ChatGPT

Modern Open-Source Developer Portal for External APIs

Introduction

An **API developer portal** is a central hub that enables external partners and public API consumers to discover, understand, and integrate with a company's APIs. It serves as a bridge between API providers and developers, offering everything from technical documentation and onboarding tools to access management and analytics [1] [2]. A well-designed developer portal not only showcases the APIs (and the services they enable) but also provides a seamless self-service experience. This improves developer adoption and satisfaction, and in many cases supports monetization of APIs [3] [4].

This report outlines the **most useful and modern features** for an open-source developer portal, focusing on external API consumers. It covers key features (including the given requirements), additional competitive features, a high-level architecture, and a breakdown of the solution into functional modules. All recommendations prioritize **developer experience**, modern best practices, and the open-source nature of the project.

Key Features and Requirements

Modern API portals come with a rich set of features to meet developer expectations. Below are the core features (including the specified requirements) that an open-source developer portal should have:

- **Admin UI for API & User Management:** An administrative interface to manage API products, users, and access control. Administrators can onboard new APIs (e.g. by uploading OpenAPI definitions), define plans or tiers, set rate limits/quotas, and manage user accounts. They should be able to approve or invite users, monitor API usage, and configure settings (like integrating Identity Providers or gateways). This admin console acts as the control center for the platform.

- **Developer Sign-Up & Account Management:** A consumer-facing portal where developers can self-register for access (or go through an invite-only onboarding flow for partner programs). The portal should support various **sign-up flows** – from open self-service sign-up to invite-only or approval-based registration – to accommodate different business models. Upon sign-up, users can create a profile, register their applications, and obtain API credentials (keys or tokens). **Single Sign-On (SSO)** integration is crucial for a smooth experience; in this case, using **Okta as the Identity Provider (IDP)** allows users to log in with corporate credentials and enables enterprise-grade security. The portal should handle the full OAuth/OpenID Connect flow with Okta, including user authentication and retrieval of profile/roles [5]. This ensures a **secure path for customers via Okta** and other IdPs (e.g. Auth0) for logging into the portal [5].

- **API Key and Token Management:** Once onboarded, developers need a way to obtain credentials to call the APIs. The portal should allow users to **create and manage API keys or OAuth client applications**. For example, a developer might register a new "application" in the portal, and the system provisions an API key or OAuth token that is tied to that app. Integration with the API gateway (Kong) is key here: when a user creates an API key, the portal can

communicate with **Kong's Admin API** to generate a credential (e.g. a key, JWT, or OAuth client) and associate it with a Kong consumer record. This way, the **Kong API Gateway** recognizes the key and enforces policies for that consumer [6]. The portal should also display the keys/tokens to the user, allow regeneration or revocation of keys, and show which APIs each key has access to. Support for both **API keys and OAuth tokens** (via Okta or Kong's OAuth2 plugin) would cover a wide range of use cases.

- **Rate Limiting and Access Plans:** To control usage, the portal must integrate with the gateway's **rate limiting and quota** features. Admins should be able to define **usage plans** or tiers – for instance, a free plan with 1000 calls/month and a premium plan with higher limits. These plans correspond to Kong rate-limiting or quota plugins configured per consumer or per API. The portal's admin UI would let you configure these limits and attach users or applications to specific plans. Consumers should be able to **view their current plan and usage**. If the portal supports monetization, plans might be tied to pricing (e.g. Free, Pro, Enterprise tiers). When a developer subscribes to a plan, the system can apply the appropriate rate-limit policy on Kong for that user or API key [7] [8]. This ensures **governance and quotas** are enforced consistently. Optionally, the portal can also support **invite-only plans or role-based access** (so that certain APIs or high-tier plans are visible only to invited partners).

- **Integration with Kong API Gateway:** Kong will act as the traffic manager enforcing auth, rate limiting, and logging. The developer portal should integrate with Kong in multiple ways:

- **API Publishing:** When admins add a new API to the portal, it could fetch or reference the corresponding **Kong service/route** configurations. In a tight integration, the portal could list APIs based on what's registered in Kong's Admin API. Alternatively, admins manually input API details, but either way, Kong is the runtime where those APIs are exposed.
- **Kong Plugins for Auth & Logging:** The portal should leverage Kong's plugins for authentication (e.g. key-auth, JWT, OIDC plugin for Okta) and logging. For example, Kong's OIDC plugin can be configured to use Okta for token validation if using OAuth2, or Kong's key-auth plugin can verify API keys. **Logging plugins** (or Kong's logging to a service) can feed into the portal's analytics. Essentially, Kong ensures each API call is checked for valid credentials and within rate limits, and it can log each call (to a file, database, or external service) for analytics. The portal acts as a front-end to configure these plugins (e.g., enabling the rate-limit plugin on a service and setting the allowed calls per user plan).

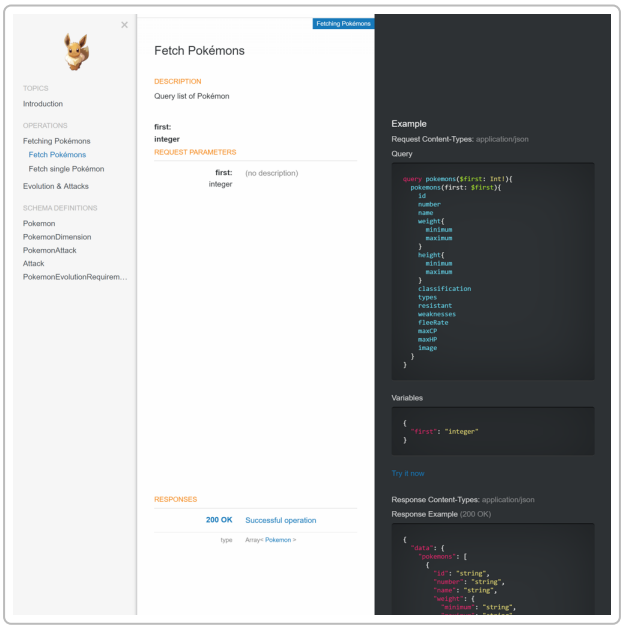
- **Automatic Key Provisioning:** As noted, the portal can **provision keys via Kong**. The Moesif open-source portal, for instance, includes a plugin that automates API key creation in Kong when a new user registers [6]. This means the signup process can generate a Kong consumer and key behind the scenes so that the user is ready to call APIs immediately. By integrating Kong's Admin API or using a provisioning plugin, the developer portal achieves smooth onboarding where everything is self-service.

