ChatGPT

# Vivified Modular Platform – Execution Plan

**Overview:** This plan describes how to evolve **Vivified** (built on the Faxbot foundation) into a production-grade modular framework with plugin-based extensibility. We will design a **3-lane communication model** for plugin interactions, enforce a **plugin-first architecture** with strong security and compliance guarantees, establish core platform services (identity, config, routing, etc.), and provide developer tooling (SDKs, validators, sandbox). Finally, we outline a phased implementation roadmap with clear milestones and deliverables.

## 1. Three-Lane Plugin Communication Model

To route data and requests between core and plugins securely, Vivified will implement a **3-lane communication model** with distinct channels for different interaction types:

- **Canonical Lane:** All plugins exchange information via **canonical data models** and events. This is the primary flow for business data. The core defines universal schemas (e.g. a `CanonicalMessage` format) for common objects so that every plugin speaks a "common language." For example, a HR plugin emitting an event (like a new user onboarding) would publish a canonical `UserCreated` event on the event bus. Other plugins subscribe to these events and transform to their domain as needed. This lane ensures **normalized, decoupled messaging** – as seen in Faxbot's canonical event model which normalized inbound/outbound events across providers [1]. Vivified's Canonical Model Engine will house these schemas and transformations (plugins register transformer functions to convert their internal data to/from the canonical format). Every major entity (user, message, transaction, etc.) gets a canonical representation, enabling seamless inter-plugin interoperability [2]. All canonical-lane messages go through the central event bus (e.g. NATS or Redis Streams), where they can be logged and filtered by the core's policy engine.

- **Operator Lane:** This lane handles **direct, operation-based calls** between services using well-defined APIs and resource IDs. When a plugin needs to perform a specific action on another plugin's domain (beyond passive event listening), it uses an *operator request*. For example, an Accounting plugin might need to fetch user details from the Identity service – it would call a defined API like `GET /identity/user/{user_id}` via the operator lane. These calls are **scoped and authenticated RPC/HTTP calls** mediated by the core (acting as an API gateway or gRPC broker). Crucially, all IDs and parameters used must conform to canonical or agreed formats (e.g. using a global user UUID or a canonical document ID) – these are the "well-defined ID contracts" ensuring both sides refer to the same entities. The core's routing service will expose internal APIs (or a service mesh) for plugins to invoke such operations. Before forwarding an operator call, the core's **permission broker** checks that the requesting plugin and user have rights to that resource (using trait and role policies). This lane provides efficient request/response semantics for common operations (read/write actions) under strict governance. Every operator-lane call is traced and audited by core for compliance (including input/output validation and logging of who invoked what).

- **Proxy Lane:** This is a **restricted, last-resort path** for less-standard or less-trusted interactions. It acts as a controlled gateway when a plugin must perform an action that isn't covered by canonical events or operator APIs – often for ad hoc or third-party integration tasks. Because such interactions might be outside the strict interfaces, the Proxy Lane is heavily sandboxed. For example, a community-contributed plugin that needs to call an external service (or a beta plugin that doesn't yet have a formal contract) would be forced through the proxy lane: the plugin issues the request to the core's proxy service, which then performs the action on the plugin's behalf under watch. The core can enforce extra security here – e.g. only allowing the plugin to reach whitelisted external domains [3], sanitizing responses, injecting additional authentication, or rate-limiting calls. In practice, the Proxy Lane might be implemented as a secure reverse proxy or broker microservice that plugins must use for any **out-of-scope operations**, ensuring the core can log and approve each action. This lane is disabled or tightly limited by default; only plugins marked with certain traits (and admin approval) can use it. It ensures that even "ad hoc" extensions cannot bypass security – they operate in a **monitored container** with the core validating each operation [4].

**Summary:** All three lanes are mediated by Vivified's core. Canonical Lane covers **standardized data flow** (event-driven, loosely coupled) [1] [2], Operator Lane covers **direct service calls** with authorized IDs, and Proxy Lane covers **exception cases** in a controlled sandbox. This multi-lane approach maximizes flexibility while maintaining security boundaries. The policy engine will uniformly enforce authentication, authorization, and auditing across all lanes (no plugin can talk outside its allowance). A high-level illustration:

```
[Plugin A] --(Canonical Event)--> 【Core Event Bus】 --(Event)--> [Plugin B]

[Plugin A] --(Op Request: e.g. getResource X)--> 【Core API Gateway】 --(RPC)-->
[Plugin B]

[Plugin A] --(Ad Hoc Request)--> 【Core Proxy】 --(Supervised Call)--> [Plugin B
or External]
```

In all cases, the core intercepts and governs the communication, ensuring trait-based access policies and logging are applied.

## 2. Plugin-First Architecture and Contracts

Vivified is designed as a **plugin-first platform** – meaning all domain-specific features are implemented as plugins, and even core capabilities are modular where possible. Key architectural principles and requirements for plugins:

- **Polyglot, Containerized Services:** Each plugin runs as an independent service (e.g. a Docker container), allowing developers to use any language or tech stack as long as it can communicate via HTTP/REST or gRPC. Plugins can be deployed as microservices or webhooks that Vivified calls. This isolation follows the idea that plugins "run inside a secure container" logically [4] – if a plugin crashes or misbehaves, it doesn't bring down the core. The core will provide reference base images and templates for popular languages (Python, Node.js, Go) to simplify plugin development and ensure consistent environment.

- **Standardized Plugin Contracts:** Every plugin must implement a well-defined interface contract declaring what it does and how it integrates. Vivified will define **abstract base interfaces** for common plugin categories – for example, CommunicationPlugin, StoragePlugin, IdentityPlugin, etc., each with specific methods to implement [5]. For instance, a CommunicationPlugin might require methods `send_message(CanonicalMessage)` and `receive_messages()` [6], a StoragePlugin might implement `store(data, meta)` and `retrieve(id)` [7], etc. These contracts ensure the core can interact with any plugin in a uniform way. A plugin's *manifest* (see below) will indicate which contract(s) it implements (a plugin could implement multiple interfaces if it provides multiple capabilities). By adhering to these interfaces, plugins become **plug-and-play** – the core knows how to call them, and other plugins can rely on standard behaviors.

- **Plugin Manifest Declarations:** Each plugin ships with a manifest (likely a JSON or YAML) that the core uses at registration time. The manifest explicitly declares the plugin's **traits, dependencies, transformers, endpoints, and lifecycle hooks**:

- *Traits:* Traits are capability or requirement flags that describe the plugin (e.g. `handles_phi` if it handles protected health data, `requires_encryption` if it must only operate on encrypted data, `provides_ui` if it contributes UI components, etc.). Traits inform both the security engine and the UI. For example, a plugin manifest might specify it requires trait "cloud" and "supports_inbound=false" [8], which the core uses to configure the UI and access rules. At runtime, the core will attach these traits to the plugin instance (e.g. `plugin.traits = ['handles_phi','requires_encryption']`) and enforce that only compatible users or services interact with it [9].
- *Dependencies:* If a plugin depends on other plugins or core services, it must list them. For example, an "HR onboarding" plugin might depend on the Identity plugin (for creating users) and an Email plugin (to send welcome emails). The core uses this to ensure all required plugins are present and to determine startup order. Dependencies can be by contract type ("needs an IdentityPlugin present") or specific plugin IDs.
- *Canonical Transformers:* The plugin should declare how it converts its internal data structures to/from Vivified's canonical models. In some cases this can be automatically handled by the plugin's SDK (especially if following standard interface), but for complex or new data types, the plugin might include a transformer module. The manifest can reference these transformer classes or functions. For example, a plugin that introduces a new domain event type would provide a transformer to map it into a `CanonicalEvent` object so other plugins can consume it via the canonical lane.

- *Endpoints & Hooks:* The manifest enumerates the endpoints (API routes or message topics) the plugin listens on and exposes. For a RESTful plugin, this could be its base URL and specific paths for its operations (e.g. `/send_email`, `/list_users`). For event-driven plugins, it might list event types it publishes or subscribes to. Lifecycle hooks are special endpoints or commands like **init** (called when plugin is first loaded), **shutdown** (graceful teardown), or **heartbeat** (for health checks). The core will call these hooks at appropriate times – e.g. invoking an init hook after registration to allow the plugin to load any initial data, or periodically pinging a health endpoint to ensure the plugin is responsive.

