ChatGPT

Mission Alignment

The current architecture **strongly supports the original mission** of a cross-domain, cross-jurisdiction regulatory compliance graph. The design is **industry-agnostic and jurisdiction-neutral** – it defines generic node types (e.g. `:Obligation`, `:Threshold`, `:Rate`, `:Timeline`) that can model rules in any regulatory domain [1]. Jurisdictions are first-class entities (`:Jurisdiction`, with hierarchies like `:Region` and treaty-based `:Agreement` nodes) to capture cross-border relationships [2]. Notably, special cases like Northern Ireland, Common Travel Area, Isle of Man, etc., are explicitly modeled rather than hard-coded, ensuring **multi-jurisdiction support**. For example, NI is a `:Region` under the UK with EU-linked regimes, CTA is an `:Agreement` node, and cross-border links (e.g. `COORDINATED_WITH`, `TREATY_LINKED_TO`, `EQUIVALENT_TO` edges) connect rules across jurisdictions [3]. This confirms the graph can represent *any* country or regulatory regime without assuming a single "primary" jurisdiction [4] [5].

Crucially, the architecture is **privacy-first** to enable multi-tenant use. All regulatory knowledge resides in a shared Memgraph **rules graph** that stores only public rule data – *never* any tenant or user PII [6]. Personal or tenant-specific information (user IDs, tenant IDs, scenarios, etc.) is strictly kept out of the graph [7]. Instead, such data lives in a separate application database (Supabase) with row-level security per tenant [8] [9]. This guarantees the graph remains a **shared, multi-tenant knowledge base** freely queryable by all tenants without privacy leakage [10]. The **Graph Ingress Guard** enforces these boundaries at write-time – any attempt to add a node/edge with disallowed properties (e.g. a user identifier) is blocked [11] [12]. Likewise, an **Egress Guard** sanitizes outputs (LLM calls, tool requests) to strip PII before leaving the system [9]. These safeguards ensure that one tenant's data or context never contaminates the global regulatory graph, aligning with the mission of a **multi-tenant PaaS/SaaS** platform.

Overall, the core platform concept – a **chat-first research assistant underpinned by a shared rules graph** – is intact [13] [14]. The graph-centric approach (with timeline reasoning and LLM agents on top) is not tied to any single industry. In practice, the current content focuses on Irish/EU tax and welfare, but that is a content scope choice, **"not an architecture limitation"** [15]. The schema can just as easily accommodate new industries (e.g. financial compliance, environmental regulations) by defining new node labels or relationships as needed, without changes to fundamental logic [16]. In summary, the architecture's **mission alignment is strong**: it was explicitly designed to handle any regulatory domain and jurisdiction, with a **shared global graph** that gets richer over time (via ingestion and SKOS-like concept capture) and a strict separation of tenant-specific data [17] [9].

Architecture Soundness

The system's architecture is **sound and well-layered**, providing a solid foundation for a scalable multi-tenant compliance platform. At a high level it is split into clear layers – a web UI, a thin API layer, a domain-agnostic **Compliance Engine**, and two data stores: **Memgraph for shared rules** and **Supabase for tenant-specific data** [18]. This separation of concerns yields several key strengths:

- **Privacy & Security by Design:** The Memgraph graph holds only public regulatory knowledge, whereas all user/tenant-specific content (conversations, user profiles, etc.) resides in Supabase under strict row-level security [8] [9]. The Graph Ingress Guard and Egress Guard serve as gatekeepers on these boundaries, *preventing any PII or tenant data from entering the graph* and scrubbing outputs before calling external APIs [9]. This enforces a hard multi-tenant boundary and protects sensitive data across the board.

- **Temporal and Cross-Domain Reasoning:** The inclusion of a **Timeline Engine** and time-aware schema elements means the platform natively handles temporal rules (lookback periods, deadlines, lock-in effects) for any domain [19]. This engine operates generically on `:Timeline` nodes/edges, so whether it's tax deadlines or compliance certification expiries, the logic is the same. The graph design and timeline engine together enable *dynamic scenario evaluation over time*, a critical capability for compliance across domains.

