outages, model-id churn, key rotation — makes "operable before intelligent" a first-order availability requirement rather than an academic nicety. Publishing the combined technique now keeps it free for the whole field to use.

---

3. Problem Statement

3.1 Formal framing

Let `M = {m_1, …, m_n}` be the platform's modules. Each module `m_i` declares a set of **required** dependencies `req(m_i) ⊆ M` and **optional** dependencies `opt(m_i) ⊆ M`, and exposes lifecycle hooks `register`, `bootstrap`, `shutdown`, and `health`. Each module is assigned a **tier** `t(m_i) ∈ {0, 1, 2, 3}` by category. We require a boot procedure `B(M)` such that:

- **(Determinism)** `B` loads modules in an order `σ` that is a topological sort of the dependency DAG; if `req` induces a cycle, `B` aborts with a precise diagnostic.
- **(Fail-fast)** If `∃ m_i, d ∈ req(m_i)` with `d ∉ M`, `B` refuses to proceed past validation — no partial boot.
- **(Tier-0 operability)** After loading exactly the Tier-0 set `T_0`, the platform is *operable*: it serves an aggregated health endpoint `H()` that reports the live state of every loaded module, and it accepts management operations — with *no* AI-runtime module and *no* business-domain module loaded.
- **(AI tier distinctness)** The AI-inference runtime constitutes Tier 2, loaded strictly after Tier 1 (OS/policy) and strictly before Tier 3 (optional domains).

- **(Later-tier isolation)** If a module in tier `≥ 2` fails `register/bootstrap` or reports `down`, that module is quarantined; modules in lower tiers remain operable and continue to report health.
- **(Identity-bound isolation)** A business-domain module's persistent storage namespace (e.g., a Postgres schema) is bound to its module id.

3.2 Derived requirements (R1..R10)

ID	Requirement	Satisfied in
R1	One identical lifecycle contract across OS, AI-runtime, and domain modules.	§6.1, §7.1
R2	Topological ordering computed from declared dependencies; cycles abort with a named diagnostic.	§7.2
R3	Fail-fast: any unmet *required* dependency refuses boot before side effects.	§7.4
R4	Numbered tiers; Tier 0 is operable + health-serving with zero AI/domain logic.	§6.2, §7.3
R5	AI-inference runtime is its own distinct tier (Tier 2), gated above Tier 1, below Tier 3.	§6.2, §7.3
R6	A later-tier module failure is isolated; lower tiers stay operable + reporting.	§7.5, §11
R7	Aggregated health endpoint that surfaces per-module status and a worst-case rollup.	§7.6, §9
R8	Optional dependencies degrade gracefully (present → wired; absent → module still loads).	§7.2, §7.5
R9	Per-domain storage isolation bound to module identity.	§8
R10	Reverse-order graceful shutdown honoring the same contract.	§7.7

4. Related Work & Prior Art

This work **builds on** an explicit, well-known body of modular-systems engineering. We name the closest sources and the rationale for adopting their ideas.

Source	Relevance / what we adopt
OSGi Declarative Services / Eclipse Equinox	The canonical Java module system: components declare provided/required services and the runtime wires them by dependency. We adopt the *declarative dependency* idea and the *uniform component lifecycle*.
ABP Framework `[DependsOn]` module graph	.NET modular framework that orders module initialization by a declared dependency attribute. We adopt *declared-dependency ordering at boot*.
Spring Boot auto-configuration + Spring Actuator	`@AutoConfigureAfter`/`@ConditionalOn…` ordering and a `/actuator/health` endpoint that *aggregates* component health. We adopt *aggregated health rollup*.
SysV runlevels / systemd targets	Staged init: bring the machine to a numbered runlevel/target where a defined set of units is active. We adopt *numbered staged bring-up*.
"Staged IPL" (US 5,379,431)	Initial-program-load in defined stages. We adopt *staged load with stage gating*.
Microkernel OS design (Mach, L4, MINIX)	A minimal kernel that is operable on its own, with services as separable, restartable components. We adopt *the operable-minimal-core principle* and *fault isolation of non-kernel services*.
Erlang/OTP supervision trees	Let-it-crash isolation: a failing child is restarted/quarantined without killing the supervisor. We adopt *isolation of a failing subordinate*.
Kubernetes init containers / readiness vs liveness	Separating "is the process alive" from "is it ready to serve." We align our Tier-0 health with *operable-before-ready* semantics.
AIOS (LLM-as-OS, COLM 2025)	An "LLM as the operating system kernel" proposal. We adopt its *AI-platform framing* but **invert** its architecture: here the OS is deliberately operable *without* the LLM, and the LLM is a gated tier — the opposite of putting the LLM at the kernel.

We do not claim novelty over any of the above individually. §5 isolates exactly what each lacks relative to the present combination.

---

5. Prior-Art Delta

The following table is the crux of the disclosure: for each novel feature, what the closest prior source *has*, what it *lacks*, and what *this* adds.

Prior source	What it has	What it LACKS	What THIS adds
OSGi / Equinox DS	Declarative required/provided services; per-component lifecycle	No platform-level *numbered capability tiers*; no notion of an *AI-inference runtime as a distinct, gated tier*; no *operable-Tier-0-before-AI* guarantee	One contract spanning OS + **AI runtime** + business domains, with AI-runtime as an explicit tier above OS and below domains
ABP `[DependsOn]`	Declared module dependency ordering at init	Boot is application-monolithic; no Tier-0-operable-without-domains guarantee; no AI-runtime tier; failures abort the app	Fail-fast topological boot **plus** a Tier-0 that serves health with *zero* domain logic; later-tier isolation
Spring Boot + Actuator	Conditional ordering; aggregated `/health`	No explicit capability tiers; no "operable before the heaviest subsystem (AI) loads"; AI not modeled as a tier	A health rollup whose *verified-operable* status is reached and reported **before** the AI tier loads
systemd targets / runlevels	Numbered staged bring-up of unit sets	OS-process granularity, not application module graph; no *AI-runtime-as-a-tier*; no per-module health contract or app-domain isolation	Application-level numbered tiers with an **AI-runtime tier** and a per-module `health()` contract feeding an aggregated rollup
US 5,379,431 (staged IPL)	Staged load with stage gating	Hardware/firmware IPL; no AI semantics; no uniform application module contract; no later-tier isolation tied to dependency declarations	AI-platform tier semantics + dependency-declared fail-fast + later-tier fault isolation that keeps the kernel up
Microkernel OS / OTP	Minimal operable core; isolate failing services	Not specialized to an AI-agent module graph; no *AI runtime as a named tier between OS and business domains*; no declared-dependency topological app boot	The *combination*: microkernel-operable Tier-0 **for an AI platform**, with AI as Tier 2 and domains as Tier 3, declared-dependency fail-fast boot
AIOS (LLM-as-OS)	AI-platform framing; LLM-centric kernel	Puts the LLM *at* the kernel — the platform is non-operable without the model; the inverse of fault isolation for the AI layer	**Inverts** it: OS is operable *without* the LLM; the LLM is a gated, isolatable tier; verified-operable health *before* AI

The combined claim — uniform contract + topological fail-fast + Tier-0-operable + AI-runtime-as-its-own-tier + later-tier isolation + identity-bound schema isolation, *as applied to an AI-agent platform* — is the novel composition this document places into the prior-art record. (See §14 for the honest patentability assessment.)