- **Integration with Okta (Identity Provider):** Using **Okta** as the IDP brings enterprise-grade identity management. Okta can handle user authentication (login via SSO), and possibly authorization (using groups or roles to determine access). The portal should support OIDC or SAML integration with Okta [9] so that external developers (who might be employees of partner organizations) can use their Okta accounts to log in. Upon first login, the portal might provision a corresponding user profile internally or in a database, and possibly assign default roles or plans. Okta integration also means the portal doesn't need to store passwords – it will redirect to Okta's hosted login and trust Okta's tokens. In an advanced scenario, the portal could also map Okta groups to portal roles or product access (for example, an Okta group "PremiumAPI" could

automatically grant access to certain APIs or plans in the portal). This setup increases security and simplifies user management by leveraging a proven external IdP [5].

- **Rich API Documentation (REST and GraphQL):** A **beautiful, user-friendly documentation** section is the heart of the developer portal. The portal should display detailed reference docs for each API, including endpoints, parameters, and example requests/responses. For RESTful APIs, this is typically driven by **OpenAPI (Swagger) specifications**. Modern portals often integrate an interactive documentation tool like **Swagger UI, Redoc, or Stoplight Elements** to render OpenAPI docs in a clean web interface. For example, Stoplight's open-source Elements can generate a **three-column layout** with a nav menu, documentation content, and an example panel, all from an OpenAPI spec [10]. This kind of layout (famously used by Stripe's API docs) makes it easy to navigate and read the reference. The documentation should be not only comprehensive but also **interactive** – developers appreciate the ability to **try out API calls** from within the documentation (e.g. using an "Try it out" feature that makes test calls against a sandbox or the live API, with their API key).

For **GraphQL APIs**, the portal should provide specialized documentation since GraphQL schemas are different from REST. Ideally, the portal can integrate a GraphQL schema viewer or a **GraphQL playground**. One approach is offering an interactive IDE like **GraphiQL or GraphQL Playground** so developers can explore the schema and execute queries live. Additionally, static docs generated from the GraphQL schema can be provided. For instance, tools like DociQL can produce an **HTML documentation page from a GraphQL endpoint's introspection** with a Stripe-like three-column style [11] [12]. This includes query and type definitions, and even example queries with "Try it now" links [12]. In summary, the portal's documentation section should handle **REST and GraphQL elegantly**, presenting each API's documentation in a polished format (with syntax highlighting, search, example code in multiple languages, etc.). A **full-text search** across the docs is very useful, as seen in Gravitee's portal which offers search out-of-the-box [13].



Example of a modern API documentation layout (three-column design) generated from a GraphQL schema. This style provides a navigation sidebar, detailed documentation in the center, and an example panel (e.g. showing query examples and responses) for an interactive learning experience [11] [12] .

- **Developer Dashboard & API Console:** Beyond static documentation, a great portal often includes a **developer dashboard or console** where users can see and manage their usage. For example, upon logging in, a developer might land on a dashboard showing their active API keys, their usage statistics (calls made, errors, etc.), and any billing information. This console could also allow the user to test APIs with their credentials (like an embedded API tester). For GraphQL, an embedded GraphiQL console would let them execute queries against a demo endpoint. For REST, a "try it out" feature in docs or even a separate **API Explorer** can serve this need. The idea is to make the portal a one-stop place not just to read about the API, but to experiment with it in real time.

- **Built-in Analytics and Metrics: Usage analytics** are crucial for both the API provider and the consumers. The portal should track and display metrics such as the number of API calls made, bandwidth used, error rates, and latency. For developers, seeing their **API usage history** (daily/weekly/monthly calls, perhaps broken down by endpoint) helps them understand their consumption and debug issues. For the admin, analytics across all users and APIs provide insight into adoption and help in enforcing quotas. Modern portals integrate with analytics platforms or include dashboards for this purpose [13] [14] . In our design, Kong will generate logs for each API call – these could be fed into an analytics service. An open-source approach is to use **Moesif** (which provides API analytics and is mentioned alongside their portal solution) or other tools. Indeed, Moesif's open-source portal requires a Moesif account to feed metrics and logs [15] [6] . Alternatively, the portal could store metrics in a database (or use Cloudflare Workers' analytics if available, or push logs to a time-series DB). The key is to provide **visual dashboards** for API usage and allow developers to dig into logs. For example, showing a **log of recent API requests** made with their key (with timestamps and status codes) can greatly help in debugging integration issues. The portal can also implement **alerts** or **email notifications** – e.g. send an email when a user's usage approaches a quota (Moesif's portal supports sending automated emails based on usage [16]).

- **Monetization and Billing Features:** Many organizations monetize their APIs, so the portal should support this out-of-the-box. **Monetization features** include the ability to define paid plans, handle subscriptions, and process payments. The portal can present pricing details for different tiers (e.g. free vs. paid plans, or pay-as-you-go models) and allow developers to upgrade or downgrade their plan. Integration with a payment gateway like **Stripe** is a modern choice for handling billing. For instance, an open-source portal can use Stripe's API to charge for paid subscriptions or for usage overages. Moesif's developer portal integrates with Stripe to let customers **subscribe and pay for APIs, supporting different pricing models (tiered, per-unit, etc.) out of the box** [17] . We want similar functionality: a developer can enter payment info and purchase an API plan, and the system will automatically handle invoicing or charge for API calls. This could be via recurring subscriptions or usage-based billing. Additionally, the portal should enforce limits corresponding to the plan (so it ties into rate limiting) and could even offer **trial plans, promo codes, or prepaid credits** for flexibility [18] . All billing-related data (current plan, usage vs quota, cost) should be visible to the user in their dashboard. This transparency is key to a good developer experience in a monetized API product.

- **Excellent User Experience & Design:** Modern developers expect a **clean, intuitive UI and responsive design**. The portal's look-and-feel should be aligned with the company's branding (logo, color theme) and be as polished as a commercial product site [19] . In fact, the portal often

acts as a **showcase** for the API product, so investing in a user-friendly design is crucial [20]. This includes having a well-organized API catalog (with categories or product groupings for APIs), clear typography in documentation, and support for **dark mode** (many devs prefer reading docs in dark mode). The content should address both technical and non-technical audiences: for example, include high-level descriptions for product managers as well as detailed specs for developers [21] [4]. Providing **use-case guides** or "quick start" tutorials can be very helpful – short guides with code samples in various languages that show how to accomplish common tasks with the API [22]. Additionally, features like **full-text search**, **navigation breadcrumbs**, and **copy-to-clipboard buttons** for code snippets contribute to a top-notch UX. Because this project is open-source, a modular and theme-able frontend is ideal – enabling easy customization by others. In summary, the portal should be aesthetically pleasing, easy to navigate, and rich in helpful content.