- **Centralized Security & Compliance: All security, compliance, and permission enforcement is handled by the core platform – never by plugins themselves.** Plugins should assume they run in a zero-trust container where every action is verified. The core will **validate every plugin operation** and block any that violate policies [4]. Concretely, this means:

- Authentication of user/API calls is done by core (e.g. using API keys or tokens at the gateway). Plugins will never see raw credentials – they only receive identity context (user ID, roles) after core auth.
- Authorization is trait- and role-based and done centrally. The core's policy engine checks user traits vs. plugin traits, as well as plugin trait compatibilities, before allowing any access [9]. If, for example, a plugin has `requires_encryption` trait but the requesting context lacks `encryption_capable`, the core will deny that operation. This **traits-first gating** was a key principle in Faxbot's design (UI and API features gated by traits [1]) and is extended platform-wide in Vivified.
- Compliance requirements like audit logging, PHI (sensitive data) handling, and data retention are enforced by core wrappers. For instance, if a plugin processes healthcare data, the core ensures no PHI leaves the secure environment unencrypted or gets written to logs (Faxbot already avoided PHI in logs [10]). The core can provide utilities to plugins for cryptographic operations so that keys never leave core control.
- **Sandboxing:** Plugins run with least privilege. They cannot directly access the database or file system of the core, nor talk to the network arbitrarily. The core (or container runtime) will enforce network policies so that a plugin can only communicate via the official lanes (e.g. only connect to the event bus, or make HTTP calls to core's gateway/proxy). For example, Faxbot's provider manifests explicitly restricted allowed outbound domains [3]; Vivified will generalize this by sandboxing plugin network access (e.g. using Docker networks or Kubernetes NetworkPolicies). Similarly, file access can be isolated (each plugin gets its own storage mount or uses the Storage service API for shared files).

In summary, **Vivified treats plugins as untrusted by default** – the core container is the secure gatekeeper. Plugins focus solely on their business logic (sending an email, handling HR data, etc.), and the Vivified core handles user interface, security checks, configuration, and cross-plugin coordination. This plugin-first approach ensures maximum extensibility: to add a new feature or integration, you *write a new plugin* (or swap an existing one), rather than modifying core code.

## 3. Secure Inter-Plugin Communication & Orchestration

To enable plugins to work together, we will set up a **secure communication middleware** that all plugins and core services use for messaging. The design combines an **event bus** for loose coupling (canonical lane) with a **gateway/broker** for direct calls (operator/proxy lanes), all overseen by a policy engine:

- **Shared Event Bus:** Vivified will incorporate a message broker such as **NATS** (high-performance Pub/Sub) or **Redis Streams** (persisted log streams) to serve as the backbone for canonical events. Each plugin, upon registration, establishes a connection to the bus (via an SDK or sidecar) and subscribes to relevant topics (e.g. a plugin handling notifications might subscribe to `events.canonical.message.sent` topics). When a plugin publishes an event, it will send it to the bus on a well-defined channel; the core can stamp each event with metadata (source plugin ID, timestamps, trace ID, etc.) for auditing. Because events flow through this central bus, the core can implement a **monitoring subscriber** that listens to all events and applies policies in real-time. For example, the policy engine might intercept a `PatientRecordEvent` from a medical plugin and strip out a field if the target plugin isn't cleared for PHI, or simply log the event and allow it. This ensures **traceability** for every cross-plugin event – we can record which plugin emitted it and which received it (maintaining an audit trail of inter-plugin communication).

- **RPC Gateway for Operator Calls:** For more synchronous operations, Vivified will expose an internal API gateway that brokers RPC-like interactions between plugins. This could be implemented with **gRPC** (defining proto service interfaces for each plugin contract) or simply an HTTP-based gateway in the core. The idea is that a plugin can make a call like `core.invoke(plugin_id=X, operation=Y, payload=Z)` (the SDK will abstract this), and the core will route it to the target plugin's endpoint if permitted. We might register each plugin's service address and available operations in a service registry at startup. The gateway will:

- **Authenticate & Authorize** the call – verify the calling plugin's identity and ensure it's allowed to call the target operation (using the trait-based policy engine and any explicit ACL rules configured by an admin).
- **Translate IDs** – ensure that any IDs passed in the request refer to valid, accessible resources. The core may maintain a mapping of global IDs to plugin-specific IDs if needed. For example, a call to `OrderPlugin.getOrder(order_id)` might accept a global `order_id` that core maps to that plugin's internal ID format.
- **Forward the Request** – either via an internal network call to the plugin's container (e.g. an HTTP call to the plugin's REST API or a gRPC method invocation). This is done over the internal network where both core and plugin are running (e.g. Docker network or Kubernetes cluster network). All traffic is mutually authenticated (the plugin trusts requests coming only from core's gateway, perhaps using mTLS or shared tokens).
- **Collect and Return Response** – the core receives the output from the plugin and then relays it back to the calling plugin or user. Before returning, core may do post-checks (e.g. remove any data that the caller shouldn't see, or logging the response for audit).

The operator lane calls are all **synchronous and request-scoped**, which differs from the fire-and-forget nature of the event bus. We will likely implement **time-outs and fallbacks** here – if a plugin doesn't respond, the core can return an error rather than hanging. Also, to avoid tight coupling, any plugin that's down or unavailable will result in a graceful failure (logged for admins). In essence, the core acts as a broker that **knows about every RPC call** happening between plugins.

- **Event-Driven Orchestration & Workflows:** By combining the event bus and operator gateway, we enable rich orchestration. For instance, consider a workflow: *New employee onboarding* – An HR plugin emits `UserCreated` (canonical event) -> a Workflow plugin listening on that event picks it up and makes an operator call to the IT plugin (`createMailbox(user_id)`) and to an Email plugin (`sendWelcomeEmail(user_id)`). Each of those calls goes through the core gateway with full auditing. The IT and Email plugins might in turn emit events like `MailboxCreated` or `EmailSent` on the bus, which the Workflow plugin or others could listen to. This mix of async events and sync calls allows flexible orchestration while keeping everything **observable** and **governed** by the central policy.

- **Trait-Based Policy Engine:** At the heart of orchestration is the **policy and permission broker**. We will implement a Policy Engine service in core that evaluates:

- **Inter-Plugin Permissions:** Which plugin is allowed to talk to which. By default, plugins might not initiate direct calls to each other unless explicitly allowed. Permissions can be derived from traits and policy rules. For example, any plugin with trait `requires_phi` might only send events to plugins with trait `handles_phi`, otherwise the core drops the event (as hinted in the Vivified design [9]). We can provide an admin-facing policy configuration (like allow/deny lists or trait-logic rules) to fine-tune these interactions. The policy engine hooks into both the event bus (filtering unauthorized events) and the RPC gateway (authorizing calls).
- **User Permissions:** The identity service (see next section) will supply user roles and traits. The policy engine ensures that if a user without admin rights attempts an operation that involves a sensitive plugin, it's blocked. For instance, if a user's role doesn't include "finance" trait, their requests will never be forwarded to an Accounting plugin.
- **Data Access Control:** In addition to controlling *which* plugins can talk, the broker also controls *what data* flows. This can involve content-based rules – e.g., if a document is marked with trait "confidential", only plugins that have `can_receive_confidential` trait will get the full data; others might get a redacted version or an error. Implementation-wise, this could be handled by requiring plugins to label data with traits (as part of the canonical message metadata [11]), and having the policy engine enforce compatibility of those data traits with the destination's accepted traits.

The **permission broker** will log every decision it makes (for audit). We will also integrate a **trace ID** or correlation ID through the entire call chain for each high-level action, so that in logs we can see a user action spawning events and calls across multiple plugins.

- **Secure Transport and Identity:** The communication between core and plugins will be secured. In a Docker compose dev setup, this might be simpler (internal network), but in production (Kubernetes or multi-host) we will enforce TLS for all communications. Each plugin when registering will get credentials (like a JWT or API key) to connect to the event bus and gateway. The event bus can be configured with authentication such that only known plugins (with credentials) can subscribe/publish on specific channels. Similarly, for RPC calls, we use an internal token so that a plugin cannot directly invoke another plugin's endpoint without going through core – the target plugin will only trust requests with a valid core-signed token. This prevents any attempt to bypass policy by calling directly.

- **Monitoring and Recovery:** We will instrument the messaging system so that it's easy to monitor plugin interactions (e.g. count of messages, queue latencies, etc.). If a particular plugin becomes unresponsive or floods the system, core can detect that (e.g. no heartbeat or queue backups) and take action such as isolating that plugin (stop routing messages to it) and alerting admins. The orchestration system should degrade gracefully: for example, if the event bus is down, core might buffer critical events or switch to a degraded mode where only direct calls work, and if a plugin is down, its events/calls are skipped with errors but the rest of the system continues.

Overall, this inter-plugin communication architecture ensures **secure, traceable, and flexible orchestration**. By using a shared bus plus a controlled gateway, we support both loose coupling and direct interactions while keeping the core "in the loop" for every cross-plugin exchange.

