AegisAI v1.0 — A Sanctioned Pathway for AI Agents Accessing Enterprise Data

Compatible with SAP API Policy 4.2026a, Section 2.2.2

AegisAI

May 2026

Contents

Executive Summary

In April 2026, SAP published version 4.2026a of its API Policy.[^1] **Section 2.2.2** prohibits API use for "interaction or integration with (semi-) autonomous or generative AI systems that plan, select, or execute sequences of API calls" — except through SAP-endorsed architectures, data services, or service-specific pathways. That definition of an agent captures essentially every AI tool currently being deployed against enterprise data.

> **AegisAI uses no LLMs, no ML models, and no probabilistic classifiers in the request path.** Every deny decision is reproducible from inputs. The "AI" in the name refers to the AI agents calling AegisAI from the *other* side of the gate; AegisAI itself is deterministic Python. See §2.1.

For the SAP-customer side of the market, this is not a future problem. Microsoft Copilot, Joule, internal RAG pipelines, and the long tail of AI agents accessing SAP data must now route through a sanctioned interface or stop. There is no third option.

AegisAI is that interface. It sits between AI agents and SAP (and AWS, Azure, GCP) data systems, applying nine deterministic stages to every request: rate-limiting, identity-propagated authentication via documented BAPIs, adaptive trust evaluation, deterministic policy, parameterised query planning, schema-driven response masking, and a tamper-evident HMAC audit chain.

This whitepaper covers, in order:

- **§1.** The policy moment and what 4.2026a actually says.
- **§2.** Design invariants AegisAI was built against.
- **§3.** The nine-stage request pipeline, stage by stage.
- **§4.** Threat model: what we mitigate and how.
- **§5.** Mapping AegisAI to the policy's four preamble goals.
- **§6.** Pilot path and commercial deployment models.
- **§7.** Honest residuals and detection limits — the 6% we haven't delivered, and the patterns AegisAI does not catch.
- **§9.** References.

A reader interested only in the architecture can start at §3. A reader doing diligence should read end to end.

1. The Policy Moment

1.1 What SAP API Policy 4.2026a actually requires

The load-bearing sentence is **Section 2.2.2**:[^1]

> "Except through and within the limits of SAP-endorsed architectures, data services, or service-specific pathways expressly identified and intended for such purposes, SAP prohibits API use for: (a) interaction or integration with (semi-) autonomous or generative AI systems that plan, select, or execute sequences of API calls, and (b) scraping, harvesting, or systematic and/or large-scale data extraction or replication."

Two things to notice. First, the *prohibition* is on direct AI agent access and on bulk extraction. Second, the *carve-out* is broad: a "SAP-endorsed architecture" is one of three valid shapes, alongside SAP-built data services and SAP-built service-specific pathways. AegisAI fits the **architecture** leg — a sanctioned control plane that any agent can use without SAP having to ship per-vendor pathways.

The policy also defines an "agent" with unusual precision — *"(semi-)autonomous or generative AI systems that plan, select, or execute sequences of API calls"*. That definition is wide enough to cover Microsoft Copilot, SAP Joule, custom RAG pipelines, internal LLM-driven automations, and most production AI tooling against SAP today. If your AI plans, selects, or executes sequential calls, it is an agent under SAP's own framing — and Section 2.2.2 applies.

1.2 The policy's four preamble goals

The policy preamble names four things its controls are designed to do:[^1]

> "…controls designed to **safeguard solution health and security**, **promote equitable access**, **prevent API misuse**, and **support the enforcement of this API Policy**."

All four are tractable with a properly-architected control plane. AegisAI was designed against these specific concerns — not around them. §5 maps each to a shipping capability.

1.3 Why a sanctioned layer is the only viable answer

Three deployment patterns are possible for AI-on-SAP under 4.2026a:

- **Inside the AI app.** Each Joule / Copilot / internal tool ships its own SAP integration. Every vendor reinvents the authorisation flow; audit format differs across tools; SAP's policy is still violated by the weakest implementation.
- **Inside SAP itself.** SAP ships a built-in AI gateway. Tied to SAP release cycles (years); AI-vendor-specific; doesn't help with non-SAP data systems (RDS, Synapse, BigQuery).
- **As a separate sanctioned layer.** One control plane, multiple AI tools, multiple data systems, customer-owned audit log, AI-vendor-neutral. AegisAI is in this category.

The third pattern is the one 4.2026a tacitly endorses by allowing "SAP-endorsed architectures" without insisting they be SAP-built.