---

6. System Architecture

6.1 Components

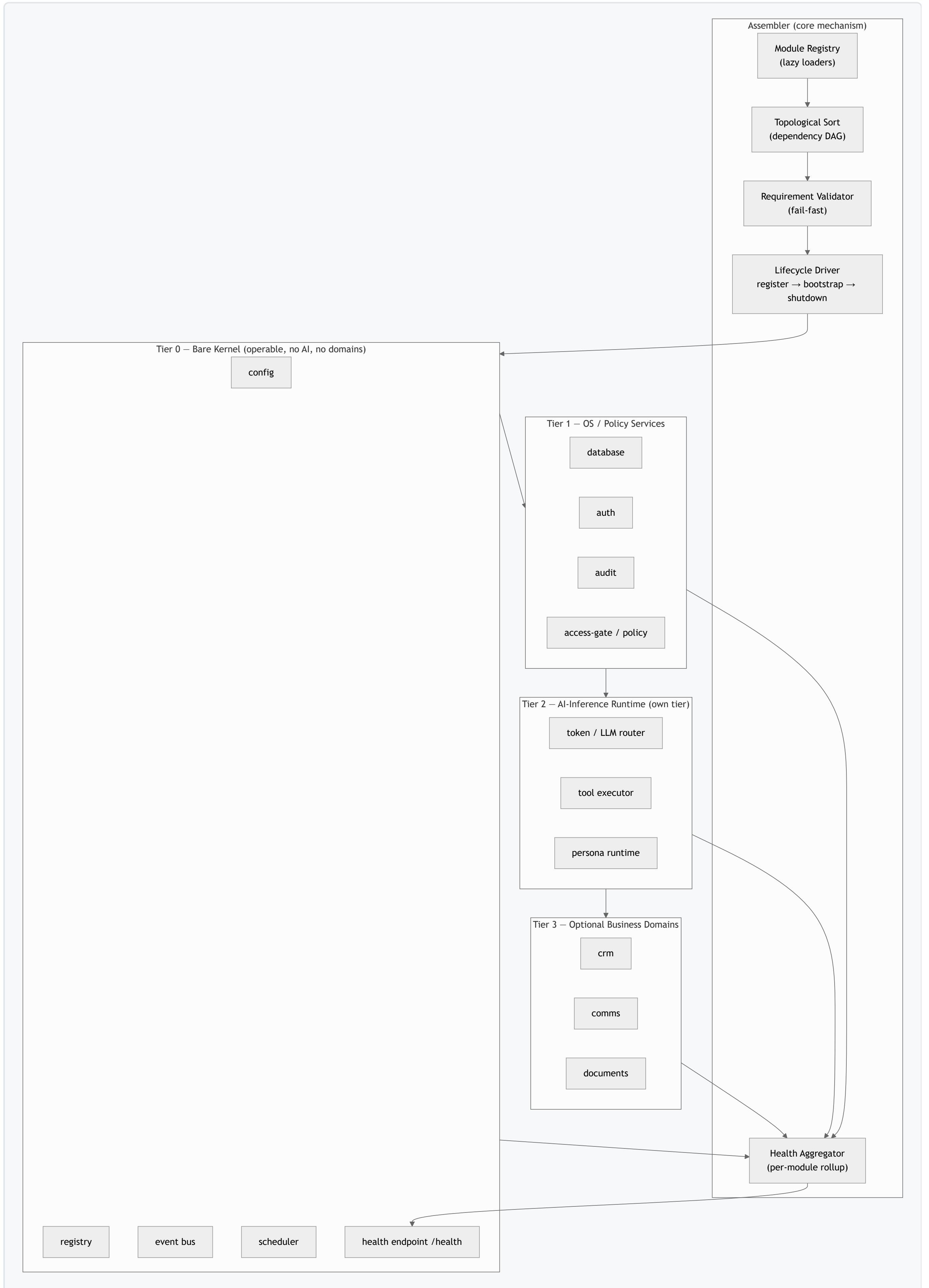

Assembler (core mechanism)
Module Registry
(lazy loaders)
Topological Sort
(dependency DAG)
Requirement Validator
(fail-fast)
Lifecycle Driver
register → bootstrap →
shutdown
Tier 0 — Bare Kernel (operable, no AI, no domains)
config
Tier 1 — OS / Policy Services
database
auth
audit
access-gate / policy
Tier 2 — AI-Inference Runtime (own tier)
token / LLM router
tool executor
persona runtime
Tier 3 — Optional Business Domains
crm
comms
documents
Health Aggregator
(per-module rollup)
registry
event bus
scheduler
health endpoint /health

Figure 1 — Component architecture: a single assembler drives one uniform contract across four numbered tiers; the health aggregator feeds the Tier-0 health endpoint.

The assembler has five parts:

- **Module Registry** — a map of module ids to *lazy* loaders. Lazy loading matters: a Tier-2/Tier-3 module is not even imported until its tier is reached, so a syntax or import error in the AI runtime cannot crash Tier-0 bring-up.
- **Topological Sort** — builds the dependency DAG from each module's `requires` and emits a load order; detects and names cycles.
- **Requirement Validator** — checks that every *required* dependency is registered; if not, boot is refused (fail-fast) *before* any `bootstrap` side effects run.
- **Lifecycle Driver** — invokes `register` (declare routes/capabilities), then `bootstrap` (start background work), in tier/topological order; `shutdown` in reverse.
- **Health Aggregator** — calls each module's `health()` and rolls up a worst-case platform status (`ok` → `degraded` → `down`), surfaced at the kernel health endpoint.

6.2 Capability tiers

Tier	Name	Contains	Operability invariant
0	Bare kernel	config, registry, event bus, scheduler, health endpoint	**Operable and health-serving with no AI and no domains.** This is the microkernel.
1	OS / policy services	database, auth, audit, cache, sessions, secrets, access-gate	Adds persistence, identity, policy; still no AI, no domains.
2	**AI-inference runtime**	token/LLM router, tool executor, persona runtime, RAG	**Its own distinct tier.** Heavy, fast-changing, failure-prone — and *isolated*.
3	Optional business domains	crm, comms, documents, billing	Optional; each bound to its own storage schema; a failure here is isolated.