- **Modularity and Provider-Agnostic AI Layer:** The "engine" layer uses an LLM router abstraction that is provider-agnostic [20]. This allows per-tenant AI configurations (different LLM providers or models per tenant policy) without altering core logic [21]. Prompt assembly is done with **aspect modules** (jurisdiction aspect, persona/profile aspect, disclaimer aspect, etc.), making it easy to extend prompts when new domains or features are introduced [22]. There is no hardcoded business logic per domain in the code – new rules or jurisdictions are introduced as data and prompt tweaks, not code changes. This architecture choice keeps the platform **flexible** as it scales to new industries.

- **Robust Graph Operations & Observability:** All writes to the graph go through a centralized service (`GraphWriteService`) that applies validation and auditing (via ingress aspects) [23]. Agents and tools only perform read queries, never writes, enforcing consistency and auditability [24]. The system is prepared for **change detection** and real-time updates – there's a patch-based graph change streaming design so that when rules are added/updated, clients can get incremental updates [25]. (For full production scale, the simple in-memory event hub will need to be replaced with a pub/sub mechanism like Redis to handle multiple servers [26], but this is a known scaling task.) Furthermore, an **authorization envelope** (with constructs for shareable content and fine-grained access control) is defined for future use with systems like OpenFGA [27], indicating forethought around enterprise access control needs.

In short, the **core architecture is solid**. It cleanly separates concerns (UI, API, engine, graph, tenant DB), uses **modern frameworks** (Next.js 16, Node 24 LTS, etc.), and has built-in mechanisms for privacy, extensibility, and temporal reasoning. The design received high marks in an internal review for mission alignment and architecture soundness [28]. Most weaknesses lie not in the design itself, but in **ancillary features not yet fully implemented** for production readiness. For example, multi-tenant *infrastructure* like rate limiting, billing integration, and feature gating are still to be completed [26] [29]. These are roadmap items rather than structural flaws. Overall, the architecture provides a **strong, scalable backbone** for the intended SaaS platform, needing only incremental hardening and feature-complete implementations to be launch-ready.

Extensibility Evaluation (Current vs. Alternatives)

Current Approach – "Graph as the Extension Point": Today, extending the system to a new regulatory domain or jurisdiction is largely a **data operation**, not a code change. To add a new domain, one would *seed new nodes and edges* into the shared Memgraph (e.g. via Cypher seed files for the new rules) and update the ingress guard's whitelist to include any new node/relationship types [30]. Optionally, one might register a new specialist agent or add prompt aspects tailored to that domain, but the core platform doesn't require a new service or plugin – it will automatically pick up the new knowledge. Thanks to concept extraction, even user conversations can "self-populate" the graph with newly discovered concepts over time [31]. This approach keeps everything in one unified graph and maintains a **clean separation of concerns** (no domain-specific code modules to maintain) [31].

Pros: The current strategy is **configuration-driven** and leverages the existing architecture. New regulatory content can be onboarded simply by loading data and config, which is efficient and ensures all tenants benefit from the expanded knowledge base [31]. There is no need to write plugin code or spin up separate databases for each domain; the graph is inherently multi-domain. It also aligns with the "living graph" concept – as usage uncovers new rules or case law, those get added to the shared graph for collective benefit [4] [32].

Cons: The **lack of isolation or feature toggling** is a major drawback of the one-graph-for-all design in a commercial SaaS context. Right now, *all tenants see all domains* in the graph [33]. There is no concept of "this tenant only has access to Domain X". Similarly, there's no built-in licensing tier enforcement – a new domain added is global, and there are *no feature flags or subscription checks* to restrict who can use, say, an advanced "What-If Scenario Engine" or a premium data pack [33]. In essence, **tenant-specific customization is not possible** with the current purely global graph: you cannot enable or disable specific content packs per tenant, and you cannot add a rule that only a particular tenant should see [34]. This is a deliberate design up to now (to maximize shared knowledge), but as a SaaS product, it limits flexibility in packaging features or content for different customer tiers.