- **Security & Privacy:** While not always visible to end-users, security features are a must. This includes **secure credential storage** (API keys/tokens should be stored hashed or in a secure vault if possible), enforcement of HTTPS, and proper access controls in the portal itself (admins vs developers, role-based access to certain features). Since Okta is used, a lot of the heavy lifting for authentication is offloaded, but the portal should still guard against things like exposing sensitive data. If there's an admin API for the portal, it should be protected (perhaps only accessible to admins with certain roles via Okta). Audit logging of administrative actions is another useful feature. Compliance considerations (GDPR, etc.) might also be relevant if the portal stores user data – being open-source, it's good to document how data is handled and allow deployment-specific configurations for privacy. **Rate limiting** not only prevents abuse but is also a security measure to mitigate DDoS by external keys. Kong will handle much of that at the API level, but the portal itself might implement some throttling for its own endpoints (e.g., to prevent spam sign-ups or excessive login attempts).

- **Community and Support Tools:** To make the portal more than just docs and keys, consider features that foster community and provide support. This could be as simple as an integrated **support contact form or chat widget**, links to community forums, or even a Q&A knowledge base. Some portals include a **FAQ section** or **troubleshooting guides** for common errors. Since this is aimed at external partners, having a way to get help quickly is important. While a full-fledged forum might be out of scope, linking to a GitHub discussions page or Slack channel for the API community can be beneficial. Another idea is providing a **changelog** or **release notes** page on the portal so developers can track what's new or changed in the APIs (some portals automate this based on API version changes). Modern portals (like Azure API Management's portal) are often content-managed, meaning admins can easily update marketing pages, publish announcements, or add Markdown articles for how-tos. Our portal could include a simple **CMS capability** for adding custom pages (or at least a Markdown-driven content section for guides).

The above features together create a **competitive and appealing developer portal** that meets today's standards. Table 1 summarizes some of these key features and the value they provide:

Feature/Module	Description & Value
Admin Interface	Manage APIs, users, roles, plans, and settings through a secure web UI. Enables governance and easy configuration for the API program.
User Onboarding & SSO	Self-service sign-up (with invite option) and Single Sign-On via Okta. Simplifies access for partners and secures authentication [5].

Feature/Module	Description & Value
API Key Management	Issue and revoke API keys or OAuth tokens. Developers can create multiple keys (e.g., for different apps) and view credentials. Keys are provisioned in Kong for enforcement [6].
API Catalog & Docs	Discoverable list of APIs with **beautiful documentation** (OpenAPI/Swagger and GraphQL). Includes interactive "try it" console, code samples, and search for a great DX [10] [12].
Plans & Rate Limiting	Define usage plans with rate limits/quotas. Enforce via Kong plugins, and display remaining quota to users. Supports free trials and tiered offerings [7].
Analytics Dashboard	Built-in analytics showing usage metrics, logs, and performance. Both developers and admins get insights (e.g., 360° API view for admins [13]). Optionally integrates with external analytics (Moesif, Grafana, etc.).
Monetization & Billing	Monetize APIs with paid plans. Integrate Stripe for payments so developers can upgrade plans and pay for usage seamlessly [17]. Automate billing and invoicing.
Gateway Integration	Deep integration with Kong API Gateway for policy enforcement: authentication, rate limiting, and logging at the gateway level. Portal automates config of Kong (via Admin API or plugins) to reflect portal settings.
Customization & Branding	Theming support to match company branding (logo, colors). Flexible layout (perhaps a CMS for pages). This makes the portal **an extension of the company's identity** and allows open-source users to adapt it [19] [20].
Support & Community	Support channels (contact forms, links to support docs, FAQ). Possibly a forum or chat integration. Helps developers get assistance and builds a community around the API.
Security & Compliance	Ensure secure handling of credentials (via Okta, vaults), RBAC for admin vs dev, and compliance with privacy laws. Logs and monitors access for security audits.

Table 1: Key features/modules of the developer portal and their benefits.

Additional Modern Features for Competitiveness

To stand out and truly delight developers, the portal can incorporate some additional modern features beyond the basics:

- **Multi-Organization Support:** If targeting partners who may have multiple developers, consider an **organization/team management** feature. This would allow a company (partner) to have an organization account on the portal with multiple user logins under it. Team members could share access to the same API subscription or keys. The admin could assign roles within an org (e.g., admin vs developer). This feature is seen in enterprise portals like Tyk's, which supports creating multiple organizations and teams to segment the developer audience [23]. It's useful for managing large partner accounts and is a differentiator in B2B API programs.

- **API Productization & Cataloguing:** Rather than just listing individual APIs, the portal can allow grouping APIs into **API products or collections**. For example, an API product might bundle several related endpoints (e.g., a "Weather API" product that includes current weather, forecast, and historical data endpoints). This aligns the portal with how the APIs deliver value (use-case based grouping) rather than just technical endpoints [23]. Each product can have its own overview page, documentation, and plan. This approach makes it easier for developers (and even non-developers) to grasp the offerings as products with clear use cases [4]. We can also allow **multiple catalogs** or views if needed – e.g., a public catalog vs a private catalog (some APIs only visible to certain users). This feature is more advanced but can be important for large API programs.

- **Dynamic Client Registration:** For OAuth-based APIs, implementing **Dynamic Client Registration** can streamline how developers get client credentials. Instead of manual setup, the portal can call Okta's (or another OAuth server's) dynamic registration endpoint to create an OAuth client on the fly when a developer signs up or creates an app. This automates issuing client IDs/secrets under the hood. Tyk's enterprise portal, for instance, mentions dynamic client registration with popular IdPs [24]. In our case, using Okta's API for this would allow an OAuth client to be created per developer app, which Okta can then use to issue tokens. This is a more complex integration but provides a slick experience for obtaining OAuth2 credentials seamlessly.

- **Advanced Documentation Features:** Modern documentation can include more than reference guides. We could incorporate **tutorials, how-to guides, and code playgrounds**. For instance, an embedded **Postman Collection** (or a "Run in Postman" button) allows developers to quickly test APIs in Postman. Another idea is to provide **SDK downloads or code generation** for various languages – given an OpenAPI spec, tools can generate client SDKs (some portals provide SDKs in popular languages to jump-start integration). At minimum, providing **copyable code examples** in multiple languages (cURL, Python, JavaScript, etc.) for each endpoint is considered best practice. There are libraries and open docs tools that can render such multi-language code tabs. These features cater to developers by reducing friction in implementing the API.