The **tier assignment is by category**, not by hand-ordering each module: a module's `category` (kernel / shared-service / policy-AI / runtime / domain) maps to a tier, and the topological sort orders modules *within and across* tiers consistent with declared dependencies. A required dependency may never point "upward" to a higher tier.

6.3 Cross-cutting properties

- **Determinism** — same module set ⇒ same load order (stable topological sort).
- **Laziness** — higher-tier modules are imported only when their tier is reached.
- **Idempotent health** — `health()` is side-effect-free and may be polled repeatedly.
- **Symmetric shutdown** — modules tear down in reverse of boot order.
- **Configurability** — the enabled module set, and thus which tiers/domains load, is driven entirely by configuration (env / config bridge / values file), not hardcoded.

---

7. Detailed Mechanics

7.1 The uniform lifecycle contract (R1)

Every module — OS, AI, or domain — exports an object of the same shape:

```
Module := {
  id        : string           // unique identifier, e.g. "database", "token-router", "crm"
  category  : "K"|"S"|"P"|"R"|"D"// maps to a tier (kernel/shared/policy-AI/runtime/domain)
  requires  : string[]         // ids of REQUIRED dependencies (fail-fast if unmet)
  optional? : string[]         // ids of nice-to-have dependencies (degrade gracefully)
  emits?    : string[]         // event names published
  listens?  : string[]         // event names consumed
  config?   : object           // declarative config schema
  register  : (kernel) => void  // Phase A: declare routes/middleware/capabilities
  bootstrap : (kernel) => void  // Phase B: start background work / connections
  shutdown  : (kernel) => void  // Phase C: graceful cleanup (reverse order)
  health    : () => HealthState // { status: "ok"|"degraded"|"down", ...metrics }
}
```

Because the AI runtime and a CRM domain implement the *same* contract as the database kernel, the assembler needs no special case for "the AI layer." This is what makes Tier-0-before-AI a *structural* guarantee rather than a convention.

7.2 Topological sort with optional edges (R2, R8)

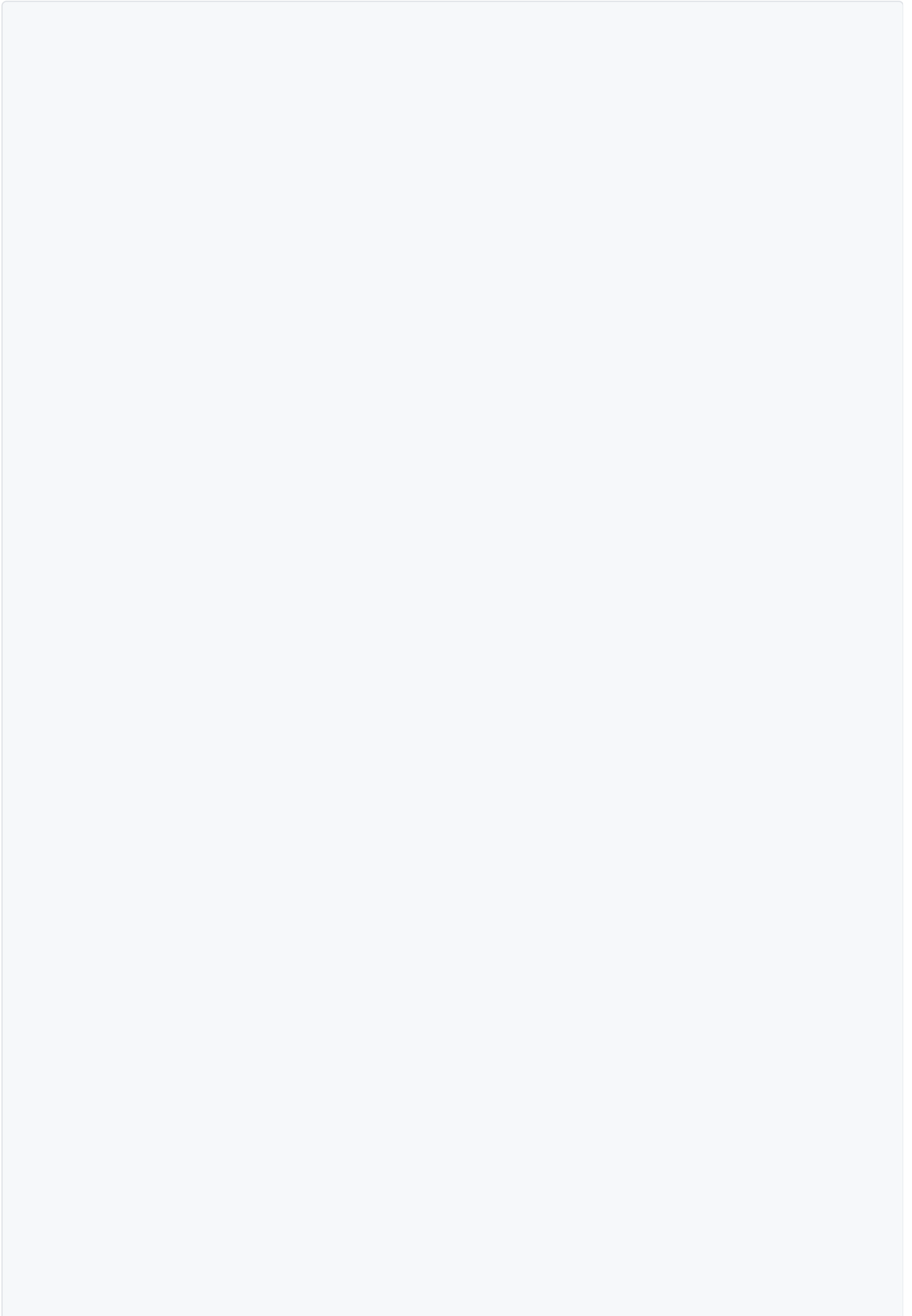
The DAG is built only from `requires` (hard edges). `optional` dependencies do **not** create ordering constraints that can refuse boot — they are wired *if present* and skipped *if absent*, so an optional edge degrades gracefully.