1.4 AegisAI is a sanctioned pathway, not a bypass

A careful reader will go to **Section 3** of the policy next:

> "Customers, partners, and third parties must not bypass, disable, or otherwise circumvent API Controls, including through intermediary services, custom code or developments, proxies, gateways, impersonation techniques, or similar mechanisms."[^1]

This is the right question to ask of any control plane — and AegisAI must answer it directly, because at first glance an "intermediary gateway" is exactly what AegisAI is.

The distinction is whether the layer **strengthens** or **weakens** the controls SAP's APIs already enforce. A bypass disables limits, masks identity, or removes the audit trail. A sanctioned pathway does the opposite. AegisAI:

1. **Adds rate limits SAP cannot enforce on aggregate AI traffic per user.** Per-user and per-tenant fixed-window counters in Redis (§3.1) are tighter than direct API access; an agent that would otherwise hammer SAP at machine speed is now bounded.
2. **Adds request ceilings SAP does not impose at the HTTP layer.** Body / URL / wall-time caps (§3.2) refuse oversize and slow-loris traffic *before* it reaches SAP.
3. **Propagates identity rather than masking it.** SAP enforces authority on the same `sub` AegisAI verified (§3.4); the calling user's SAP credentials flow with the request — no service-account substitution.
4. **Calls only documented BAPIs.** `BAPI_USER_GET_DETAIL`, `SUSR_GET_PROFILE_AUTH_OBJECTS`, plus customer Z-RFCs through an explicit extension point. No undocumented, internal, or reserved-namespace endpoints (§3.4 / §3.8).
5. **Records every decision in a tamper-evident audit chain SAP can re-walk.** Postgres-backed HMAC chain with public integrity probe (§3.10). The audit *exceeds* what direct API access leaves behind.
6. **Refuses to serve when its own dependencies are unhealthy.** Redis / audit DB / JWKS down → fail-closed (§2.4 / §3.10). A bypass would degrade silently; AegisAI returns 503.

If a hostile reviewer asks "is this an intermediary gateway?", the honest answer is yes — and the immediate follow-up is "and here are the six places it tightens, not loosens, the controls the policy is concerned with." Section 3 reads as a prohibition on circumvention by gateway, not on enforcement by gateway.

2. Design Invariants

AegisAI was built against four invariants. Every architectural decision is downstream of these.

2.1 Deterministic

Same identity + same intent + same context + same data = same response. Every time. No probabilistic security decisions; no black-box LLM in the policy path. The policy engine evaluates a deny-by-default expression set against a fixed AST whitelist. The query planner emits parameterised SQL with `:named` placeholders, never string interpolation. The response firewall applies a configured `MaskStrategy` based on field tags, not heuristic detection.

This isn't a stylistic preference. Probabilistic guardrails ("do not return PII" prompts, output keyword blocklists) are a hope, not a control. SAP's stability and governance goals require deterministic enforcement. We deliver it.

2.2 Defense in depth

Seven independent layers can each deny: rate-limit middleware, request-ceiling middleware, JWT verification, SAP/cloud authority lookup, adaptive trust evaluation, policy expression set, query planner subset check. A single bypass cannot leak data on its own. No layer trusts the prior one to have caught everything.

2.3 Identity-propagating

The JWT's `sub` and `tenant_id` flow all the way into the SAP RFC call. SAP enforces authority on the same identity AegisAI verified, not on a generic service account. Auditors get a chain of identity from end-user to data row.

2.4 Fail-closed by default

Infrastructure outage (Redis, audit DB, JWKS endpoint) denies before it serves. Explicit environment opt-in can toggle degraded-mode for non-production tiers; PRODUCTION ignores those opt-ins.

3. The Nine-Stage Request Pipeline

A `POST /query` traverses nine deterministic stages. Any one of them can deny the request.

```
POST /query
     |
     v
[Stage 1]  Rate-limit middleware          (Redis-backed)
[Stage 2]  Request-ceiling middleware     (body / URL / timeout)
     |
     v
[Stage 3]  Authenticate                   (JWT; HS/RS/ES/PS + JWKS)
[Stage 4]  Authorise                      (SAP BAPI walk; cloud IAM)
[Stage 5]  Adaptive trust                 (Redis-backed signals)
[Stage 6]  Policy                         (safe-AST expression set)
[Stage 7]  Plan                           (parameterised SafeQuery)
[Stage 8]  Execute                        (MODE-gated backend)
[Stage 9]  Mask                           (schema-driven firewall)
     |
     v
[Audit]    Tamper-evident HMAC chain      (Postgres + row lock)
     |
     v
{ decision, data, trust_*, trace_id, trace }
```

3.1 Stage 1 — Rate-limit middleware

Per-user and per-tenant fixed-window counters in Redis. Returns HTTP 429 with an accurate `Retry-After` header (seconds until the current window rolls). Fails open on Redis hiccup; the trust system in stage 5 is the real gate.