- **API Mock/Sandbox Environment:** To encourage exploration, the portal might offer a **sandbox environment** or mock server for the APIs. This could be as simple as an option to use a mock server (possibly generated from the OpenAPI spec) for trying out calls without affecting production data. For GraphQL, a public read-only sandbox endpoint could be provided. This feature is particularly useful for public APIs where you want to let developers play around before they have full credentials. Some API providers use mocking tools (like Prism by Stoplight or Sandbox from Stripe) to achieve this. Including guidance or tools for mocking can enhance the developer onboarding experience.

- **Automated Testing & Monitoring Tools:** As an extra, the portal could integrate with API testing or monitoring services. For example, allow developers to define test calls or use something like **API monitoring** (perhaps via the gateway's capabilities or an external service) to ensure their usage is working. This might be out of scope for an MVP, but mentioning it shows forward-thinking – some platforms let developers set up monitors for their API endpoints (to get alerts if the API is down, etc., possibly leveraging the provider's infrastructure).

- **AI-Assisted Features:** A cutting-edge idea is to incorporate AI to help developers with the APIs. For example, an **AI chatbot or assistant** on the portal could answer questions about the API or even generate example code. This is not a standard feature yet, but with large language models, a portal could allow users to ask natural language questions like "How do I create a new invoice with the API?" and get guided answers or code snippets. While implementing this is non-trivial,

it's worth noting as a modern trend for developer tools. At the very least, using AI to power advanced search (e.g., semantic search in the docs) could improve the experience.

- **Search and Navigation Enhancements:** In addition to basic full-text search, consider features like **filters** (filter APIs by category, version, or tag), and an **autocomplete search bar** that can quickly find APIs or docs as you type. If the portal hosts multiple API versions, providing version switching in the docs is important (e.g., a dropdown to view v1 vs v2 of an API). Also, ensure the portal is **mobile-responsive** – while most developers will use it on desktop, a responsive design is part of modern expectations.

- **Open-Source Contribution Readiness:** Because this portal is open-source, design choices should facilitate community contributions. This includes clear documentation for developers who want to extend the portal, a modular architecture, and plugin-friendly design. The Moesif portal, for instance, has a plugin architecture allowing custom key provisioning logic [6]. We can emulate that by making components like the gateway integration or analytics **swappable modules**. This way, if someone wants to use a different gateway or add a feature, they can do so without forking the entire project. Additionally, using TypeScript and clear interfaces will make it easier for contributors to understand the code. Emphasizing tests and following standard open-source practices (like linters, CI, and contribution guidelines) will also make the project attractive on GitHub. Modern developer expectations extend to how the project itself is presented to contributors – a good README, and an example deployment (possibly a one-click deploy to Cloudflare Workers or a Docker setup for local testing) would be beneficial.

In summary, these additional features – from team accounts to AI assistants – can increase the portal's appeal. Project scope will determine how many of these make it into the initial release, but designing the architecture to allow adding them later is wise.

High-Level Architecture Design

Building the portal with **TypeScript on Hono (Bun)** and targeting **Cloudflare Workers** deployment implies a serverless, edge-optimized architecture. Hono is a **fast, lightweight web framework** that can run on Cloudflare Workers (as well as Node, Deno, Bun, etc.) [25]. This choice means our backend will be highly scalable and globally distributed, which is great for an API portal accessed by developers around the world.

At a high level, the architecture will have the following components:

- **Cloudflare Workers Backend (Hono):** This is the core application server for the portal, written in TypeScript using the Hono framework. It will provide the APIs and server-side logic for both the consumer-facing portal and the admin UI. Running on Cloudflare Workers means it executes at the edge, providing low latency to users and easy scaling. The backend will handle routes for things like user signup/login (redirecting to Okta as needed), serving the API documentation data, managing user API key requests, etc. Hono's router will map these routes efficiently, and because it adheres to Web Standards, the same code could potentially run in other environments if needed [26].

- **Frontend (React/Vue or Static Pages):** We have a decision to make for the frontend. We could build a single-page application (SPA) in a framework like React or Vue that communicates with the Hono backend via JSON APIs. Alternatively, we could leverage Hono to serve SSR (server-side rendered) pages or static content. Given Cloudflare Workers' constraints (they can serve static

HTML/CSS/JS and also perform SSR for moderate content sizes), one approach is to use a lightweight frontend that is embedded into the Worker (perhaps via Hono's templating or returning HTML responses). However, a more flexible approach is to separate the frontend as a static site (could be deployed on Cloudflare Pages or the same Worker) and use the Hono backend as a pure API (for login, data, etc.). The **API documentation** could be a static React app (for example, using Stoplight Elements or Redoc as a component) that fetches the OpenAPI spec from the backend. The GraphQL docs might similarly load the schema via an API call or have it pre-bundled. For simplicity, we might start with server-rendered pages for core features (Hono can return an HTML page with docs embedded), and later enhance with client-side interactivity. Cloudflare Workers can also serve static assets from a KV store or from the bundle.

- **Database/Storage:** Since Cloudflare Workers are stateless by nature, we will need storage for persistent data. Options include **Cloudflare KV (key-value store)** for simple data, **Cloudflare D1 (SQLite database)** for relational data, or Durable Objects for more complex stateful logic. Likely, we'll use Cloudflare D1 as it provides SQL capabilities on the edge (useful for querying users, plans, etc.). Data that we need to store includes: developer profiles (though much of that is in Okta, we may store a user record linking to Okta ID and containing their roles or preferences), API definitions (the OpenAPI specs or at least references to them, unless we just fetch from a URL), application records (linking a user to their API keys and chosen plan), analytics data (if storing raw logs or aggregated metrics ourselves), and billing records (if we track subscriptions locally in addition to Stripe). **Stripe** transactions might mostly live in Stripe's system, but the portal might store subscription status or customer IDs to know what plan a user is on. Cloudflare D1 can handle moderate volumes of such data. For caching documentation or spec files, Cloudflare KV could be used (e.g., cache the OpenAPI JSON under a key for quick retrieval, or cache frequently accessed content like the docs HTML).