## 4. Core Platform Services & Infrastructure

We now outline Vivified's **foundational core services** that underpin the architecture. These core components are responsible for security, configuration, routing, and providing common functionality to plugins. They can be built as a single consolidated application (especially initially) or as microservices that communicate internally – in either case, they are logically distinct modules:

- **Identity & Access Service:** The Identity service manages **users, authentication, and roles/traits assignment**. It is central to enforcement of who can do what. Initially, we can implement a simple user database (e.g. PostgreSQL for persistence) with support for API key auth (as in Faxbot's `X-API-Key` usage [12]) and basic username/password or OAuth integration. This service issues tokens for users and verifies them on requests. It also stores user traits/permissions – e.g., Admin users might have `admin_capable` trait, certain users might have domain-specific traits like `finance_access`. For future-proofing, we'll design it to integrate with external IdPs (LDAP, SAML, OAuth providers) so that enterprises can plug in their existing user directories [13]. The identity service exposes APIs for user management (create user, assign roles, etc.) and is the authority on permission checks (the Policy engine queries it for user's roles/traits). It will also handle session management for the Admin Console login. In summary, this service ensures every request is tied to an authenticated identity and provides the **RBAC foundation** for Vivified.

- **Configuration Service:** Vivified needs a robust config system to manage both **global settings and plugin-specific configurations**. The Configuration service will load configuration from various sources (env files, databases, plugin manifests, etc.) and present a unified view. We will implement a hierarchical config store [14] – meaning there can be system-wide defaults, overridden by environment/instance settings, and further overridden by dynamic admin changes. For example, an "EmailGateway" plugin might have configurable SMTP server credentials; these are stored in the config service (possibly encrypted for sensitive keys) rather than in the plugin container. On startup, the core and each plugin will query the config service for their settings (the core can pass relevant config to plugins via environment variables or API calls). We will also provide Admin Console UI to edit configuration at runtime (with validation). In initial phases, a simple approach is to use a JSON config file (like Faxbot's `faxbot.config.json`) [15], but we will evolve this into a database-backed config with an API (the PRs in vivified repo about hierarchical config suggest this is already in progress). The config service also manages feature flags (e.g. enabling the plugin discovery feature as Faxbot did with `FEATURE_V3_PLUGINS` [15]) and secure storage of secrets (like API keys for third-party services) – possibly integrating with a secrets vault for production.

- **Routing & Orchestration Service:** This is the core's **communication hub** as described in section 3. It encompasses the event bus client, the RPC gateway, and the policy engine. We sometimes refer to it as the "Vivified Core Gateway" or orchestrator. Its responsibilities:

- Maintain the registry of active plugins (their metadata, endpoints, health status).
- Subscribe to all canonical event streams and route events to subscribers (including applying filtering rules via the policy engine).
- Expose internal APIs for plugins to invoke operations on each other, forwarding those requests appropriately.
- Enforce ordering or transactionality if needed for certain workflows (e.g. orchestrating a multi-step transaction across plugins).

- Provide an interface for the Admin Console to observe the system (e.g. listing all plugins and their statuses, viewing event traces).
- In a microservice deployment, this could be split into separate services (one focusing on events, one on RPC calls), but initially a single module can handle both since they are closely related. We will ensure this service is highly available (possibly stateless or using the event bus as its state) so it doesn't become a single point of failure.

- **Storage Service:** The platform will offer a **shared storage and persistence** layer. This twofold: (1) an **Object storage** interface for files/blobs that plugins might need to save (for example, if an HR plugin generates a PDF contract, it can store it via the core rather than manage its own S3 bucket; Faxbot similarly had pluggable storage for fax files – local or S3 [16]). We can integrate with cloud storage (AWS S3, etc.) or local disk, configured via the config service. The Storage service would expose API like `store_file(plugin_id, file)` returning a handle or ID, and `get_file(id)` for retrieval, possibly with access controls. (2) a **Database** for structured data: Core will definitely need a database for its own data (users, config, audit logs). We can also allow plugins to use the core's database for simple needs via a provided abstraction (to avoid every plugin spinning up its own DB). For instance, a small plugin can use a key-value store or get a schema within the core database. However, larger plugins (like an ERP module) might have their own database by design. The key is to make sure any data containing cross-plugin references (like foreign keys linking a record in plugin A to a user in plugin B) are done via canonical IDs to maintain loose coupling. The Storage service might thus also handle **ID generation** and mapping to ensure uniqueness across the system. Initially, we'll set up a Postgres container for core data and optionally for plugin use.

- **Canonical Model Engine:** This component manages the definitions of all canonical data structures and the transformations to/from plugin-specific formats. It is essentially a library used by core and available to plugins (via SDK). It will include a schema registry for canonical models (like schemas for `CanonicalMessage`, `CanonicalIdentity`, etc., possibly versioned). When a plugin registers, it can add or update canonical model definitions if it introduces new ones (subject to admin approval). The engine also holds the *transformer registry*: mapping plugin types or data signatures to transformer functions. For example, if a plugin "Accounting" defines a new event type `InvoicePaidEvent` with its own payload, it also provides `to_canonical(event) -> CanonicalEvent` and `from_canonical(canonical_event) -> plugin_event` functions. These get registered so that when the event bus distributes events, any subscriber plugin that didn't originate the event can automatically get a canonical version. The core uses these transformers whenever it needs to route data between heterogeneous plugins, especially on the canonical lane. We will design the transformer interface and likely enforce that transformers are pure functions for testability. This engine also works closely with the **Trait system**, as canonical data may carry trait tags (metadata) that need interpretation (for security or UI).

- **Plugin Manager & Sandboxing:** A core service will handle **plugin registration, validation, lifecycle and sandboxing**. The Plugin Manager is responsible for:

- **Registration:** when the system starts (or when a new plugin is hot-added), the manager reads the plugin's manifest and validates it (see below). It then possibly **launches the plugin container** (if not already running). In a Docker Compose dev environment, plugins might be started manually, but in a production setup, Vivified could have a controller that auto-deploys plugin containers (possibly integrated with Kubernetes operators). Initially, we assume plugins are defined in the docker-compose or k8s manifests, and the manager just needs to coordinate handshake. It will call an initialization sequence: e.g. send a registration request to the plugin's endpoint or read a well-known HTTP endpoint like `/manifest` from the plugin to verify it's online and to fetch any runtime info.

- **Validation:** The manager uses a **Plugin Manifest Validator** (see developer tools) to ensure the plugin's manifest meets schema and security requirements. It will check things like: unique plugin ID and name, allowed traits (no conflicting traits), declared endpoints match the interface of the plugin type, dependencies are available, and the plugin's Docker image or code signature matches what's expected (to prevent tampering). If any check fails, the plugin is not activated. If dynamic installation is allowed (like an admin adding a plugin at runtime), these validations happen before allowing it.
- **Sandboxing & Resource Isolation:** When launching plugins, the manager will apply sandbox settings. For Docker, that means using a minimal runtime profile (no root user, limited memory/CPU if needed, network access only to core and approved external sites). For example, we might run plugin containers with a specific Docker network that only includes the core and possibly internet via proxy. We will also mount a volume for plugin logs or data if needed, but isolate it from other plugins. In Kubernetes, we'd apply a PodSecurityPolicy or Restricted Pod template to each plugin deployment. The manager ensures each plugin runs under these constraints and can optionally integrate with security tools (like scanning plugin container images for vulnerabilities before running).

- **Lifecycle Management:** The manager monitors plugin health (perhaps via a periodic heartbeat ping or by subscribing to the event bus for health events). If a plugin stops unexpectedly, the manager can attempt to restart it or mark it as unavailable (so core won't route calls to it). It also handles graceful shutdown: when Vivified core is shutting down or when a plugin needs removal, the manager calls the plugin's shutdown hook to let it clean up. We will design the manager such that **plugins can be added, updated, or removed without bringing down the whole system** – enabling online updates. This could mean versioning plugins and possibly running two versions side by side during a hot swap (advanced, could be tackled later). For now, ensure the manager can disable a plugin (stop routing to it) and unload it on command.