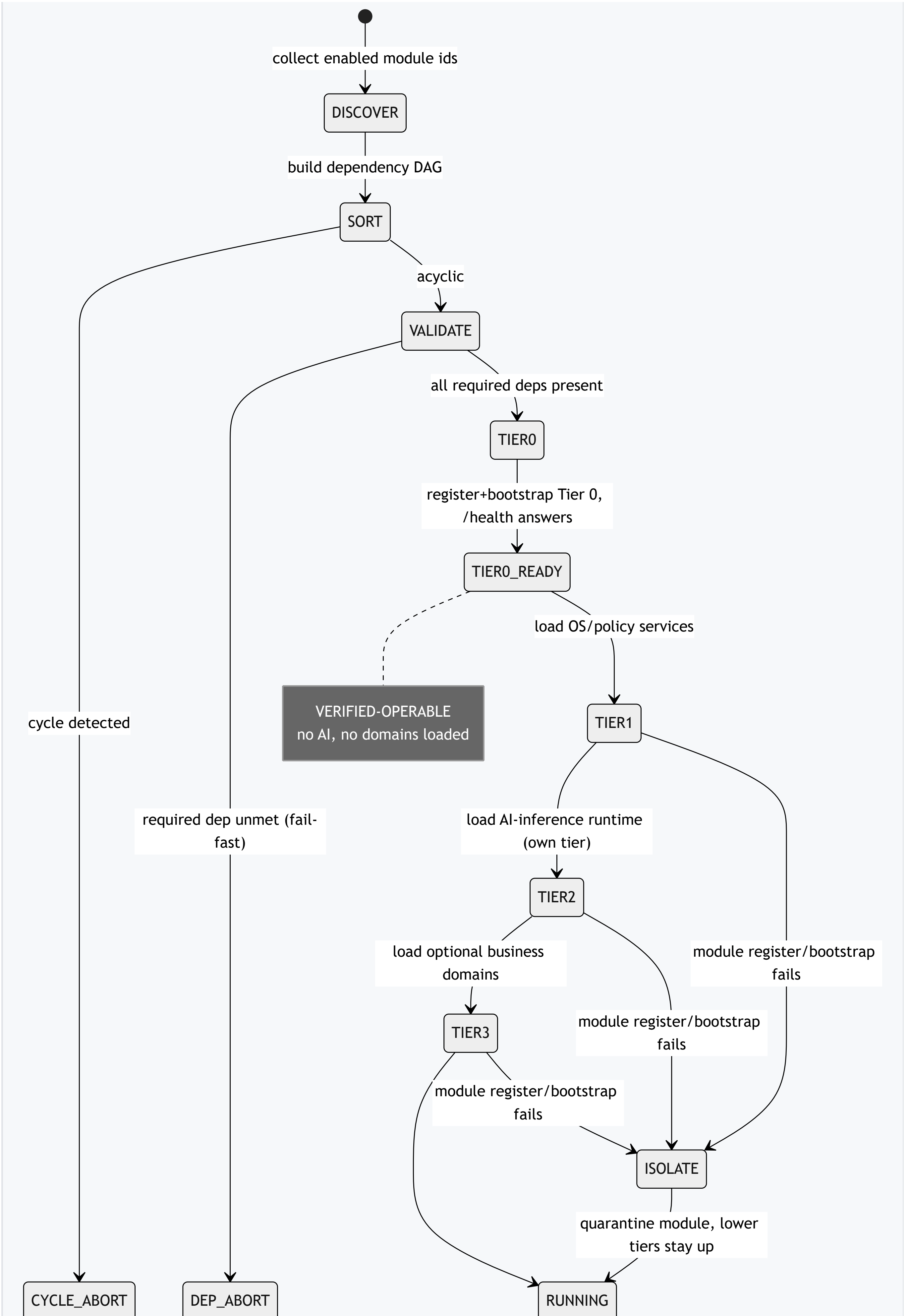
```
function topologicalSort(modules):
    sorted   = []
    visited  = set()
    visiting = set()           // for cycle detection

    function visit(id):
        if id in visited: return
        if id in visiting:
            throw CycleError("circular dependency involving " + id)
        visiting.add(id)
        m = modules[id]
        if m:
            for dep in m.requires:          // ONLY hard edges
                if dep in modules: visit(dep)
        visiting.remove(id)
        visited.add(id)
        if m: sorted.append(m)

    for id in modules.keys(): visit(id)
    return sorted               // dependencies precede dependents
```

The depth-first visit with a `visiting` frontier yields O(V+E) sorting and detects cycles with a *named* module in the error — a precise diagnostic, not a stack overflow.

7.3 Tier-ordered loading (R4, R5)

After the topological order is computed, modules are partitioned by tier (from `category`) and loaded **tier-by-tier, ascending**, preserving the topological order within each tier. The boot reaches a **Tier-0 ready barrier**: once Tier 0's modules are registered and bootstrapped and the health endpoint answers, the platform records a *verified-operable* state. Only then does loading advance to Tier 1, then the **AI runtime Tier 2**, then optional Tier 3.




collect enabled module ids
DISCOVER
build dependency DAG
SORT
acyclic
VALIDATE
all required deps present
TIER0
register+bootstrap Tier 0, /health answers
TIER0_READY
load OS/policy services
VERIFIED-OPERABLE
no AI, no domains loaded
cycle detected
TIER1
required dep unmet (fail-fast)
load AI-inference runtime (own tier)
TIER2
load optional business domains
module register/bootstrap fails
TIER3
module register/bootstrap fails
module register/bootstrap fails
ISOLATE
quarantine module, lower tiers stay up
CYCLE_ABORT
DEP_ABORT
RUNNING

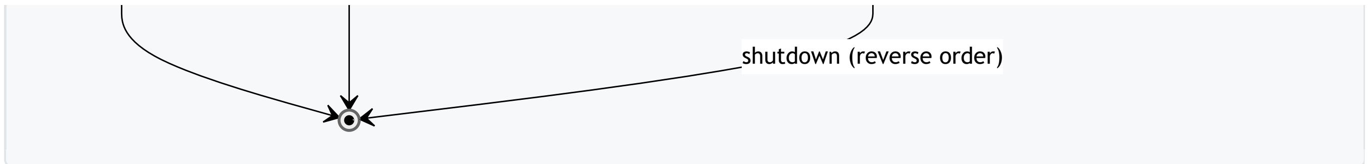


Figure 2 — Boot state machine. The TIER0_READY state is the verified-operable barrier reached before any AI or domain logic; later-tier faults route to ISOLATE, not to process exit.

7.4 Fail-fast requirement validation (R3)

```
function validateRequirements(sortedModules, moduleMap):
    errors = []
    for m in sortedModules:
        for dep in m.requires:
            if dep not in moduleMap:
                errors.append(m.id + " requires " + dep + " (not registered)")
    if errors:
        throw UnmetDependencyError(errors)   // refuse boot BEFORE bootstrap side effects
```

Validation runs *after* the sort but *before* any `register/bootstrap`, so the platform never performs side effects (open a port, write a row) on the way to an inevitable failure. The error lists *every* unmet dependency at once, not just the first — operators fix the whole graph in one pass.