- **Integration Points:** The portal backend will integrate with several external services via REST APIs or SDKs:

- **Okta API/SDK:** for authenticating users. The typical flow is that the frontend will redirect to Okta for login (using OAuth2 authorization code flow with PKCE). Okta will then redirect back with a code which the portal backend exchanges for an ID token/Access token. The portal verifies the ID token (or uses Okta's introspection) to authenticate the user. We might use an existing library or Cloudflare Worker middleware for OIDC. Post-login, the portal might create a local session (perhaps a JWT stored in a secure cookie) so that subsequent requests identify the user. Alternatively, since it's all front-end/back-end in one domain (potentially), we might just use the Access token in Authorization headers. But storing a simple session cookie signed by the Worker could be easiest. The portal will also use Okta's API to get user info (if needed) and potentially to manage groups or dynamic client registration as discussed. Okta becomes the source of truth for user identity.
- **Kong Admin API:** for creating and managing consumer entities and credentials. The portal needs to be configured with the address of the Kong Admin endpoint (Kong could be running elsewhere, maybe within the company's infra or in Konnect cloud). When a developer requests a new API key, the portal will send a request to Kong (e.g., POST /consumers, then POST /consumers/{user}/key-auth) to generate the key, or use a pre-created consumer. Similarly, if a user is removed or their access revoked, the portal may call Kong to delete the credentials. The portal could also fetch data from Kong, like which APIs (Services/Routes) are available, and possibly which Kong plugins are enabled for them (to display in the portal). However, syncing everything with Kong might be complex, so a simpler approach is that the portal has its own catalog of APIs and only uses Kong for runtime enforcement and key gen. We must also consider **Kong Dev Portal vs our portal**: Kong Enterprise has its own portal, but here we are replacing

that with our custom solution. So Kong in our architecture is headless – just the gateway and its admin API controlled by our portal.

- **Stripe API:** for billing. The portal will use Stripe to handle payments and subscriptions. Likely we'll integrate Stripe Checkout or Billing. For example, when a user chooses a paid plan, the portal might redirect them to a Stripe Checkout page to enter payment details, then Stripe calls a webhook (which our portal can handle via a special Worker route) to confirm the subscription. The portal's database updates the user's plan accordingly. We can also use Stripe's API to fetch usage or to charge for overages if using usage-based billing. Since Cloudflare Workers can call external APIs, this integration is feasible (taking care to secure webhooks and API keys). Moesif's portal uses Stripe as the **only supported billing provider** and suggests that alternate flows can be custom-coded [27]. We'll likely follow a similar model: assume Stripe initially, with the ability for others to extend it.
- **Moesif or Analytics Service:** While optional, if we want to leverage Moesif for analytics, the portal would send data to Moesif's APIs. For example, Kong can be configured with the Moesif plugin to automatically send API call data to Moesif, which then the portal can retrieve via Moesif API to display analytics. Alternatively, the portal itself could call Moesif's API to get aggregated stats for a user. Since Moesif offers a rich set of analytics out-of-the-box (like user cohorts, email notifications, etc.), an open-source portal can integrate it to avoid reinventing the wheel for analytics [16]. However, as an open-source project, we might allow plugging in other analytics backends (perhaps a simple internal one using Cloudflare's logs, or integration with an open source stack like Prometheus).

- **Email/Notification Service:** To send out invite emails or usage alerts, the portal might use a service like SendGrid or Mailgun, or even Cloudflare Workers Email (if any). This is not core, but worth planning for. For instance, for an invite-only portal, when an admin invites a new user, an email with a registration link should be sent. Similarly, password resets (if not fully on Okta, though Okta can handle user password resets if it manages credentials) or general announcements might be emailed. As a modern feature, integrating a transactional email service ensures good deliverability of those emails.

- **Architecture Diagram:** The system can be visualized as follows (described conceptually): The **frontend** (in browser) interacts with the Cloudflare Worker (Hono backend) via HTTPS. The Worker, depending on the request, may interact with **Okta** (for auth flows), **Kong** (for API key creation and enforcement, though actual API traffic goes directly from the developer's app to Kong), **Stripe** (for payments), and the **Database** (Cloudflare D1 or KV) for reading/writing portal data. Meanwhile, API traffic flows from the developer's application to Kong Gateway, which uses its plugins to authenticate (checking keys via its datastore which was populated by the portal) and log the request. Logs/metrics might be pushed to **Moesif/Analytics**. The portal can pull metrics from that analytics store to display to the user. In effect, the portal is the **control plane** (managing users, keys, plans, and providing docs), and Kong is the **data plane** (enforcing at runtime). This separation ensures that at high traffic, Kong handles it, while the portal is only handling the out-of-band management traffic.

- **Performance and Scalability:** Using Cloudflare Workers means the portal's backend is serverless and scales automatically at the edge. Hono's efficiency ensures low overhead for handling requests. This setup can handle a large number of concurrent users browsing docs or creating keys without dedicated servers. For the data storage, Cloudflare D1 is in beta but should scale for typical usage (and we can expect thousands of users or keys to be fine). If extremely high scale is needed for analytics, an external specialized store is better (hence Moesif or others). We should also be mindful of **rate limiting the portal's own APIs** – e.g. prevent abuse of the

sign-up or key generation endpoints. Cloudflare can help here (there's built-in Worker KV based rate-limit patterns, or simply rely on Cloudflare's network-level protections).

- **Security and CORS:** The portal backend will likely serve its content from a domain (say `developers.example.com`) and needs to interact with the API gateway domain (`api.example.com` or so). If the documentation "Try it out" feature calls the actual APIs, we must handle Cross-Origin Resource Sharing (CORS). E.g., enabling CORS on the API Gateway for requests coming from the portal domain (this can be a policy in Kong or configured directly). Alternatively, the portal could proxy those calls through the Worker (so the calls appear same-origin to the browser). Both approaches are possible. Using the Worker as a proxy might simplify CORS but adds a slight overhead; however, Cloudflare's network is fast and it might be negligible.

- **Logging and Monitoring:** We will implement logging in the Worker for portal actions (like user sign-ups, errors, etc.), possibly storing logs in a centralized log service (or even Cloudflare's own logging). Cloudflare provides an environment to capture exceptions and send to a service like Sentry for error monitoring of the portal. For Kong, we rely on its logging (which might go to stdout if self-hosted, or to a file/ELK, etc.). The portal might query some logs via Kong's Admin API (Kong can provide some metrics via its Admin or Prometheus plugin). But more likely, we either integrate something like **Kong + Prometheus + Grafana** for open-source metrics or simply use Moesif as mentioned.