Alternative Approaches for Extensibility: Going forward, there are a few approaches to introduce per-tenant extensibility and modularity:

- **A. Feature-Flagged Domain Packs:** In this model, every tenant's subscription would enumerate which domains (and features) they have enabled. For example, a `TenantSubscription` record might list allowed domains like "tax:IE" or "pensions:UK" and enabled features like "scenario_engine" [35]. The application would then simply hide or block access to graph content outside those domains for that tenant (essentially a **feature toggle** approach). The upside is simplicity – it builds on existing structures (just check flags before answering a query) [36]. However, this "binary on/off" gating could get unwieldy as domains proliferate [37]. With dozens of possible domain packs, managing long lists of flags per tenant and enforcing them in every query could become complex. It also doesn't naturally support *partial* access (e.g. read-only vs. full analysis of a domain) – it's either you have the domain or you don't.

- **B. Plugin/Module Architecture:** This approach treats each regulatory domain pack as a **self-contained module** (e.g. an NPM package or separate code plugin) that can be installed or activated per tenant. A `DomainPack` might bundle its node definitions, agent configurations, prompt aspects, and even custom ingress rules in code form [38] [39]. Tenants would dynamically load or be associated with these plugin modules (perhaps "activating" them registers new graph data or agents for that tenant's use). The clear benefit is modularity – domain packs can be developed and tested in isolation, and licensed separately [40]. It also provides a convenient unit for sales (e.g. sell a "UK Pension Pack" as an add-on). The downside is increased complexity: introducing code plugins raises deployment and maintenance overhead [41]. It risks duplicating infrastructure (if each pack needs slight variations of graph logic or ingestion) and could challenge the unity of the core graph. Without careful design, plugins might break the global consistency or require complicated integration points. In short, it's powerful but heavy-weight.

- **C. Tenant-Isolated or Segmented Graphs:** The most straightforward isolation method is to give each tenant their **own graph instance or partition**. This could mean separate Memgraph instances per tenant or logically partitioning the graph by tagging every node/edge with a tenant ID. This ensures perfect isolation – no tenant ever sees another's data or even unused domains – and makes entitlements trivial (a tenant simply doesn't have certain data in their graph if not licensed). However, this approach *sacrifices the core value* of the platform's design: the shared global knowledge. Each tenant's graph would diverge, and any new regulatory content would have to be duplicated across all tenants who need it. It would undermine the "network effect" where the graph gets richer for all users as new information is added. It also complicates maintenance (multiple graphs to keep in sync or update) and raises data consistency issues (if one tenant's graph updates and others lag behind). Thus, **full graph isolation is not ideal** if we want maximum reuse of global knowledge and ease of updates.

- **D. Runtime Query Filters / Overlays (Hybrid):** A middle-ground approach (and the one emerging as **recommended**) is to keep a *single shared graph* but implement **query-time scoping** to achieve per-tenant views [42]. In this model, the platform maintains metadata about which domains/jurisdictions each tenant is allowed, and every query or graph view is filtered accordingly. For example, a tenant's "graph scope" might allow only the jurisdictions `['IE','UK']` and domains `['Tax','Welfare']`, and the system ensures that tenant's queries only return nodes/edges fitting those criteria [43] [44]. This could be done by embedding the filters in graph queries (e.g. requiring a `tenantAllowed` property or checking labels against an allowed list) or at the application layer by post-filtering query results. Additionally, this approach can support **tenant-specific extensions** by overlaying a small number of private nodes per tenant (stored perhaps in a separate table or marked in the graph with a tenant-specific flag) without forking the entire graph [45] [44]. Essentially, the shared graph remains the single source of truth for public regulatory knowledge, but each tenant sees a slice of it plus any custom nodes of their own. This hybrid method requires more sophisticated query logic (and careful performance considerations for filtering), yet it leverages the existing architecture and avoids data duplication [44]. It also dovetails nicely with a feature-flag system for premium capabilities: alongside domain/jurisdiction filters, a tenant's profile can include flags like `scenarioEngine: true/false` or usage quotas [46] that the application checks before enabling certain tools [47].