- **Trait-Based UI Adaptation:** Vivified includes an **Admin Console (GUI)** that is "traits-first", meaning the UI automatically adapts to the active plugins and user permissions. We will extend the approach used in Faxbot's Admin UI [1] [17]. The core will have an **UI Schema Generator** that aggregates information from all plugins to inform the frontend what to display:

- The Plugin Manager exposes a list of installed plugins, their status, and their declared UI components or config screens. For instance, if an Accounting plugin provides a UI module for managing invoices, the core will include that in the admin UI navigation (for users who have access).
- Each plugin's manifest or a separate UI manifest will describe what admin UI elements it contributes: e.g., menu items, forms for configuration, dashboards, etc. We'll define a schema so that plugins can supply UI in a decoupled way (possibly via a React component bundle or a simple JSON form schema that the core UI knows how to render). Initially, we might not support fully custom UI, but at least the core UI will show **plugin configuration screens** (fields defined in manifest). Faxbot v3's UI, for example, lists providers with toggles and schema-driven settings forms [17]; Vivified will generalize that for any plugin settings.
- **Dynamic Feature Gating:** The trait engine in core will ensure that UI elements are shown only when relevant. For example, if a plugin is disabled or not present, its menu doesn't show. If a plugin is present but a user lacks permission (their traits/role are incompatible), the UI either hides that

plugin's sections or shows them disabled with a message. This dynamic rendering is accomplished by the frontend querying an endpoint like `/admin/features` or using the `/admin/providers` and `/admin/config` data as in Faxbot [1]. The data will include trait flags. For instance, core could send a structure: `{ plugin: "EmailGateway", traits: ["notifications"], allowed: true/false for currentUser }`.

- **Unified UX:** The Admin Console remains the single control center. Whether it's configuring core (identity settings, global config) or managing plugins (enable/disable, plugin-specific configs), it's all done through this GUI. The core services (identity, config, etc.) expose REST endpoints that the GUI uses. We will leverage the existing React + MUI admin console from Faxbot as a starting point, then modularize it. Possibly we'll keep it as part of the core container (serving the React app via the API server) [18] for simplicity in deployment. Eventually, supporting custom UI components from plugins might involve loading plugin-provided UI code (which is a complex topic, potentially using micro-frontend architecture), but initially, we can get far with schema-driven forms and informational pages.

- **Logging & Audit Service:** A centralized logging and audit service will ensure **everything is recorded for debugging and compliance**:

- **Application Logs:** The core will aggregate logs from itself and, if possible, from plugins. We'll set up a logging framework where each core service logs to a common sink (e.g. console or file, which can be picked up by Docker logging or a sidecar like Fluentd). For plugins, we encourage them to use STDOUT logging too, which operators can aggregate. In a production cluster, we might integrate with ELK/EFK stack or a cloud logging service to gather all logs. We will standardize log format to include context (timestamp, plugin or core module name, correlation IDs, user IDs if applicable).
- **Audit Trails:** For security-sensitive events, the Audit service will produce structured audit records. These include: user login attempts, configuration changes, any administrative actions (like enabling a plugin), cross-plugin operations (like plugin A called plugin B's API), and data access events if required by compliance. Each audit record will have who (user or system component) did what, when, and outcome. Audit logs will be stored in an append-only manner (e.g. in a separate database table or a WORM storage) to prevent tampering. The Admin Console will have an Audit view for admins to review these events. We might also provide an alerting mechanism for certain critical events (like a high-severity security rule violation could trigger an email or notification to admins).

- **Metrics & Monitoring:** Core will include instrumentation to emit metrics (like Prometheus metrics). Key metrics: number of active plugins, message throughput, response times for operator calls, CPU/memory per plugin, etc. We will deploy a monitoring stack (Prometheus/Grafana or similar) as part of the production compose/K8s to allow visualization of system health. Each plugin template will include hooks to register its own metrics as well. This way, developers get insights into their plugin's performance and usage. For example, the Email plugin can expose "emails_sent_count" and the core will scrape that and incorporate into the global dashboard.

- **Additional Core Features:** Vivified core also includes some advanced features out-of-the-box (as noted in the vision):

- **AI Integration Guardrails:** Although not the main focus of this plan, the core platform has provisions for connecting AI/LLM services (via Model Context Protocol servers) with security filters

[19] . We will keep this as an optional module (possibly disabled unless needed). It's mentioned for completeness, as it's part of Vivified's appeal (AI with compliance).
- **Multi-tenancy and Scalability:** In future phases, if needed, the core should be designed to handle multiple tenant organizations (each with isolated data and plugins), as well as scale horizontally (multiple instances of core services). For now, our modular design (separating stateless services and using message bus) lays the groundwork for scaling. The identity service could be extended for multi-tenant user directories, and the config service could support tenant-specific config layers.

To illustrate the separation of concerns between core and plugins, consider this breakdown (from Vivified's architecture description):

```
Vivified Core (Platform Foundation) – handles enterprise concerns:
├── GUI-First Admin Console (global UI for config & monitoring)
├── Traits-First UI Rendering (dynamic UI based on plugin capabilities)
├── Multi-Backend Provider System (adapters for various services)
├── Canonical Event & Error Model (universal message schemas)
├── Security & Compliance (audit trails, HIPAA/SOC2 alignment)
├── Identity & Permissions (users, roles, SSO integration)
├── Messaging & Routing Engine (event bus, RPC gateway, orchestration)
└── Plugin Sandbox & Security Boundaries (isolation, policy enforcement)

Your Plugins (Extensible Modules) – handle domain-specific logic:
├── Domain-Specific Providers (e.g. integrations like Salesforce, Stripe)
├── Business Logic Workflows (e.g. HR onboarding process plugin)
├── Custom UI Extensions (if any, e.g. reporting dashboards)
├── Data Processing Components (transform or analyze data streams)
└── Communication Adapters (channels like Email, SMS, Fax via providers)
```

*(Adapted from Vivified's documentation, highlighting how the core delivers common infrastructure while plugins implement specific features [20] [21] .)*

In summary, the core services establish a **secure, configurable, and observable foundation**. They enforce that all plugins operate within known guardrails and they provide the "glue" (UI, identity, messaging, storage) that turns a collection of plugins into a cohesive platform.

## 5. Developer Experience and Plugin SDKs

To encourage a vibrant plugin ecosystem, Vivified will provide a first-class **Developer Experience (DX)**. Key components include SDKs, tools, and environments that make building and testing plugins straightforward:

- **Multi-Language SDKs:** We will deliver SDKs or adapter libraries for at least **Python, Node.js, and Go**, since those are common for service development. The SDKs abstract away the low-level details of communicating with Vivified core, so developers can focus on implementing the required interface methods. For example, the Python SDK might provide:

- A base `Plugin` class and specific mixins for each contract (e.g. `class MyPlugin(CommunicationPlugin)` that already includes a Flask or FastAPI server setup to handle incoming `send_message` calls from core).
- Utilities to publish events to the bus easily (like `self.publish_event(canonical_event)` which under the hood attaches plugin ID and uses the message bus client).
- A client to call other plugin services via the core gateway (perhaps as simple as `self.invoke('TargetPlugin', operation, payload)`).
- Built-in support for registering with the core on startup (perhaps the SDK can call the core's registration endpoint or handle heartbeats automatically).
- Logging and metrics integration that sends data to core's aggregators in the proper format.

The Node.js and Go SDKs would offer analogous functionality (maybe using Express or gRPC in Node, and net/http or gRPC in Go). Ensuring the SDKs present an **identical API surface** across languages was a goal of Faxbot's SDKs [22] and remains so for Vivified. This means, for example, sending an event or retrieving config should look similar in all languages. Where possible, the SDKs will also enforce compile-time or runtime checks that the plugin respects the contract (e.g. if you claim to be a StoragePlugin but haven't implemented `store` method, the SDK can warn or refuse to register).

- **Plugin Manifest and Code Templates:** We will provide a **CLI tool** (e.g. `vivified` CLI) to generate plugin scaffolding. As highlighted in the Vivified documentation, running a command like `vivified create-plugin --type communication --name my_plugin` will produce a skeleton [23]. This includes:
- **Project structure** with best practices (for instance, a Python plugin might get `app.py` with a Flask app, a `plugin.json` manifest, and Dockerfile).
- **Stub implementations** of required interface methods (with `TODO` comments for developer to fill in logic).
- **Example transform functions** for canonical data if applicable.
- **Unit test stub** to get the developer started with testing their plugin logic in isolation.
- If the plugin type involves UI, it could also scaffold a placeholder for a UI component or config schema.

The goal is that a developer can run this generator and immediately have a minimal plugin that registers with the core and responds to, say, a health check. They can then incrementally implement the real functionality. The manifest file in the template will contain sections for them to declare traits, dependencies, etc., with documentation in comments.

- **Manifest Validation Tool:** To reduce errors, we will create a **Plugin Manifest Validator** as both a standalone tool and integrated into core's plugin manager. This could be a simple command (`vivified validate plugin.json`) that checks the manifest against a JSON Schema. It will ensure required fields are present (id, name, version, etc.), traits are recognized, dependencies refer to known contract names or plugin IDs, and that endpoint definitions make sense. If we maintain a central catalog of trait names and contract types, the validator checks correctness (e.g. if type is "communication", do we see expected actions like send/receive defined?). This helps developers catch mistakes early. The core will run the same validation on startup for safety, but offering it as a dev tool means faster feedback.