In essence, the architecture is **modular**: the Cloudflare Worker (Hono) is the central orchestrator connecting IDP (Okta), API Gateway (Kong), Payment (Stripe), and Data/Analytics stores. This modular approach aligns well with open-source, as each integration can be a module that could be swapped (for example, using Tyk instead of Kong, Auth0 instead of Okta, etc., with minimal changes, if the code is structured abstractly). The diagram below conceptually illustrates these components and interactions (requests from developers vs admin, flows to external services, etc.):

(Imagine an architecture diagram here, showing: **User** -> **Developer Portal (Cloudflare Worker)** -> various services (Okta, Kong, Stripe, DB, Analytics) and how **API calls** go from User's App -> Kong -> (if approved) upstream API. *Since embedding an actual diagram isn't possible in text, we describe it.)*

This high-level design ensures that the portal is **cloud-native, serverless, and globally accessible**, while integrating all required systems to provide a seamless experience.

Modular Functional Breakdown

To organize development and facilitate open-source contributions, the portal can be broken down into **functional modules or packages**. Each module encapsulates a set of related features and could be developed somewhat independently. This modular breakdown also helps when converting into tasks or user stories for implementation. Below are the major modules with their key responsibilities:

1. **Authentication & User Management Module** – Handles everything related to user identity and access.
2. **Okta Integration:** Implement the OAuth2/OIDC flow for login via Okta, including redirect endpoints and token verification. Ensure the portal trusts Okta JWTs and maps them to portal user sessions.
3. **User Database:** Maintain a minimal user profile store (if needed) linking Okta user IDs to portal roles or metadata (e.g., what plan they have). Possibly store invite codes and their statuses for invite-only flows.

4. **Registration & Onboarding:** If self-signup is allowed (non-SSO scenario), handle user registration forms, email verification, and password management (though with Okta, user credentials are managed by Okta – so this may defer to Okta's user registration flow or an Okta hosted sign-up widget if allowed).
5. **Invite System:** Provide admin the ability to invite new users (generate invite links or emails). Enforce that only invited emails can register if the portal is in invite-only mode.
6. **Roles & Permissions:** Define roles (e.g., Admin, Developer, maybe read-only auditor) and enforce authorization in the portal (e.g., only Admin role can access the admin UI routes). Okta groups or app metadata might be used to manage roles.

7. **Profile Management:** Allow users to view and edit some profile info (maybe their name, email – though if Okta is master, editing might happen in Okta). Possibly allow password reset (delegated to Okta's flow).

8. **API Catalog & Documentation Module** – Manages API listing and documentation content.

9. **API Catalog Management:** Data structures and admin interfaces to add or update API entries. Each API might include fields like name, description, category, OpenAPI spec URL or content, GraphQL endpoint URL (for introspection), version info, and which plan(s) it's available under. Admins can create/edit these.
10. **Documentation Rendering:** Utilize documentation tooling (Swagger UI, Redoc, Stoplight Elements, etc.) to display REST API docs. Integrate a GraphQL doc viewer or embed GraphiQL for GraphQL APIs. This module could be responsible for serving the documentation pages. Possibly it pre-processes OpenAPI files (e.g., to add auth examples with the user's API key).
11. **Content Pages & Guides:** Provide a way to create markdown pages for guides, tutorials, and changelogs. This could be a simple folder of markdown files rendered by the portal or a lightweight CMS interface. It should include a **Quick Start** guide for each API or use-case, with code samples.
12. **Search Functionality:** Implement search over API docs and guide content. This could be done via a indexed search library in the browser or a simple search API in the backend. Ensuring that developers can quickly find endpoints or topics by keyword is the goal.
13. **Versioning:** Handle multiple versions of API docs. This might involve storing multiple OpenAPI specs and providing a UI toggle for versions. It's important for APIs that evolve (e.g., v1, v2).

14. **API Console (Try It Out):** Provide interactive docs. For REST, possibly use Swagger UI's "try it out" or implement fetch calls from the browser (with CORS or proxy as needed) to let users test endpoints with their key. For GraphQL, embed GraphiQL pointing to either the real endpoint or a sandbox. Ensure that these calls are safe (maybe disabled for destructive operations or clearly labeled as test).

15. **API Key Management & Gateway Integration Module** – Responsible for linking the portal with Kong and managing credentials.

16. **Kong Client/Plugin:** Abstract the interactions with Kong's Admin API. For example, have functions to create a consumer (with username = user's email or ID), create credentials (key-auth or JWT secret, etc.), assign the consumer to appropriate groups or ACLs if using ACL plugin for plan enforcement. If using Kong's OAuth2 plugin, possibly register an OAuth client. This module handles those API calls and errors (e.g., if Kong is down or returns conflict if consumer exists).
17. **Key Lifecycle:** Provide the UI and API for developers to create new API keys, list existing keys, and revoke keys. Under the hood, this calls Kong. Store any needed info in the portal DB (maybe not strictly necessary if we can fetch from Kong, but caching is good for performance). For

example, if Kong issues an API key value, store it hashed (for display maybe show only once like some sensitive tokens are shown only on create).

18. **Rate Limit & Quota Enforcement:** When a user is assigned a plan, configure Kong's rate-limiting plugin for that user or key. One way is to use Kong's consumer-level rate limiting (via plugin config per consumer). Alternatively, manage it via Kong's "ACL and Plugin per group" approach: define Kong rate-limit plugins on combinations of API and plan, then add consumers to a group that corresponds to the plan. This module would encapsulate that logic. The portal DB should also know the user's plan and limits.
19. **API Access Control:** If some APIs are restricted (e.g., only certain users can see/call them), Kong can enforce with ACL plugin or by issuing different credentials for different APIs. This module might need to tag consumers with which APIs they're allowed or implement a check. However, this can be complex; many API programs simply expose all public APIs to all keys of certain plan, etc. For fine-grained control, the portal could integrate with Kong's ACL plugin (create groups for each API or API bundle).

20. **Logging integration:** Configure Kong to send logs to an endpoint or service that the portal can read. Possibly set up Kong's **Logging plugin** (e.g., HTTP log plugin pointing to a Worker endpoint or directly to Moesif). If logs come to the Worker, we can process or store them (though that could be a lot of data, so maybe better to go to Moesif or a storage bucket). In any case, this module defines how logging and metrics get from Kong to the portal/analytics. It might also contain logic to query Kong's Admin API for some statistics if needed (Kong has some /metrics if Prometheus is enabled, etc.).

21. **Analytics & Monitoring Module** – Focused on collecting, storing, and displaying usage data.