Assessment: The *current approach* of a unified graph has served well for development – it ensured focus on building the knowledge graph and engines once, for all content. But as a commercial SaaS, it hits a ceiling in extensibility. **Alternatives A and B** (simple feature flags or full plugin modules) represent two ends of a spectrum: one is easy but not scalable in the long run, the other is very extensible but adds complexity. **Approach C (hybrid scoping)** appears to capture the best of both: it keeps one graph (maximizing knowledge reuse and minimizing overhead) but introduces a layer of **tenant-specific scope** to allow modular access and extensions. This aligns with the design philosophy in the docs: keep core reasoning centralized, but use configuration to vary behavior per tenant [20] [22]. Notably, the architecture already contains pieces that facilitate this – e.g. **Profile Tags** are used currently to mark which profiles/personas a rule applies to, functioning as a filter in queries [48]. Similarly, jurisdictions are attached to every rule node, so scoping by jurisdiction is natural. These mechanisms hint that **data-driven filtering** is the preferred path over heavy code branching. Therefore, implementing a robust scoping/entitlement layer on top of the shared graph is likely the optimal extensibility strategy moving forward.

Recommendations

1. Adopt Tiered Graph Scoping with Domain Entitlements: The platform should implement the recommended **hybrid model** where the Memgraph rules graph remains shared, but each tenant is associated with a *Graph Scope* (allowed jurisdictions, allowed domains, and enabled features) [49] [44]. Concretely, define a **tenant subscription schema** (in the Supabase DB) that lists which domain packs or jurisdictions a tenant has access to, as well as feature flags and usage quotas (e.g. number of queries per month) [50] [46]. The application's query layer (the Compliance Engine or GraphQL resolvers) should then enforce these scopes, for example by appending filters on `Jurisdiction` or domain label when retrieving graph data. This will require development of a filtering mechanism (perhaps an allow-list of node labels or a Cypher clause injection based on tenant context), but it **avoids duplicating the graph**. All tenants still hit the same knowledge base, but *see only what they are entitled to*. This approach is scalable: adding a new tenant or domain mostly means updating configuration, not spinning up new infrastructure. It also ensures **global updates propagate to everyone** – e.g. when a new law is added to the graph, any tenant licensed for that jurisdiction will immediately benefit, whereas those without access simply never query it.

2. Implement Domain Packs as Data Modules: To facilitate sales and onboarding, the team can formalize **Domain Packs** as distinct bundles of content that can be toggled. This doesn't require code plugins, but rather organizing the graph data (and related prompt/agent configurations) into logical groups. For instance, an *"IE Tax Pack"* might consist of all Ireland tax-related nodes/edges plus any specialized agent or prompt aspect for tax domain. These packs can be documented and versioned. Technically, they remain part of the one graph, but the **tenant's subscription** will reference them to allow query access [43]. On the backend, maintain mapping from domain pack to the actual labels or node properties that identify that domain's nodes (for example, all Ireland tax rules might have `domain: "TAX"` and `jurisdiction: "IE"` attributes). This way, turning a domain pack "on" or "off" for a tenant is a matter of adding or removing entries in their allowed domain list, which the query filter logic respects. By using data tagging rather than separate graphs, you achieve **maximum reuse** while still gaining modular control.