- **Testing Sandbox Environment:** Developers should be able to **test their plugins locally** in an environment simulating the Vivified core. We will provide two approaches:

- **Lightweight Simulation:** The SDK could include a mode where you run your plugin in a special way (like `PluginDevServer` mode), which starts a dummy core thread that can accept one plugin registration. This dummy core would simulate the event bus (looping back events) and allow the developer to call their plugin's methods directly or via SDK calls, asserting responses. Essentially, a mini in-memory harness to do unit tests without needing the full platform up. This is useful for automated testing (e.g. CI pipelines for plugin projects).
- **Docker Compose Dev Environment:** We will extend the repository with a `docker-compose.dev.yml` (or a profile) that can spin up the core platform and a sample plugin together. For example, it might mount the developer's plugin code into a container for rapid iteration. The environment could include monitoring tools as well. The idea is to let the developer run "platform + my plugin" easily and interact via the Admin Console or API to verify integration. We might include a couple of dummy plugins in the default compose for demonstration (like a hello-world plugin that echoes messages) that developers can reference or even replace with their own during testing.

Additionally, we plan to support **hot-reload or iterative development** where possible: for instance, if the plugin runs inside Docker, using volumes to allow code changes without rebuilding images constantly.

- **Documentation and Examples:** Comprehensive **developer docs** will accompany the SDKs, including how-to guides for common plugin scenarios (e.g. "Building a Notification Plugin", "Integrating an External API as a Plugin"). We will maintain a **Plugin Developer Guide** that covers the manifest file reference, traits list, core API endpoints, etc. Moreover, the two reference plugins (discussed in the next section) will serve as **canonical examples** – their source code will be available as templates. Perhaps we'll include them in the repository under `examples/` or even provide them as starting points via the CLI generator.

- **Security Guidelines and Linting:** Because plugins run in a sensitive environment, we'll also provide developers with **security guidelines** (for example, avoid storing PII in logs – core will filter it anyway, but plugin authors should be mindful; or how to handle secrets passed from core, etc.). Possibly a static analysis tool or linter could be included in SDK to check for risky patterns (this could be a later enhancement). At minimum, documentation will highlight best practices for writing compliant plugins.

- **Plugin Marketplace (Future):** Though not immediate, the architecture anticipates a **plugin registry/marketplace** where developers can share plugins. We see hints of this in Vivified's docs (marketplace and discovery features under development [24]). For now, our plan includes making the plugin contracts and packaging consistent so that a plugin is easy to publish as a Docker image + manifest that others can reuse. We will use semantic versioning for plugins and ensure backward compatibility of core contracts when possible to support an ecosystem of third-party plugins.

By prioritizing DX, we ensure that extending Vivified is not reserved for the core team, but open to any developer. A smooth SDK and testing workflow will greatly accelerate adoption. Our aim is that a developer can go from idea to a working plugin in **hours, not weeks**, with the confidence that if they follow the framework, their plugin will automatically inherit enterprise-grade security and integration.

*(As a concrete example of the developer tooling, Vivified already envisions a CLI generator that produces a plugin skeleton including contract stubs, canonical model hooks, admin UI components, and SDK integration* [23] *. We will implement this as one of the early deliverables to jumpstart plugin development.)*

## 6. Phased Development Plan (Milestones & Deliverables)

Finally, we outline a realistic step-by-step execution plan to build this platform. The plan is divided into **phases** with specific milestones, deliverables, and an indicative sequence. We also describe the repository structure and how services will be containerized for each phase. This ensures clarity on implementation order and helps track progress.

**Repository Layout:** We will structure the `vivified` repository to separate core modules, plugin examples, and tooling. An initial layout could be:

```
vivified/
├── core/                # Core platform source code
│   ├── identity/       # Identity & auth service
│   ├── config/         # Configuration service
│   ├── policy/         # Policy engine and trait logic
│   ├── messaging/      # Event bus and RPC gateway integration
│   ├── plugin_manager/ # Plugin registration & sandbox logic
│   ├── ui/             # Admin Console frontend (and possibly backend serving
code)
│   └── ... (other core utils, common models)
├── plugins/             # Reference or built-in plugins
│   ├── example_email_gateway/    # Example plugin 1
│   └── example_user_management/  # Example plugin 2
├── sdk/                 # SDKs for plugin development
│   ├── python/
│   ├── nodejs/
│   └── go/
├── docs/                # Documentation for architecture and plugin development
├── tools/               # Dev tools (manifest validator, CLI generator source)
├── docker-compose.yml   # Compose file for core and example plugins
├── k8s/                 # Kubernetes manifests or Helm charts for deployment
└── etc...               # Config templates, CI/CD config, etc.
```

*(This structure may evolve, but it provides clear separation: core vs plugins vs SDK. It also aligns with delivering an SDK and example plugins as part of the repo.)*

**Service Boundaries (Containers):** In a development environment (Docker Compose), we will run the following containers: - **vivified-core:** the main backend (could be a single container including identity, config, policy, messaging as separate threads or FastAPI endpoints within one app to start). We might call this service simply `api` (similar to Faxbot's usage) [25] . - **event-bus:** e.g. a NATS server or Redis instance container, if using those for messaging. - **database:** a PostgreSQL container for storing identity and config (and potentially audit logs). - **plugin-email-gateway:** container running the example Email Gateway plugin (flask/express app, etc). - **plugin-user-mgmt:** container running the example User Management plugin. - (Optional) **admin-ui:** if the Admin UI is built separately (could be a static site served by core, so this may not need its own container). - (Optional) **monitoring stack:** e.g. a Prometheus and Grafana container for metrics (perhaps included in a separate compose profile).

In Kubernetes, each of these would be a Deployment/Service (with perhaps the two plugin examples as separate deployments, core as a deployment, and we'd use a ConfigMap/Secret for .env values).

Below are the phases with tasks and outcomes:

## Phase 1: Core Scaffolding & Plugin Interface Baseline

**Objective:** Set up the basic project scaffolding, get the core application running with minimal functionality, and define the plugin interface structures.

**Tasks:** 1. **Initialize Repository & CI:** Create the `vivified` repository (if not already). Set up basic README, license, and CI pipeline for building Docker images and running tests. Define coding standards and linters (e.g. black/flake8 for Python, ESLint for Node portions). 2. **Core Skeleton App:** Implement a skeletal core application (e.g. a Python FastAPI or Node Express server) that can start up and expose a health-check endpoint (`/health`). This will later host the identity, config, etc., but initially it's just a placeholder service responding "Vivified core running" at `GET /health`. Containerize this as **vivified-core** image. 3. **Plugin Interface Definitions:** Define the standard plugin interfaces in code (could be as abstract base classes or Protocols in Python, or TypeScript interfaces for Node, etc.). Include classes for CommunicationPlugin, StoragePlugin, IdentityPlugin, etc. with method stubs [5]. Also define the canonical model data classes (e.g. `CanonicalMessage`, `CanonicalIdentity`, etc.) in a shared module [2]. At this stage, these are just definitions; implementation will come later. 4. **Basic Plugin Registration Flow:** Implement a very simple plugin manager in core that can accept a plugin's registration. For now, this might be a hard-coded list or a dummy endpoint. For example, core could read a static config file listing two example plugins. We'll create data structures for plugin manifests and have the core load those on startup. No real validation yet, just log that plugin X is "registered". This sets the stage for dynamic loading later. 5. **Example "No-op" Plugin:** Create a minimal plugin (e.g. `example_user_management`) as a proof of concept. It can simply respond to a ping or implement a trivial interface (like an IdentityPlugin that has an `authenticate` method which always returns a dummy success). The goal is to exercise the plugin interface. Run this plugin in a container that registers itself with core. Possibly, we implement a `/register` endpoint in core and have the plugin call it on startup with its manifest. At this stage, security isn't enforced on that registration. 6. **Docker Compose Up:** Write a `docker-compose.yml` that starts core, the database (even if core not using DB yet, include it for future), and the no-op plugin. Verify that all come up and can communicate minimally (perhaps core logs show plugin registered). This will flush out any network configuration needs. 7. **Deliverable:** By end of Phase 1, we have a **running platform skeleton**: the core service is up and one example plugin is running and recognized by core. There won't be real functionality, but the structure (directories, containers, initial classes) is in place. We also have documentation in the README about how to run this minimal system. This deliverable proves the basic plumbing (container wiring, config, etc.) is correct and provides a foundation for subsequent phases.

## Phase 2: Core Services Implementation (Identity, Config, Basic Security)

**Objective:** Build out the key core services (Identity & Auth, Configuration management) and enforce basic security and trait mechanisms. By the end of this phase, the system should support real user login/auth and loading configuration for plugins, and enforce trait-based access checks in a simple form.