3.2 Stage 2 — Request-ceiling middleware

ASGI-level enforcement of:

- Body size <= `AEGIS_MAX_BODY_BYTES` (default 64 KB) → HTTP 413
- URL length <= `AEGIS_MAX_PATH_BYTES` (default 2 KB) → HTTP 414
- Per-request wall time <= `AEGIS_REQUEST_TIMEOUT_S` (default 10 s) → HTTP 504

Both `Content-Length` and streaming `http.request` body counts are checked, so a missing or lying `Content-Length` doesn't bypass the cap.

3.3 Stage 3 — Authenticate

JWT verification supporting HS256, RS256, ES256, and PS256. JWKS rotation cached process-wide via `PyJWKClient`. `iss`, `aud`, `exp`, `nbf`, signature, and configurable clock-skew leeway all enforced. Admin-claim resolution accepts list, comma-separated, and space-separated encodings (Okta, Azure AD, PingFederate flows).

3.4 Stage 4 — Authorise

The SAP authority check walks `BAPI_USER_GET_DETAIL` to fetch ACTIVITYGROUPS and PROFILES, then queries `SUSR_GET_PROFILE_AUTH_OBJECTS` for each profile to retrieve the auth-object → field-value tuple tree. Field-value enforcement (e.g. `BUKRS = 1000`) honours wildcards. No service-account masking; the calling user's SAP credentials flow with the request.

For cross-cloud requests, Stage 4 also runs:

- AWS: `iam:SimulatePrincipalPolicy` against the principal ARN resolved via STS `GetCallerIdentity`.
- Azure: `AuthorizationManagementClient.role_assignments.list_for_scope`.
- GCP: `bigquery.Client.test_iam_permissions`.

A connector that errors fail-closes. A connector that's not configured is "not in scope" in LOCAL mode and a hard deny in PRODUCTION.

3.5 Stage 5 — Adaptive trust

Tenant-scoped Redis ledger. Signals collected per user:

- **Frequency** over a configurable window (default 1 hour).
- **Scope expansion** — new scope values vs recent.
- **Coverage ratio** — fraction of authorised scope touched.
- **Revisit ratio** — repeat-access patterns.
- **Coordination** — arithmetic-progression detection across users (catches multi-user exfiltration).
- **Global coverage growth rate** — distinct new scope values per second across all users in the tenant.

Score combines these into `0.0..1.0`. Trust levels: HIGH >= 0.8, MEDIUM >= 0.4, otherwise LOW. Rate limits and request restrictions adjust per level.

This is the stage that addresses the policy's *"sequences of API calls"* concern directly: AegisAI watches the sequence shape, not just the individual call.

3.6 Stage 6 — Policy

Priority-weighted expression set evaluated by a safe AST whitelist. Only `len, any, all, min, max, abs, str, int, float, bool` callables are allowed. Lambdas, comprehensions, imports, and dunder attribute access are rejected at compile time. Conflict resolution: highest priority wins; deny wins on tie; default deny if nothing matches.

3.7 Stage 7 — Plan

Intent compiler validates and structures the request. Query planner emits a `SafeQuery` dict:

```
{
```

```
  sql:                "SELECT id, customer, amount FROM sales
                       WHERE tenant_id = :tenant_id
                         AND bukrs IN (:bukrs_0)",
  params:             {tenant_id: "tenant_a", bukrs_0: "1000"},
  entity:             "sales",
  fields:             ["id", "customer", "amount", ...],
  tenant_id:          "tenant_a",
  row_policy_names:   ["tenant_isolation"]
}
```

User-derived values appear only as `:named` placeholders. Tenant isolation is rendered as a row policy on every entity.

3.8 Stage 8 — Execute

MODE-gated dispatch. `MODE=PRODUCTION` requires the real backend. The SAP simulation refuses to run in PRODUCTION; the in-memory fixture dataset refuses to run in PRODUCTION. Backend selection follows the request's `context.system`:

- `sap` → `SAPAdapter` (pyrfc + OData) under the user's identity.
- `aws` → RDS Data API or Redshift Data API.
- `azure` → Synapse SQL endpoint.
- `gcp` → BigQuery client API.