3. Preserve Shared Graph for Global Knowledge – No Forking: It is recommended *not* to split the underlying graph per tenant. The current architecture's use of a single Memgraph for all tenants is a strength, as it guarantees one canonical source of truth for regulations. Continue to use the **privacy guardrails** (ingress/egress guards) so that no sensitive customer-specific info ever enters that shared graph [6] [12]. This allows the graph to remain safely multi-tenant. If a particular tenant needs to extend the knowledge base with their **own proprietary rules or interpretations**, consider storing those in a separate space (for example, a tenant-specific schema or a parallel set of nodes marked with a tenant identifier that the system keeps separate). These could be merged in *at query time* for that tenant's sessions only. The architecture already hints at this capability with an optional `tenantNodes` concept in the proposed TenantGraphScope interface [45]. Implementing this means a tenant could have a few custom nodes (say, internal policy documents or custom compliance checks) that are only added to query results for them, without ever polluting the global graph. This approach satisfies **tenant-specific needs without data leakage** – effectively a read-time overlay for that tenant.

4. Build Feature Flags and Licensing Hooks: In addition to domain-based scoping, implement a **feature flag system tied to subscription tiers**. The architecture is already primed for this – e.g. different tenants can use different LLM models or toggle the forthcoming Scenario Engine [46]. Formalize these flags in the subscription model (e.g. a boolean for `scenarioEngineEnabled`, `advancedAnalyticsEnabled`, etc.) and enforce them in the application. For example, wrap the Scenario Engine invocation or bulk-export tool in a check that the tenant's plan includes that feature.

This ensures a clear **license-based modularity**, enabling monetization of advanced features. The code snippet for TenantSubscription in the docs illustrates this concept, including quotas for usage [51]. By integrating this, the platform can offer tiered plans (Free vs. Pro vs. Enterprise) cleanly separated by config, not forks in code.

5. Continue Hardening for Enterprise SaaS: Finally, address the remaining *operational gaps* to make the platform truly production-grade for multi-tenant SaaS. This includes adding **billing integration** (so subscription tiers and domain packs tie into real purchase plans) [52], **rate limiting** per tenant to prevent abuse [53], and an admin dashboard or onboarding flow to manage tenants. None of these alter the core architecture, but they are essential complements to the above recommendations. On the technical side, upgrading the event streaming (for graph change notifications) to a scalable pub/sub and fully integrating the authorization model (OpenFGA or similar) will ensure the system can grow to many tenants and complex sharing scenarios [26] [27]. These steps, combined with the **tiered graph scoping design**, will transform the already well-architected prototype into a **scalable, enterprise-ready SaaS platform** that achieves the original vision: a multi-tenant, cross-jurisdictional regulatory intelligence graph with **maximal knowledge reuse and minimal data isolation risk**. The result will allow onboarding new tenants (even in new industries or countries) with configuration rather than code, leveraging the global regulatory graph while still respecting each tenant's entitlements and privacy boundaries. [17] [42]

---

[1] [2] [8] [9] [15] [17] [18] [19] [20] [21] [22] [26] [27] [28] [29] [30] [31] [33] [34] [35] [36] [37] [38] [39] [40] [41] [42] [43] [44] [45] [46] [47] [49] [50] [51] [52] [53] PAAS_SAAS_ARCHITECTURE_REVIEW.md
https://github.com/airnub-labs/regulatory-intelligence-copilot/blob/0234f8cca28c51f659d47e58253cdb7c61583cea/docs/architecture/PAAS_SAAS_ARCHITECTURE_REVIEW.md

[3] [11] [23] [24] [25] [32] schema_changelog_v_0_4.md
https://github.com/airnub-labs/regulatory-intelligence-copilot/blob/0234f8cca28c51f659d47e58253cdb7c61583cea/docs/architecture/graph/archive/schema_changelog_v_0_4.md

[4] [5] [6] [7] [10] [12] [16] [48] schema_v_0_3.md
https://github.com/airnub-labs/regulatory-intelligence-copilot/blob/0234f8cca28c51f659d47e58253cdb7c61583cea/docs/architecture/graph/archive/schema_v_0_3.md

[13] [14] concept_v_0_6.md
https://github.com/airnub-labs/regulatory-intelligence-copilot/blob/0234f8cca28c51f659d47e58253cdb7c61583cea/docs/architecture/copilot-concept/concept_v_0_6.md