**Tasks:** 1. **Identity Service:** Implement user authentication in core: - Set up a database schema for users (maybe using SQLAlchemy or Prisma or TypeORM depending on language). Include fields for username, password hash (if native auth), API keys, and trait/role assignments. - Implement API endpoints: e.g. `POST /auth/login` (issue JWT or session cookie), `GET /auth/me` (to get current user info), and `CRUD /admin/users` (for managing users, which can be protected to admins). - Password-based auth for now (storing salted hashes), and support multiple API keys as Faxbot did [12] for programmatic access. - Also create a middleware in core that authenticates every incoming request (for both UI requests and plugin-to-plugin calls) using these credentials. - Possibly integrate a simple RBAC: define some roles (Admin, Developer, Viewer etc.) and assign to users, with corresponding traits (like Admin role -> `admin_capable` trait). 2. **Trait & Permission Engine (Basic):** Implement the core data structures for traits and a simple check function. For now, this can be a utility that given a user trait list and a plugin trait list, returns allowed/denied as per a basic rule (e.g. all plugin traits must be contained in user's traits, otherwise deny). Use the example given [9] for logic. Integrate this check into critical points: e.g., if a user calls a plugin operation via core, core uses this to decide. Also use it to filter UI (if building UI this phase). - We can maintain a list (or database table) of which traits are required for which plugin (basically plugin's own trait list). - In this phase, we might hardcode or configure a few core traits (like `admin_only` for certain operations). 3. **Configuration Service:** Implement config loading: - Decide on config storage: likely use the same database to store key-value pairs or JSON blobs for config. Alternatively, start with a `.env` and JSON files approach (like Faxbot) and plan to migrate to DB. To align with vivified PRs, perhaps implement a simple hierarchical config: environment variables override config file, and DB overrides those if present. - Provide an API endpoint `GET /admin/config` that returns current config (filtered by permissions so only safe info is shown to normal users) [1]. And `POST /admin/config` to update settings (admin-only). - Ensure plugin-related configurations are included. For example, core should store which plugins are enabled/disabled, and any plugin-specific settings (like an Email plugin's SMTP server). This could be stored as part of plugin manifest or separately. In this phase, keep it simple: plugin manifest can come from a config file, and the service just reads that on startup. - Plan for Phase 3 where plugin enable/disable can be toggled via this service. 4. **Secure Plugin Registration:** Enhance the plugin manager to incorporate security: - When a plugin registers, require it to provide an authentication token or perform a handshake. For now, this might be basic (since the plugin is within a trusted network), but lay groundwork for verifying a plugin's identity (maybe each plugin has an ID and pre-shared secret from config for phase 2). - Validate the plugin's manifest using the upcoming validator (if ready) or at least check required fields manually in code. - Store the plugin info in a core registry (in-memory dict or DB table). Mark the plugin as active and store its traits, endpoints, etc. The identity service might also create a pseudo-user/principal for the plugin to use in internal calls. 5. **Initial Admin Console (Backend):** At this point, consider setting up the Admin UI backend routes. This includes: - Endpoint(s) for listing plugins and their status (`GET /admin/plugins` returning names, versions, enabled/disabled). - Endpoints for the UI to retrieve trait lists, available settings, etc. For example, `GET /admin/providers` in Faxbot returned active providers and their traits [1]. We'll implement similar: e.g. `GET /admin/plugins` returns each plugin's manifest info and traits. - Serve a basic static page for admin (could just be a placeholder HTML or a simple React app that calls the above APIs). If front-end is not tackled yet, at least verify through an API client that these endpoints return correct data. 6. **Polish Example Plugins to Use Core Services:** Update the example `EmailGateway` and

`UserManagement` plugins (which we scaffolded earlier, perhaps as no-ops) to actually utilize the new core services: - For instance, the UserManagement plugin can implement the IdentityPlugin interface now: it can defer authentication to core (or simply call core's identity service if needed). However, since Identity is core, perhaps UserManagement plugin in our context is more of a placeholder for some user domain logic (if any). Maybe instead, make the second example plugin a "Notifications" plugin which listens for events and sends emails. - The EmailGateway plugin can be made functional: give it a config (SMTP server credentials) via the config service, and have it expose an endpoint `/sendEmail`. It would subscribe to a canonical event like `MessageSendRequest` on the bus or wait for direct calls. This might be early to implement fully; at least ensure it can fetch its config from core (e.g. core passes it through env or plugin calls core `/admin/config`). - Essentially, have at least one plugin demonstrate reading from config service and performing an action when invoked via an API call or event. 7. **Deliverable:** Phase 2 deliverables include **user login and trait-based auth in core** (one can log into the Admin API with an admin user), **persistent config management**, and a **secure plugin registration workflow**. The system should enforce that a non-admin user cannot, for example, enable a plugin or access restricted plugin data (tested via the trait policy checks). We also expect to see the Admin Console start to come alive: by browsing `/admin/plugins` or similar, one can see the installed plugins and their traits. Documentation will be updated for how to set up initial users and config. This phase essentially turns the skeleton into a minimally functional platform with real auth and config – a foundation for the heavy lifting in Phase 3.

## Phase 3: Inter-Plugin Communication Backbone

**Objective:** Implement the core communication infrastructure: set up the event bus, enable plugins to publish/subscribe events, implement the operator RPC mechanism, and integrate the policy checks into these flows. By end of Phase 3, two reference plugins should be able to communicate through the core (e.g. one raises an event, another receives it; one calls an API on another via core).

**Tasks:** 1. **Integrate Event Bus:** Choose the event streaming technology (e.g. NATS). Add a **messaging broker** container to docker-compose (e.g. `nats:latest` or Redis). In core's messaging module, implement connection to this broker: - The core will act as an admin client on the bus, able to create subjects (topics) and maybe listen to all. - Establish a convention for topics, e.g. `plugin.{pluginID}.>` for plugin-specific channels, and `events.{eventType}` for canonical events. Or possibly use NATS wildcard subscriptions for all events. - Update the plugin SDKs to connect to the bus as well. For example, using NATS libraries in Python/Node to subscribe/publish. The SDK should hide details such that a plugin developer can do `self.publish_event(canonical_msg)` and under the hood it does something like `nats.publish("events.UserCreated", msg.serialize())`. - Implement core's **event listener** (Policy Engine's hook into bus): core subscribes to all `events.*` and on receiving any message, logs it and enforces policy. If a message is disallowed (e.g. due to trait mismatch), core can either drop it or redact parts of it before allowing it through (depending on policy rules decided). - Basic functionality: allow all events for now (in dev mode), just ensure messages can flow from one plugin to another via bus. - Test case: have plugin A publish a test event, plugin B subscribe and log it. Confirm through logs or a simple UI that the event was delivered. 2. **Implement Operator RPC Gateway:** Develop the mechanism for direct calls: - Decide between internal REST vs gRPC. For quick implementation, we might use REST calls through the core (since we already have HTTP in core). e.g. a plugin can send an HTTP request to core at `/gateway/{targetPlugin}/{operation}`. Core will authenticate the caller (perhaps with a token the plugin includes, issued at registration), then look up target plugin's address and forward the request. - Maintain a registry of plugin service endpoints. E.g. when a plugin registers, it might provide its service URL (or if on same docker network, core can resolve by container name). In compose, we can use service DNS names.