22. **Data Collection:** Set up integration to collect API usage. As discussed, options include using Moesif (which requires simply configuring their SDK or plugin), or rolling our own collection. If rolling our own: possibly a Cloudflare Durable Object or Queue that receives log entries (from Kong or from Cloudflare's own access if portal proxies requests) and stores them in D1 or a time-series DB. For now, assume using Moesif or a similar service for ease [15].
23. **Usage Database:** If not using an external service, design tables to store usage records (API call logs: who, which endpoint, timestamp, response time, status). This can be large, so maybe store aggregated counts per day per user per API to limit size, or use an analytics service.
24. **Analytics Dashboard UI:** Implement front-end components (charts, tables) to display metrics. For example, show a chart of API calls over time, a pie chart of calls by endpoint, a list of recent error responses, etc. Use libraries (maybe Chart.js or D3) to render graphs. Provide filtering (e.g., view by specific API or specific time range). For the admin, allow viewing aggregate usage across all users and identifying top users or potential abuse.
25. **Alerts & Notifications:** Optionally, allow users to set up alerts (like "email me if I exceed 80% of my quota"). This ties into both the usage tracking (to trigger the alert) and an email service. For simplicity, we might send automatic emails for certain events (e.g., when a user hits 100% of quota, or when a new user signs up – the latter for admin notification). Moesif's portal mentions "Behavioral emails" which likely refers to automated emails based on usage thresholds [16]. We can start with fixed notifications for key events.

26. **Service Monitoring:** While mostly the provider's concern, the portal might show API status (like uptime or current status from a status page). Possibly integrate with a status page API (if the company has one) to display "All systems operational" or incidents. This builds trust with developers.

27. **Billing & Monetization Module** – Handles payment processing and plan management.

28. **Plan Definitions:** Create data structures for plans (name, price, included quota, overage policy, etc.). Provide admin UI to manage these plans. Some plans might be free, some paid, some internal (for special partners). Plans tie in with both Kong (for enforcement) and Stripe (for payment).
29. **Stripe Integration:** Use Stripe's API to manage customer accounts and subscriptions. Likely, when a user chooses a paid plan, create a Stripe Customer for them (if not existing), and create a Subscription to the plan's product. We need to have set up products/prices in Stripe beforehand (or the portal can use Stripe API to create them on the fly based on plan definitions). Implement webhook handlers to catch events like successful payment, subscription canceled, payment failure, etc., to update the portal state.
30. **Billing UI:** In the developer's account section, show their current plan, next payment date or remaining trial time, usage vs quota (if usage-based billing). If using Stripe's hosted billing portal, provide a link for them to update payment info or cancel. Alternatively, implement simple forms to handle upgrades/downgrades (which behind scenes call Stripe).
31. **Usage Metering for Billing:** If plans have overage charges (e.g., $X per 1000 extra calls), the module should calculate usage beyond quota and report it to Stripe for billing. Stripe's usage-based pricing can be used by reporting the usage via their API. If not using Stripe's automated usage, we might invoice manually (but ideally automate through Stripe's metered billing).
32. **Free Trial / Credit Handling:** Allow new users to have a free trial or some credits. Perhaps generate Stripe coupons or just handle it in logic (like don't bill first month). Manage one-time credits if needed (if admin wants to grant extra quota or free calls).

33. **Monetization Toggle:** Some open-source users of this project might not want any billing (e.g., if they provide free APIs). So design the module to be optional – e.g., if no Stripe API key is configured, the portal runs in "free mode" with no payment, possibly hiding pricing info. This would widen adoption of the open-source project.

34. **Admin Dashboard Module** – A separate front-end (and related backend routes) for administrators to manage the system. (This could be part of each above module's admin aspects, but it's worth grouping.)

35. **Admin UI Front-end:** Could be a section of the portal app that is only accessible by admin users (after login, check role). It should allow toggling between the consumer view and admin view if the same user has both access. Alternatively, it could be a separate path like `/admin` that loads an admin SPA or interface.
36. **User Management UI:** List all users, search by email, view their details (keys, usage, plan). Admins should be able to modify user attributes: e.g., assign a different plan, deactivate a user (which might revoke keys), or impersonate a user to see what they see (for support). Also, manage invites: see pending invites, send new invites, revoke invites.
37. **API Management UI:** Interface to add/edit APIs. Possibly integrate an OpenAPI editor or at least allow uploading an OpenAPI file or providing a URL. If the API is GraphQL, perhaps allow uploading a GraphQL schema file or introspecting a live endpoint once to get the schema. Admin can mark APIs as published/unpublished (maybe to draft docs before releasing).
38. **Plan & Billing Management:** UI to create or edit plans (for internal use; public developers likely only choose, not create plans). If we integrate deeply with Stripe, some of this might just reflect Stripe data. But at least, define quotas and link to price IDs in Stripe. Also, show summary of current subscriptions (how many users on each plan, revenue maybe). For a small open-source project this might be overkill, but in a comprehensive portal, these insights are helpful for the API product manager.
39. **Content Management:** If the portal has custom pages (guides, FAQ, etc.), provide a simple way to update them. This could be as simple as an admin Markdown editor or an upload mechanism.

Or instruct admins to update files in the repo (less friendly). Because we target open-source maintainers too, maybe content can be managed via Git (e.g., docs pulled from a GitHub repo). However, an in-portal editor is more user-friendly for a non-technical API manager to quickly update text.

40. **Analytics & Logs for Admin:** Beyond what developers see, admins might need advanced analytics – e.g., filter by user, see global trends. Also, view system logs: login attempts, errors, etc. This helps in support and monitoring.

41. **Settings:** A section for configuration – e.g., managing the Okta configuration (client IDs, etc.), Kong connection info, Stripe keys, email server, feature toggles (like enable/disable monetization or invite-only mode). These settings likely live in environment variables or config files in deployment, but exposing some of them in UI (with proper protection) could ease operations. Possibly for security, we only allow viewing some settings, or none in UI and keep them as code config.

42. **Core Infrastructure Module** – Underlying tools and frameworks that support the above modules. (This is more of an internal grouping for developers.)