7.5 Tier-differentiated fault handling (R6, R8)

Failure handling depends on the tier of the failing module:

Failing module tier	`register/bootstrap` throws →	`health()` reports `down` →
Tier 0 (kernel)	**Fatal**: a broken kernel is unrecoverable; abort boot with a precise error.	Rolls up to platform `down`; operators alerted.
Tier 1 (OS/policy)	**Fatal by default** for hard-required services (e.g., DB), but a *policy-optional* service may be quarantined.	`degraded` if optional, `down` if required.
Tier 2 (AI runtime)	**Isolated**: quarantine the module, mark the AI subsystem `degraded/unavailable`, keep Tier 0/1 up and health-serving.	Platform `degraded`; AI features fail closed/open per policy; kernel stays up.
Tier 3 (domain)	**Isolated**: quarantine the one domain; other domains and lower tiers unaffected.	Platform `degraded` (that domain `down`); the rest stays `ok`.

This asymmetry is the heart of "a broken later-tier module cannot take down the kernel." Tier-0/Tier-1-required failures are fatal *by design* (you cannot run without a database). Tier-2/Tier-3 failures are *isolated* — exactly the modules most likely to break (AI, business logic) are the ones whose failure is contained.

7.6 Health aggregation (R7)

```
function aggregateHealth(kernel):
    result = { status: "ok", modules: {} }
    for (id, m) in kernel.loadedModules:
        h = m.health ? m.health() : { status: "unknown" }
        result.modules[id] = h
        if h.status == "down":      result.status = "down"
        elif h.status == "degraded" and result.status != "down":
                                     result.status = "degraded"
    return result            // worst-case rollup + per-module detail
```

The rollup is **worst-case monotone**: any `down` makes the platform `down`; any `degraded` (absent a `down`) makes it `degraded`. Because Tier 0 alone produces a complete, answerable rollup, the verified-operable barrier in §7.3 is *observable*.

7.7 Graceful shutdown (R10)

Shutdown walks the loaded modules in **reverse** boot order, calling `shutdown(kernel)` on each and tolerating per-module shutdown errors (logged, not fatal) so one stuck domain cannot block draining the kernel.

7.8 Configuration points

Config key	Effect
`enabledModules`	The set of module ids to load — drives which tiers/domains come up. Absent ⇒ all known modules.
`tierMap` (category→tier)	Overrides the default category-to-tier mapping.
`failFast`	If `false`, demotes unmet *optional* dependency wiring to warnings (required deps always fail-fast).
`healthPollIntervalMs`	Cadence at which the aggregator refreshes cached module health.
Per-domain `schema`	The Postgres schema bound to a domain module's id (§8).

---

8. Data Model

The assembler is largely in-memory, but two persistent structures matter: the **module registry record** (what the platform knows about each module) and the **identity-bound domain schema** (R9).

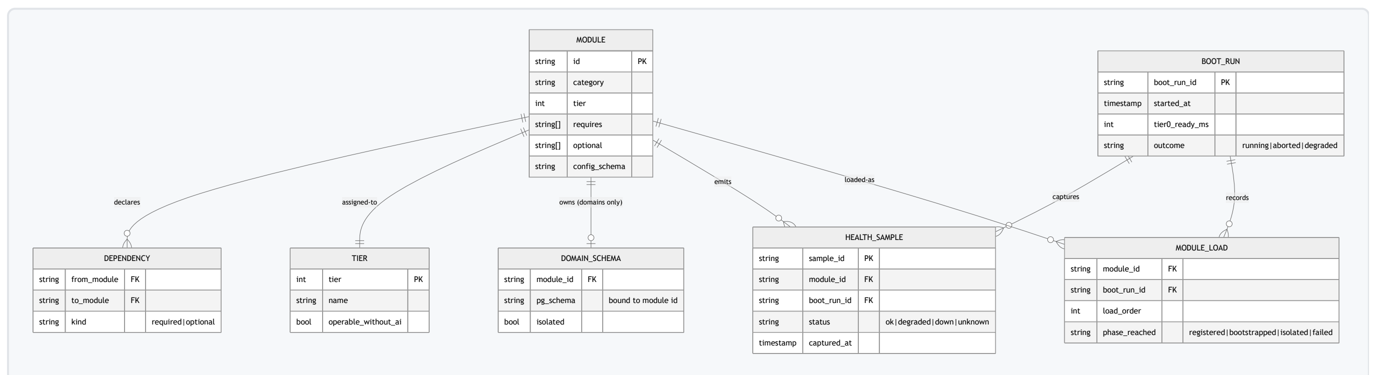


Figure 3 — Data model. A domain module owns exactly one Postgres schema bound to its id (identity-bound isolation). A `BOOT_RUN` records `tier0_ready_ms` — the timestamp of the verified-operable barrier —

and per-module load outcomes including `isolated`.

Identity-bound schema isolation (R9). Each Tier-3 domain module `d` owns a Postgres schema named for `d.id` (e.g., module `crm` ⇒ schema `crm`). The binding is 1:1 and established at registration: the assembler grants the domain access to *only* its own schema, so domain data isolation tracks the module boundary. A domain that fails to load never has its schema activated, and a quarantined domain's schema is left intact and untouched by other domains.

---

9. Reference Implementation & Enablement

The `src/` directory contains an **original, clean-room** reference implementation — roughly 250 lines of dependency-light JavaScript (Node, standard library only) — that reduces the full combination to practice:

- `src/assembler.js` — the assembler: lazy registry, topological sort with cycle detection, fail-fast requirement validation, tier-ordered loading with the Tier-0 ready barrier, tier-differentiated fault isolation, health aggregation, and reverse shutdown.
- `src/modules.js` — a synthetic 12-module platform spanning all four tiers, including a deliberately broken Tier-2 "AI" module and a deliberately broken Tier-3 domain to demonstrate isolation.
- `src/demo.js` — an executable demonstration + self-check that boots the synthetic platform, asserts Tier-0 is operable and health-serving *before* the AI tier loads, injects a Tier-2 failure and asserts the kernel stays up, and prints the health rollup.

How it reduces the invention to practice. Running `node src/demo.js` prints the load order, the moment Tier-0 becomes verified-operable (with no AI/domain loaded), the isolation of the broken AI module, and the final health rollup showing the kernel `ok` while the AI subsystem is `degraded`. The self-check exits non-zero if any invariant (R3, R4, R5, R6, R7) is violated — making the enablement *testable*.

Configuration points are surfaced as a config object passed to `boot()`: `enabledModules`, `tierMap`, `failFast`. Nothing is hardcoded; the same code boots a 2-module or a 200-module platform.

The reference code is **illustrative and clean-room** — it does not reproduce any production source, contains no secrets, hostnames, or proprietary logic, and is not intended for production use as-is.

---

10. Worked Example / Scenario

Scenario. An operator restarts the platform during a third-party LLM provider outage. The AI runtime (Tier 2) will fail to validate its provider credentials and throw during `bootstrap`. We want: the kernel up, the operator able to read `/health` and see *exactly* which subsystem is down, and every non-AI domain serving normally.