Alternatively, plugin might expose a callback endpoint that core knows. - Support a few basic operations for demonstration: e.g., define that the UserManagement plugin (or Identity core service) has an operation `get_user_info` that returns user profile by ID. The Email plugin might call this via core to get a user's email address before sending a message. Implement this flow: Email plugin -> core `/gateway/user_mgmt/get_user_info?user=123` -> core verifies -> core calls `http://user_mgmt_plugin:port/get_user_info/123` -> gets result -> returns to Email plugin. - Similarly, implement a call in reverse: maybe UserManagement plugin calling Email plugin's `send_email(userId, content)`. But we can simulate one path if enough. - Include timeouts and error handling: core should respond with appropriate error codes if target is down or returns error, etc. - At this point, enforce basic trait checks: e.g. mark the `get_user_info` operation as requiring the caller to have `can_view_user` trait; core checks the calling plugin's traits. Also ensure user context: if this call is initiated by a user action (say an admin clicked "resend welcome email"), propagate the user identity and ensure the user has rights. This might mean including a user token in the initial call from plugin to core, which core then uses to evaluate permissions for the forwarded call. 3. **Proxy Lane Setup:** Implement a rudimentary proxy service (could be part of core gateway or separate): - This could be as simple as core exposing an endpoint `/proxy?url=<external_url>` that plugins can call to have core fetch something on their behalf. Core would check an allowlist for the plugin (from manifest's `allowed_domains` [3] ). For example, if a plugin wants to call an external REST API, instead of doing it directly, it sends the request to core's proxy. Core verifies the domain is in plugin's allowed list and then performs the external HTTP call, streaming back the response. - This ensures no plugin code is directly opening sockets to arbitrary hosts. It also allows core to log those external calls (for audit) and possibly cache or filter them. - Test this with a scenario: maybe the Email plugin needs to call an external email API (if not using SMTP) – use the proxy. Or simply allow a test plugin to fetch a known URL via proxy to see it working. - Security: ensure this proxy route itself is protected (only accessible to authenticated plugins, not public). 4. **Advanced Policy Rules:** Expand the Policy Engine to cover more cases now that events and calls are flowing: - Implement the logic for trait-based event filtering. For example, define that certain event types carry a required trait. If a plugin without that trait subscribes, core either does not deliver or masks sensitive fields. This might require labeling events with a security level. In practice, we can simulate: e.g., tag the `UserCreated` event with trait `contains_pii`, and ensure only plugins with `handles_pii` trait actually get the full event. - Introduce an internal config for allowed plugin-to-plugin calls (maybe not a UI yet, but a config map: e.g. Email plugin is allowed to call Identity plugin's `get_user_info`, but not allowed to call Finance plugin's `get_account_balance`, etc.). If a disallowed combination is detected, the gateway refuses the call. - Ensure **audit logging** for inter-plugin actions: every event delivered, every RPC forwarded should create an audit log entry like "Plugin A invoked operation X on Plugin B – allowed/denied by policy Y". 5. **End-to-End Use Case Demo:** By the end of Phase 3, demonstrate an end-to-end use case involving both example plugins: - **Example:** *User Onboarding Workflow* – Suppose the UserManagement plugin (simulating an HR system) creates a new user. It emits a `UserCreated` canonical event on the bus. The EmailGateway plugin is subscribed (since it needs to send a welcome email). Core sees the event, checks policy (allowed because EmailGateway has trait `notifications` which is compatible with that event), and delivers it. EmailGateway receives it, then uses the operator lane to fetch additional info: it calls core's gateway to invoke `IdentityPlugin.get_user_permissions` (or a profile fetch) to personalize the email. Core brokers that call with proper auth. Then EmailGateway sends out an email (we could log it or actually send if SMTP configured). It might then emit an `EmailSent` event which core logs and maybe UserManagement plugin listens for to mark the onboarding complete. - We should simulate this flow and ensure each step works and is logged. We don't necessarily need a UI for it yet, this can be observed via logs or a test script. 6. **Deliverable:** Phase 3 yields a **functional communication layer**. Deliverables include: - The event bus integration with at least one real event going through. - The RPC call gateway with an example call succeeding (and denied calls tested). - The proxy mechanism for external calls in place. - Policy enforcement in effect (demonstrated by intentionally violating a rule and seeing core block it). - Updated documentation on how to subscribe to events, how to define plugin allowed domains, etc. - At this point, Vivified is capable of doing real work across plugins, albeit with minimal UI. It sets the stage for adding more plugins and features with confidence in the underlying communication.

## Phase 4: Security Hardening, Compliance & Advanced Features

**Objective:** Strengthen the platform's security and compliance features (fine-grained permissions, audit UI, sandbox enhancements) and flesh out remaining core capabilities like the Admin Console UI and any enterprise features (e.g. AI guardrails integration if needed). Also refine any rough edges from earlier phases (error handling, scaling concerns).

**Tasks:** 1. **Fine-Grained Access Control:** Expand the trait/role system: - Implement role hierarchy or group membership if needed (e.g. group of users can be assigned a trait collectively). - Integrate permission checks into all Admin API endpoints. For instance, only Admin trait can hit `/admin/config` or enable/disable plugins. Ensure these are enforced server-side (even if UI hides it, the API must enforce). - Possibly introduce a **policy DSL or configuration** so that certain rules can be adjusted without code. For example, an admin could define a custom rule like "Plugin X can only access data from department Y" – though this might be too granular for now. At minimum, ensure the infrastructure allows adding such rules later. - If multi-tenancy is a target, start scoping data by tenant ID, but this might be beyond current scope. 2. **Compliance Logging & Alerts:** Finalize the Audit logging: - Write audit logs to a separate store. Maybe a new table `audit_log` with structured fields (timestamp, actor, action, object, outcome). - Build an **Audit view in Admin Console**: a page where an administrator can see recent actions. This likely involves an API like `GET /admin/audit?since=...`. - Implement log rotation or archival policy for audit logs (to handle growth, or forward to an external SIEM system if needed in future). - If applicable, add **alerting**: e.g. if the policy engine blocks an action due to a security rule, maybe fire an alert event that a Security plugin could pick up or simply log it with high severity. Perhaps integrate with email/SMS to notify system admin on critical events (this could even be done via a special "Notification plugin" – eating our own dog food by using a plugin to send alerts). 3. **Plugin Sandboxing & Isolation Review:** At this stage, review how plugins are deployed and tighten security: - Ensure each plugin container runs with a minimal OS image and user privileges. We can adopt Docker best practices (use non-root user in Dockerfile, read-only file system if possible, etc.). - Network: Implement network segmentation in Docker Compose – e.g. create two networks, one for core<->plugins, and an optional one for core's external access. That way, plugins might not even have a default route to the internet, forcing them to use core's proxy. In Kubernetes, this would be done via NetworkPolicy (only allow plugin pods to talk to core pod on specific ports). - Consider using Linux Security Modules (AppArmor/SELinux) profiles for containers to restrict syscalls, etc., if needed for high security environments. - Package scanning: incorporate an image scan in CI for both core and plugin images (looking for vulnerabilities). - Consider signing plugin images or manifests and verify signature at registration (ensuring plugin code hasn't been tampered with). This could be done via simple checksum or using something like Notary in Docker. Possibly an allow-list of approved plugins (as Faxbot had allowlist and checksums for plugin installation [15]). 4. **Admin Console UI Completion:** Build out the React Admin Console to be fully functional: - Implement login screen tied to Identity service (likely via an API call to /auth and storing a session or JWT). - Implement main dashboard showing system status (maybe number of active plugins, health status of core). - Plugins management UI: a page that lists plugins, allows enabling/disabling them (which calls a core API to maybe toggle a flag and possibly start/stop a plugin container if dynamic control is possible – though that might require orchestrator support; at least mark as disabled so it doesn't get traffic). - Configuration UI: screens to edit global config and plugin config. This can be schema-driven. For core config (like switching identity provider or email server), we create forms. For plugin config, since manifest likely describes config options, the UI can generate fields accordingly (similar to Faxbot's schema-driven settings [17] ). If a plugin manifest doesn't provide UI hints, we at least show a JSON editor or instruct editing .env for now. - User management UI: allow admin to create users, assign roles/traits. This calls identity service API. - Audit logs UI: as mentioned, a table view of audit records with filters by date or user. - Real-time monitoring: optionally, integrate some live status – for example, subscribe to an SSE or WebSocket (core can provide an SSE endpoint as indicated by Vivified PR18) to get events of plugin up/down or new events flowing, to display in UI. This is a nice-to-have if time permits. - **UX Polishing:** Ensure the UI is responsive (Faxbot's UI was mobile-responsive [26] , continue that), and that features show/hide correctly with traits (e.g. login as non-admin user, verify you don't see admin-only sections). 5. **Include AI/LLM Guardrails (if applicable):** If the platform advertises AI integration (MCP servers) as part of core, we might integrate the optional containers (Vivified could reuse Faxbot's approach: optional `faxbot-mcp` servers [27] ). While not central to plugin framework, it is an advertised feature, so: - Add an optional profile in docker-compose to run MCP servers (Node and Python LLM tool servers). - Ensure core's config can enable/disable these and that the Admin UI has a section for AI settings (like enabling AI features, or monitoring LLM usage). - Implement content filtering and rate limiting hooks in the core if an AI plugin is used. (This could leverage the policy engine: e.g. treat AI responses as needing certain traits). - This step can be low priority if time is short, since it's not explicitly asked in the plugin framework question, but good to keep in mind for completeness. 6. **Performance and Scalability Pass:** Do a round of testing for performance: - Simulate many events and ensure the event bus and core can handle them (tweak NATS/Redis config, maybe use horizontal scaling if needed). - Test with multiple instances of core (if stateless enough, we can scale core behind a load balancer in K8s; ensure things like the event subscription doesn't duplicate events incorrectly – might need a queue group in NATS for core if scaled). - Test plugin scale: e.g. two instances of Email plugin for load – does core route to both? (This might require service discovery logic if multiple instances per plugin ID). - These scaling concerns might be beyond initial scope, but we document any limitations (like "in v1, only one instance per plugin, scaling to be improved in future"). 7. **Deliverable:** Phase 4 produces a **production-hardened platform**. Deliverables include: - A fully functional Admin Console (you can manage users, plugins, and config through the GUI). - Comprehensive security controls in place (trait enforcement, network isolation, audit logs visible). - Compliance checklist updated (e.g. HIPAA alignment: we can demonstrate that audit logs, PHI handling, encryption etc. meet requirements). - Updated documentation: security architecture section, user guide for the admin UI, etc. - If possible, some basic performance metrics showing the system can handle, say, X events per second and Y concurrent plugin calls, with tuning guidelines.