3.9 Stage 9 — Mask

The response firewall walks every row, looks up each field's `FieldTag`, and applies the configured `MaskStrategy`: `NONE | REDACT | HASH | PARTIAL | AGGREGATE | DROP`. Untagged fields that contain PII-shaped content (SSN, email, credit card) fail the request — that's a schema-drift / leaky-connector signal worth surfacing.

If every field of a row was redacted, the firewall fails closed rather than returning `{}`.

3.10 Audit append

Every decision lands in a Postgres-backed table. Per-row computation:

```
canonical = canonical_json(payload)            # sorted keys, compact
prev      = previous row's row_hash            # via SELECT ... FOR UPDATE
row_hash  = sha256(prev || canonical)
hmac_sig  = HMAC-SHA256(AEGIS_AUDIT_HMAC_KEY, row_hash)
```

A row-level lock on the previous row prevents chain races under concurrent writes. The HMAC prevents an attacker who writes directly to the database from extending the chain undetected — they'd need the key, which lives in your KMS or Vault.

`GET /api/audit/verify` (admin-gated) walks the full chain end to end and returns the first break id if any. `GET /api/audit/integrity` (unauthenticated, low-information) is the same check intended for k8s probes.

4. Threat Model

Threat	Mitigation
Forged JWT	iss / aud / exp / nbf / signature verification; JWKS rotation
Token replay after expiry	`exp` + short leeway + clock sync
Cross-tenant read	`tenant_id` in JWT + row policy on every query
SQL injection via intent	Parameterised SafeQuery; no string interpolation
Scope expansion via broad intent	Planner subset check + LOW-trust aggregation deny
Field-level PII leak	ResponseFirewall mask from FieldTag
Inference via repeated queries	Trust coverage ratio + revisit + coordination
Coordinated cross-user attack	Global coverage-growth + density signals
Audit tamper	HMAC hash chain + verify endpoint
Trust ledger DoS (Redis down)	Fail-closed by default
Oversize / slow-loris	Ceiling middleware (body / path / timeout)
Request-rate abuse	Rate limit middleware (per user + per tenant)
Unauthenticated admin actions	`require_admin` dependency
Config drift between subsystems	Single `DataSchema` registry
Default-secret deployment	Startup-time refusal in PRODUCTION when readiness blockers exist

Not yet mitigated and explicitly on the roadmap: cross-region encryption-at-rest key management, formal third-party pen-test, DDoS upstream of the gateway, and an automatic kill-switch on chain-break.

5. Mapping to the Policy's Four Preamble Goals

The policy preamble names four things its controls exist to do.[^1] Each maps to a specific shipping capability in AegisAI v1.0.

Goal 1 — Safeguard solution health and security

- JWT verification (HS256 / RS256 / ES256 / PS256) with JWKS rotation; mandatory `iss`, `aud`, `exp`, `nbf`.
- Identity-propagated SAP authority via documented BAPIs (no service-account masking).
- Schema-driven response firewall: per-field classification (PUBLIC / INTERNAL / CONFIDENTIAL / RESTRICTED) × user clearance × auth-object gate.
- Fail-closed defaults at every layer (Redis / audit DB / JWKS outage denies before it serves).
- PRODUCTION refuses to start with default secrets.

Goal 2 — Promote equitable access

- Per-user and per-tenant fixed-window rate limits in Redis (§3.1) — one tenant cannot monopolise SAP capacity at the expense of another.
- Trust-based throttling drops misbehaving callers in proportion to their score, leaving capacity for well-behaved ones.
- ASGI-level request ceilings (§3.2) prevent oversize / slow- loris requests from starving the gateway and downstream SAP.
- One well-behaved client to SAP, not thousands of agents — the load shape SAP sees is bounded and predictable.

Goal 3 — Prevent API misuse

- Intent compiler validates and structures every request; vague, scope-expanded, or syntactically suspicious intents are rejected at compile time before any backend is touched.
- Deterministic policy engine evaluates a deny-by-default expression set against a fixed AST whitelist — no probabilistic guardrails to second-guess.
- Adaptive trust signals (§3.5) — frequency, scope expansion, coverage growth, cross-user coordination — directly address the policy's *"sequences of API calls"* concern.
- Documented BAPIs only; no undocumented or reserved-namespace interfaces; no ODP large-scale extraction patterns.
- Parameterised `SafeQuery` planner — no string interpolation, no SQL injection surface, even on the LOCAL fixture path.

Goal 4 — Support the enforcement of this API Policy

- Tamper-evident HMAC audit chain (§3.10) — Postgres-backed with `SELECT ... FOR UPDATE` row lock so the chain stays linear under multi-pod writes.
- Public integrity probe (‘/api/