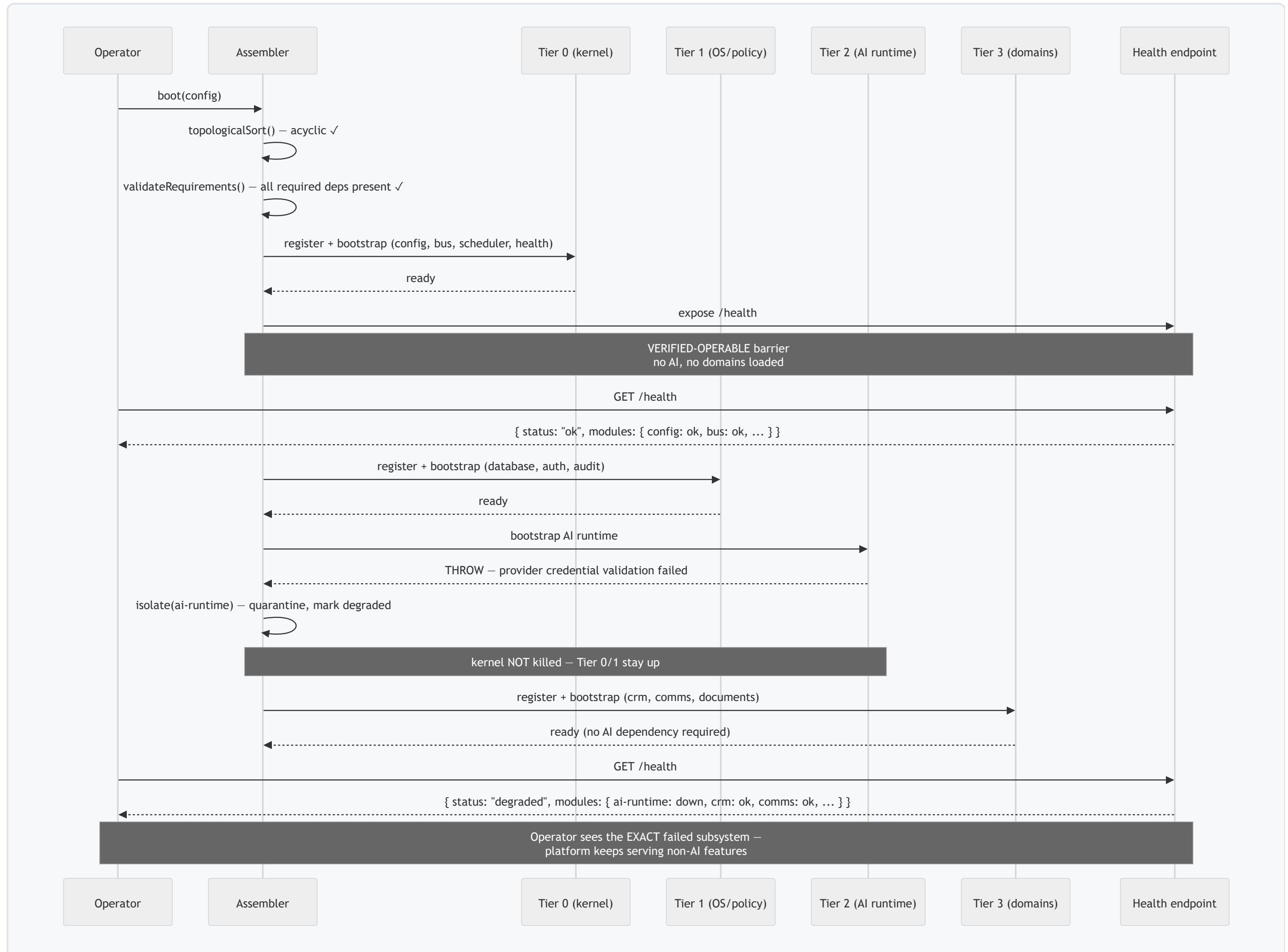


Figure 4 — Worked example: an AI-provider outage isolates Tier 2 while Tier 0/1/3 stay operable and the health endpoint pinpoints the failed subsystem.

Outcome. Without this architecture, the Tier-2 throw in a monolithic boot would abort the process — no health endpoint, no operable kernel, blind operators. With it: Tier-0 reached verified-operable *before* Tier 2 was even attempted; the AI failure was *isolated*; non-AI domains (which do not *require* the AI runtime — they may declare it `optional`) booted normally; and the health rollup names `ai-runtime: down` while the platform serves at `degraded`.

---

11. Security, Safety & Failure Modes

11.1 Fail-open vs fail-closed posture

The boot mechanism itself is **fail-closed on required dependencies** (refuse to boot rather than run half-formed) and **fail-open on optional dependencies and later tiers** (isolate and degrade rather than crash). Whether a *feature* fails open or closed when its AI tier is degraded is a policy decision delegated to the access gate / router (out of scope here): a governance/judge route may fail closed (no silent downgrade) while a general route fails open. The *boot* contract guarantees the kernel survives so that those policies can be evaluated at all.

11.2 Failure-mode table

Failure mode	Tier	Detection	Containment	Residual risk
Circular `requires`	sort	cycle detector names a module	abort with diagnostic before side effects	mis-declared deps must be fixed in code
Missing required dependency	validate	requirement validator lists all	refuse boot (fail-fast)	none at runtime — caught pre-bootstrap
Kernel module throws	0	register/bootstrap throw	fatal abort (kernel is unrecoverable)	platform cannot run without a kernel — by design
Required OS service throws (DB)	1	bootstrap throw	fatal by default; configurable	running without persistence is unsafe
AI runtime throws	2	bootstrap throw / health `down`	**isolate**; kernel stays up	AI features unavailable until repaired
One domain throws	3	bootstrap throw / health `down`	**isolate**; other domains unaffected	that domain unavailable; data intact
Health check itself throws	any	aggregator try/catch	module marked `down`; rollup `down`	noisy but observable
Slow/stuck shutdown of a domain	any	reverse-order shutdown	per-module error tolerated, logged	drain may take longer

11.3 STRIDE-style notes (boot mechanism scope)

STRIDE	Consideration for the assembler
Spoofing	Module ids are trusted from the registry; a hostile module is out of scope (supply-chain controls apply upstream).
Tampering	Identity-bound schema isolation limits a compromised domain to its own schema.
Repudiation	`BOOT_RUN` + `MODULE_LOAD` + `HEALTH_SAMPLE` give an auditable boot ledger.
Information disclosure	Health rollups expose status, not secrets; per-module `health()` must not leak credentials.
Denial of service	Later-tier isolation prevents a single broken module from denying the whole platform.
Elevation of privilege	A required dependency may never point "upward" to a higher tier; domains cannot inject themselves below the kernel.