## Phase 5: Developer Tools & Plugin Ecosystem Enablement

**Objective:** Finalize the developer support tooling (CLI, SDKs, templates) and create the reference plugins (Email Gateway and User Management) as full examples. This phase ensures that external developers can easily extend the platform and serves as a "reference implementation" proof.

**Tasks:** 1. **Finalize SDKs:** Complete the implementation of the Python, Node.js, and Go SDKs: - Provide clear README or docs within each SDK folder on how to use it. - Publish them to package managers if appropriate (e.g. PyPI, npm) for easy consumption. - Ensure they handle reconnection to event bus, exponential backoff, and edge cases gracefully (as production readiness). - Write unit tests for SDK components (simulate a fake core responding and ensure SDK correctly calls hooks). 2. **Vivified CLI Tool:** Develop the CLI (could be in Python or Node for convenience, or even just a set of cookiecutter templates if time is short, but ideally a CLI app). - Implement commands: `create-plugin` (with options for type & language) to generate a new plugin project scaffold. Use templates under `templates/` directory possibly. - `validate-manifest` command that runs the manifest validator on a given file. - Possibly `dev-start` to launch a test environment for a plugin (though this might be complex; at minimum, instruct how to use docker-compose for dev). - Make sure the CLI is easy to install (maybe `pip install vivified-cli` or a simple binary). 3. **Manifest Schema and Validator:** Finalize the JSON schema for plugin manifests. Include it in `docs/` and enforce it in core plugin manager. - Implement the validator logic (could use a JSON Schema library or custom checks). - Test the validator on the reference plugin manifests to ensure it catches errors and passes correct manifests. - Include traits validation: ensure unrecognized traits are warned (maintain a registry of known core traits). - Ensure the validator is integrated into CI (so if someone adds a plugin manifest in the repo, CI can auto-validate it). 4. **Reference Plugin: Email Gateway:** Implement the Email Gateway plugin fully: - Purpose: Listen for any canonical “Message” events (or a specific event like `SendEmailRequest` if defined) and send emails. Also expose an endpoint so that core or other plugins can directly request an email to be sent (e.g. via operator call). - For simplicity, implement sending via an SMTP library (config provided for SMTP server). Alternatively, integrate a third-party email API using the proxy feature (this could demonstrate proxy lane: e.g. sending via SendGrid API through core’s proxy). - Include logic to handle different types of content (maybe just text in this example). - Make sure to incorporate security: e.g. if an email contains sensitive info, maybe require that trait. - Provide a manifest with appropriate traits (like `'notifications'` trait, maybe `'external_api'` trait if it calls external). - Test it: have a dummy scenario where an admin triggers an email (maybe through Admin UI or a curl command to core that uses the plugin). - Use this plugin as a showcase in docs: explain how it works and why it’s structured that way. 5. **Reference Plugin: User Management:** Implement the User Management plugin: - Since core already has Identity, this plugin’s role might be to extend user profile info or manage additional user-related workflows. For example, it could maintain an “employee directory” with department info, etc., which is outside core’s basic auth user info. - It could expose endpoints like `/profile/{user}` to get extended profile, and listen to `UserCreated` events from core to auto-create a profile entry. This shows plugin reacting to core events. - It might also provide a UI extension: e.g. an extra admin console page “User Profiles” (to demonstrate custom UI, perhaps just a simple table of profiles). - It could depend on the Email plugin (if on user creation, it wants to send a welcome email via the Email plugin – demonstrating inter-plugin call). - Traits might include `'user_data'` or `'internal_plugin'`. - This plugin solidifies the concept of a domain logic plugin that is not just an integration but part of the application’s core domain (managing users). - Ensure it respects core’s identity (maybe it doesn’t handle credentials, just additional data). 6. **Documentation & Examples:** Update all documentation to include: - Tutorial: “Building Your First Plugin” which perhaps walks through creating a simple plugin using the CLI, similar to how vivified’s landing copy outlines it [23]. - Reference manual for plugin manifest options and core API endpoints. - Architecture diagram (we can include the one we adapted above). - Security best practices for plugin dev. - A section showcasing the two reference plugins – how they are implemented, with code snippets, to guide developers. - Deployment guide for production (covering Docker/K8s, scaling, backup of DB, etc.). 7. **Final Testing & Release:** Before declaring v1.0: - Conduct an **end-to-end test** simulating a small production scenario: deploy core and the two reference plugins on a cloud VM or k8s cluster, configure them (e.g. set up SMTP details), and run through key use cases (creating a user via UI, which triggers email, etc.). Monitor logs and fix any issues (memory leaks, etc.). - Perform a security review: maybe run some static analysis or even a penetration test on the running instance to ensure no obvious vulnerabilities (e.g. ensure no open ports except intended ones, test that a plugin can’t call DB directly, etc.). - Optimize any performance hotspots found (e.g. if our policy checks are slow, consider caching, etc.). - Prepare the repository for open-source release: ensure README and docs are clear, examples are cleaned up, and all badges (CI passing, license) are in place.

**Deliverable:** Phase 5 culminates in the **Vivified Platform v1.0** release. We will have: - Two fully working reference plugins (Email Gateway and User Management) serving as blueprints. - The CLI and SDKs enabling others to add more plugins. - Documentation and test results showing that the platform meets the goals (extensibility, security, etc.). - A sample Docker Compose that spins up the whole system (core + example plugins + any dependencies) so anyone can try it out with minimal effort (similar to the quick start in vivified docs [28] but now including our new plugins). - Possibly a demonstration (in docs or a blog) of another hypothetical plugin (like an Accounting plugin) to prove the concept extends to other domains.

With this delivered, Vivified will be a **production-grade modular framework** built on Faxbot's architecture but generalized to any domain. It will allow organizations to plug in modules for accounting, HR, notifications, etc., all under a unified, secure platform. Each phase built upon the previous, ensuring that at every step the system remained runnable and testable, leading to a robust final product.

## Conclusion

By following this execution plan, we scaffold Vivified into a secure, plugin-first platform that leverages the proven traits-first and canonical-model approach of Faxbot [1] while adding layers for generalization and security. The **3-lane communication model** ensures structured interactions: canonical events for interoperability, direct calls for precise operations, and a guarded proxy for the rest. The core platform shoulders all heavy duties (UI, identity, policy enforcement, logging), acting as a **security blanket around plugins** [4] so that each plugin can focus on its domain. With robust core services and a rich set of developer tools, Vivified will enable rapid development of new enterprise functionalities as plugins – from an Accounting module to an AI-powered analytics tool – without compromising the compliance and integrity of the system. This phased plan delivers incremental value, validating the architecture at each step, and culminating in a production-ready modular framework with two reference plugins demonstrating real-world usage. The result is a realistic, implementation-ready Vivified core that truly supports infinite extensibility through secure API-integrated plugins, fulfilling the vision of a **universal, security-first plugin platform** [29] [20].

**Sources:**

- Vivified Architecture & Landing Page [20] [21] [5] [2] [30] [23] [4] (design principles, plugin contracts, canonical models, trait enforcement, CLI tooling, security model)
- Faxbot Project Documentation [1] [17] [3] (traits-first UI, plugin manifest security, canonical event approach)
- Implementation notes from Vivified development (GitHub PRs and Issues) [14] (config service, plugin system evolution)

---

[1] [10] [12] [18] [22] [27] README.md
https://github.com/DMontgomery40/Faxbot/blob/7e0c15fa960d24b59c7581eeb1163b3d93439e32/README.md

[2] [4] [5] [6] [7] [9] [11] [13] [19] [20] [21] [23] [24] [25] [26] [28] [29] [30] vivified-landing-copy.md
https://github.com/DMontgomery40/vivi-site/blob/5ef2bb2d38aa293be239aa27ed5fc4d4fcf2d87f/vivified-landing-copy.md

[3] [8] [15] [16] [17] index.md
https://github.com/DMontgomery40/Faxbot/blob/7e0c15fa960d24b59c7581eeb1163b3d93439e32/docs/plugins/index.md

14 PR13: Hierarchical config DB + models
https://github.com/DMontgomery40/Faxbot/pull/14