---

12. Standards & Framework Mapping

We claim **semantic alignment**, *not* certified compliance, with the following.

Framework / standard	Relevant clause	How this aligns
ISO/IEC 25010 (software quality)	Reliability → *Fault tolerance, Recoverability*; Maintainability → *Modularity*	Later-tier isolation = fault tolerance; uniform contract = modularity; reverse-order shutdown = recoverability.
NIST SP 800-160 Vol. 1 (systems security engineering)	*Loss containment, graceful degradation*	Tier isolation contains loss; the platform degrades (kernel up, AI down) rather than failing catastrophically.
Kubernetes liveness/readiness	readiness ≠ liveness	Tier-0 verified-operable maps to *liveness*; full-tier readiness maps to *readiness*; the kernel can be alive while not yet AI-ready.
The Twelve-Factor App	IX. *Disposability* (fast startup / graceful shutdown)	Deterministic boot + reverse-order graceful shutdown.
OSGi service lifecycle	declarative dependency wiring	Uniform `requires/optional` contract is a declarative-dependency model.
OpenTelemetry health/semantic-conventions	service health signals	The aggregated rollup is a natural source for `up/degraded` health signals.

No certification, conformance mark, or audit is asserted. The mapping is provided to help architects situate the technique, not to claim regulatory approval.

---

13. Evaluation Methodology

We describe *how* one would evaluate an implementation. Any numeric values below are **illustrative** placeholders to show the interpretation, not measured results.

Dimension	Metric	How measured	Interpretation
Operability-before-AI	`tier0_ready_ms`	Wall-clock from boot start to Tier-0 health answering	Lower is better; should be ≪ full-boot time. *(illustrative: ~120 ms vs ~6 s full boot)*
Fault isolation	Kernel-survival rate under injected later-tier faults	Inject N Tier-2/Tier-3 failures; count boots where Tier-0 stays operable	Target 100% for non-required tiers.
Fail-fast precision	Unmet-deps reported per failing boot	Count distinct missing deps surfaced in one validation pass	Higher = fewer fix-rebuild cycles. *(illustrative: all-at-once)*
Determinism	Load-order stability	Boot K times with same module set; diff orders	Should be identical across runs.
Shutdown safety	Modules cleanly drained on shutdown	Count modules whose `shutdown` completed without error	Higher is better; one stuck module must not block others.
Health fidelity	Rollup correctness	Compare rollup vs ground-truth injected statuses	Worst-case monotone rollup must match.

Methodology note. The reference `src/demo.js` self-check is the minimal evaluation harness: it asserts the Tier-0-before-AI and isolation invariants and exits non-zero on violation. A fuller benchmark would script fault injection across a realistic module graph and record a `BOOT_RUN` ledger (§8) for each trial.

14. Novelty & Inventive Claims

The following claims are stated in **prose** to delineate the disclosed combination. They are published as prior art; **no patent is sought.** Claim 1 is independent; Claims 2–15 are dependent, each narrowing on a distinct novel feature.

Claim 1 (independent). A method of initializing a multi-module AI-agent platform, comprising: (a) providing a plurality of software modules each exporting an *identical* lifecycle contract declaring required and optional capability dependencies and a health-reporting function; (b) at boot, computing a topological ordering of the modules from said declared required dependencies and refusing to proceed past boot (fail-fast) when any declared required dependency is unmet; (c) loading the modules in ordered, numbered capability tiers such that a first tier instantiates a kernel that exposes an aggregated platform-health endpoint and is fully operable and manageable while no AI-inference runtime module and no business-domain module is loaded; (d) loading the AI-inference runtime as a distinct subsequent tier gated above the operability of the first tier and below an optional business-domain tier; whereby the platform attains and reports a verified operable health state independent of, and prior to, loading any AI or domain logic, and whereby a single later-tier module failing its declared dependency is isolated without preventing the lower tiers from remaining operable and health-reporting.

Claim 2. The method of claim 1, wherein the identical lifecycle contract comprises the fields `{id, category, requires, optional, register, bootstrap, shutdown, health, emits, listens, config}` and is implemented identically by operating-system-service modules, the AI-inference-runtime module, and business-domain modules.

Claim 3. The method of claim 1, wherein computing the topological ordering uses only the *required* dependencies as ordering edges, and wherein optional dependencies are wired when present and omitted when absent without affecting the boot order or causing a fail-fast refusal.

Claim 4. The method of claim 1, wherein the fail-fast refusal is performed after computing the topological ordering but before invoking any module's `register` or `bootstrap` phase, such that no side effects are performed on the path to an unmet- dependency failure.

Claim 5. The method of claim 4, wherein the fail-fast refusal enumerates *every* unmet required dependency across all modules in a single validation pass.

Claim 6. The method of claim 1, wherein detecting a cycle in the dependency graph aborts the boot with a diagnostic naming a specific module involved in the cycle.

Claim 7. The method of claim 1, wherein the first (kernel) tier comprises at least a configuration module, an event-bus module, a scheduler module, and the module that exposes the aggregated platform-health endpoint, and excludes any AI-inference-runtime module and any business-domain module.

Claim 8. The method of claim 1, wherein loading proceeds tier-by-tier in ascending tier number, and a verified-operable barrier is recorded once the first tier is registered, bootstrapped, and answering the health endpoint, prior to loading any higher tier.

Claim 9. The method of claim 1, wherein the AI-inference-runtime tier is loaded strictly after an operating-system/policy-services tier and strictly before an optional business-domain tier, and wherein a failure of the AI-inference-runtime tier is isolated such that the operating-system/policy-services tier and the kernel remain operable and health-reporting.

Claim 10. The method of claim 1, wherein a business-domain module that fails its declared dependency or reports a down health status is quarantined without affecting the operability of other business-domain modules or of lower tiers.

Claim 11. The method of claim 1, wherein each business-domain module is bound to a distinct persistent-storage namespace named for, and isolated to, that module's identifier, such that domain data isolation tracks module identity.

Claim 12. The method of claim 1, wherein the aggregated platform-health endpoint produces a worst-case monotone rollup over per-module health statuses, mapping any down status to a platform-down status and any degraded status, absent a down status, to a platform-degraded status.

Claim 13. The method of claim 1, further comprising shutting down the loaded modules in reverse of their boot order by invoking each module's `shutdown` function, and tolerating a per-module shutdown error without blocking shutdown of the remaining modules.

Claim 14. The method of claim 1, wherein the set of modules to load, the category-to-tier mapping, and the fail-fast posture for optional dependencies are supplied by external configuration rather than hardcoded, such that the same boot method initializes platforms of differing module composition.

Claim 15. The method of claim 1, wherein higher-tier modules are imported lazily only when their tier is reached, such that an import-time error in the AI-inference- runtime tier or a business-domain tier cannot prevent the kernel tier from attaining its verified-operable state.

---

15. Limitations & Threats to Validity

- **Ingredients are individually old.** Topological dependency boot (OSGi, ABP), staged init (systemd, US 5,379,431), aggregated health (Actuator), and microkernel fault isolation (Mach, OTP) are each well-trodden. The disclosure's contribution is the *combination and the AI-platform-specific tier semantics*, which §5 isolates. This is also why a patent is *not* sought (see §14 and Appendix D).
- **Obviousness is close.** A skilled engineer building an LLM platform might predictably tier by dependency/blast-radius and place a heavy optional AI runtime above the kernel. We acknowledge this directly; the publication's purpose is precisely to ensure the *specific* combination remains free to practice.
- **Policy is out of scope.** Whether a degraded AI tier causes a given feature to fail open or closed is delegated to a separate routing/governance layer; this document only guarantees the kernel survives to evaluate such policy.
- **Single-process framing.** The mechanism is described for an in-process module graph. A distributed variant (tiers as separate pods) is sketched in §16 but not claimed here.
- **Illustrative numbers.** Every quantitative figure in §13 is illustrative, not measured; an implementer must benchmark their own graph.

- **Intentional withholding.** Empirically-tuned operational specifics (exact module rosters, per-environment tier maps) are deliberately *not* the subject of this publication; the disclosure covers the *mechanism*, which is what must remain free.

---

16. Future Work & Open-Source Reference App

The planned open-source reference app (see docs/OPEN-SOURCE-APP.md) will:

- Boot a synthetic 12–20-module platform through the assembler and render a live **tier-health dashboard** that visibly turns Tier-0 green *before* the AI tier loads.
- Provide a **fault-injection panel** that crashes a chosen Tier-2/Tier-3 module and shows the kernel staying up.
- Ship a generic **Kubernetes/Helm** deployment (no internal cluster identifiers) so the demo runs on any AKS or vanilla cluster.

Roadmap. (1) Single-process reference + dashboard. (2) Fault-injection + boot-run ledger persistence. (3) A *distributed* variant where each tier is a separate Deployment, the kernel's readiness gates the AI Deployment, and the health aggregator scrapes per-tier `/health` — extending the in-process guarantee to a pod topology.

---

17. Conclusion

A modular AI-agent platform should reach a *verified-operable* state before it loads the expensive, fragile thing that makes it intelligent. By applying one uniform lifecycle contract across every module, computing a topological fail-fast boot, and loading numbered capability tiers in which the AI runtime is its own isolatable tier above the OS and below the domains, the platform guarantees that the kernel and its health endpoint come up first and survive a later-tier failure. We publish this combination as prior art so that the technique — useful to anyone building a large agent platform — remains free for all to practice and cannot be enclosed by a later patent.

Appendix A — Prior-Art Landscape (well-trodden vs candidate-novel)

Well-trodden (NOT claimed as novel):

- Topological dependency ordering at boot (OSGi DS, Eclipse Equinox, ABP `[DependsOn]`, Spring auto-config ordering).
- Aggregated component health endpoints (Spring Actuator `/health`).
- Staged/numbered init (SysV runlevels, systemd targets, US 5,379,431 staged IPL).
- Microkernel-minimal-core + service fault isolation (Mach, L4, MINIX, Erlang/OTP).
- Liveness-vs-readiness separation (Kubernetes).

Candidate-novel as a combination (the subject of this disclosure):

- One *identical* lifecycle contract spanning OS + AI-runtime + business domains.

- A topological *fail-fast* boot that refuses partial startup, surfacing *all* unmet deps before side effects.
- An explicit **Tier-0-operable-and-health-serving-before-any-AI** guarantee.
- The **AI-inference runtime modeled as its own distinct tier** between OS/policy and optional business domains, with later-tier fault isolation that keeps the kernel up.
- Identity-bound per-domain schema isolation tracking module boundaries.

Honesty attestation. The prior-art search supporting this document is *directional, not exhaustive*. It reflects a good-faith review of widely known modular-systems, init-system, and AI-platform literature as of 2026-06-25. No claim is made that it surveyed every patent, product, or paper. The combined technique is published precisely because the individual ingredients are old and close art exists; placing the combination on the public record is the appropriate, honest response.

Appendix B — Glossary

Term	Definition
Assembler	The component that discovers, sorts, validates, and lifecycle-drives all modules at boot.
Lifecycle contract	The uniform `{id, category, requires, optional, register, bootstrap, shutdown, health, emits, listens, config}` shape every module exports.
Capability tier	A numbered band (0–3) of modules brought up together; lower tiers are operable without higher tiers.
Tier 0 / bare kernel	Config, registry, bus, scheduler, health endpoint — operable with no AI and no domains.
AI-inference runtime (Tier 2)	Token/LLM routing, tool execution, persona orchestration — its own distinct, isolatable tier.
Fail-fast	Refusing to boot when a required dependency is unmet, before any side effects.
Verified-operable barrier	The recorded moment Tier 0 is registered, bootstrapped, and answering health — before any higher tier loads.
Isolation / quarantine	Containing a failed later-tier module so lower tiers stay operable.
Identity-bound schema	A domain module's Postgres schema named for and isolated to its module id.
Health rollup	The worst-case monotone aggregation of per-module health into a single platform status.

Appendix C — Reference-Implementation Index

File	Purpose	Demonstrates
`src/assembler.js`	The assembler: registry, topo-sort, fail-fast validation, tier-ordered load, isolation, health aggregation, reverse shutdown.	R1–R8, R10, Claims 1–15
`src/modules.js`	A synthetic 12-module, 4-tier platform incl. a broken Tier-2 AI module and a broken Tier-3 domain.	R4, R5, R6, R9
`src/demo.js`	Executable demo + self-check; boots, asserts Tier-0-before-AI, injects faults, prints rollup, exits non-zero on invariant violation.	R3, R4, R5, R6, R7
`src/README.md`	What the sample shows, how to run it, clean-room disclaimer.	—

Appendix D — Defensive-Publication Deposit & Timestamp

- **Publication date:** 2026-06-25.
- **Publisher / copyright holder:** Gus IT LLC (Florida, USA).
- **Author:** Gustavo Assuncao, PhD.
- **Deposit channel:** to be assigned (IP.com / Zenodo / arXiv) — establishes a public, dated, citable prior-art record. No DOI is asserted at the time of writing.

- **Intent:** *This disclosure is intentionally public to bar later patenting of the described combination by others and to keep the technique freely practiceable*. The publication, together with the AGPL-3.0-or-later license (express patent grant) and the committed source history, constitutes a dated, public record of the combined technique as prior art as of the publication date.