43. **Hono Server Setup:** Define the Hono app, global middlewares (for parsing JSON, handling sessions, etc.), and routes mounting. For example, routes under `/api/*` for the JSON APIs (used by the front-end or admin SPA calls), and maybe routes for server-rendered pages (if doing SSR).
44. **Middleware:** Common middleware like an **AuthGuard** that checks if a request has a valid session (and optionally correct role) and redirects to login if not. Another middleware for **CSRF protection** if we have forms, or for logging requests. Also, CORS handling if needed (for API routes if they might be called from other origins).
45. **Session Management:** Implement session handling, perhaps using a secure cookie. Cloudflare Workers don't have a filesystem, but we can use signed cookies (where the cookie contains user info in JWT form, encrypted or signed). Or store sessions in KV. Using JWT (with Hono or a library) might be simplest – essentially stateless sessions. Okta's token can be part of it. Ensure to secure it (HttpOnly, Secure cookies, etc.).
46. **Error Handling:** A centralized error handler to catch exceptions and return friendly messages or error pages. Also logging errors to a monitoring service (could integrate Sentry's JS SDK for Workers, for example).
47. **Testing & CI:** Set up the project structure to allow unit tests for each module (e.g., test the Kong integration module with mock responses, test the OpenAPI documentation parser, etc.). Also, provide examples or scripts to run the portal locally (like a dev server using Miniflare for Cloudflare Worker simulation).
48. **Deployment Scripts:** Configuration for deploying to Cloudflare (Wrangler config), including environment variables for secrets (Okta client secret, Stripe secret, etc.). Make it easy for users to fork and deploy their own instance.

Each of these modules can be further broken into user stories. For example, under *API Key Management*, a user story might be *"As a developer, I want to create a new API key for my application so that I can authenticate API calls separately from my other apps."* Under *Analytics*, a story could be *"As a developer, I want to see a graph of my API usage over time so I can track my consumption against my quota."* By structuring the project into modules, contributors can focus on areas of interest (e.g., someone could contribute a new "Github OAuth integration module" to replace Okta, or a new theme for the documentation module).

Examples and References

To ensure we build a state-of-the-art portal, we can look at successful projects and products with similar goals:

- **Moesif Open-Source Developer Portal:** Moesif, an API analytics company, has open-sourced a developer portal that closely aligns with our needs. It supports Okta/Auth0 for identity, Stripe for payments, and provides plugins for Kong, Tyk, and other gateways to provision API keys [28] [27]. Notably, it focuses on API monetization and analytics. For example, it allows **self-service subscription to API plans with Stripe** and has out-of-the-box support for **tiered and usage-based pricing models** [17]. It also integrates with Moesif's analytics to show usage metrics and even send automated usage emails [16]. This project is a great reference for how to architect the integration between IdP, gateway, and billing in an open-source way. We can draw inspiration from its plugin-based approach (the ability to swap in different key provisioning mechanisms) [6] and its emphasis on developer experience. Since it was released in late 2024, it's fairly modern and designed for production use.

- **Kong Enterprise Developer Portal:** Kong's Enterprise offering includes a developer portal which, while not open-source, exemplifies many best practices. It provides a **single source of truth for developers to discover and consume services** [29]. It supports SSO (as evidenced by guides on integrating Okta SSO with Kong's dev portal [9]) and has features like documentation, try-it-out, and consumer management. Key lessons from Kong's portal are the tight integration with the Kong Gateway (since it's the same product family) and the emphasis on customization and branding. Kong's portal is highly customizable (you can inject custom CSS/JS, use your own domains, etc.) – our portal should offer a similar level of flexibility in theming. Also, Kong's portal uses the concept of **Workspaces** to segregate APIs for different audiences – for our design, a simpler approach of categories or orgs should suffice, but it's worth noting for large-scale uses.

- **Tyk API Management (OSS & Enterprise):** Tyk is an open-source API gateway and has a developer portal component. The **Tyk Classic (OSS) Portal** was relatively basic (CMS-like with key request flows), but the **Enterprise Portal** introduced more advanced features: **custom pages, API products, multiple teams/orgs, multiple API catalogs, IdP integration, and full control of developer signup flow** [19]. This aligns with our list of advanced features. Tyk's portal is a benchmark for flexibility, as it allows grouping APIs into products and tailoring access for different audiences. It even supports dynamic client registration with popular IdPs [24], similar to what we envision with Okta. Studying Tyk's approach can guide our implementations of multi-organization support and product-based catalog.

- **Gravitee API Platform (OSS):** Gravitee is a fully open-source API management solution. Its **developer portal** component offers **custom themes, full-text search, and integrated API documentation**, along with an **analytics dashboard** [13]. It also has built-in support for **application registration and dynamic client linking between its API Management and Access Management components** [30]. Gravitee's example shows that even in open source, you can achieve a polished portal with search and analytics. It also underscores the importance of the **API lifecycle** – their portal ties into API versioning and an approval workflow for subscriptions. For instance, Gravitee allows defining plans and developers must subscribe to a plan (which the admin can approve). We might not implement an approval workflow initially (we assume auto-approval/self-service), but it's something to keep in mind for enterprise use (some APIs might want manual approval for access).

- **Stoplight and Other Documentation Tools:** While not full portals, tools like Stoplight Elements [31], Redoc, Swagger UI, and GraphQL Playground serve as building blocks for our documentation needs. They demonstrate how to create a **delightful documentation experience** with interactive features and attractive layouts. Stoplight Elements specifically is open-source and used by companies like Spotify for developer docs [32]. By leveraging such tools instead of writing our own from scratch, we save time and ensure the docs meet developers' expectations. Another tool example is **DociQL** (for GraphQL) which produces docs in the style of Stripe's, showing that even GraphQL can have first-class docs outside of an IDE [11] [12].

- **Community Portals & Best Practices:** There are many blogs and guides on what makes a great developer portal (such as the blobr.io blog [3] [33] and Moesif's guide [2] [34]). Common themes include having **clear use-case driven content, intuitive design, comprehensive documentation, easy onboarding (keys & auth), and clear monetization models**. We have incorporated these principles. For example, ensuring the portal addresses not just developers but also provides value proposition clarity for less technical stakeholders [21], offering quick starts and use-case guides, and being transparent about how to use and pay for the APIs.

In conclusion, our open-source developer portal design brings together **admin control**, **developer self-service**, **seamless Okta and Kong integration**, and **rich documentation** in a modern, TypeScript-based stack. By learning from successful portals like Moesif's and Tyk's and using modern frameworks like Hono on Cloudflare Workers, we can deliver a solution that is not only feature-complete and performant, but also attractive for the open-source community to adopt and contribute to. This portal will empower API providers to productize and monetize their APIs while delighting external developers with a world-class experience [35] [17]. The combination of features and architecture described will ensure that external partners and public API consumers have everything they need – from intuitive docs to secure access and insightful analytics – to integrate with confidence and speed.

Sources:

1. Moesif API Developer Portal – *Open-source portal with Okta, Kong, Stripe integration* [5] [17]
2. Gravitee API Management – *Open-source API platform features (portal, analytics, plans)* [13] [7]
3. Tyk Enterprise Portal – *Features of a modern API portal (customization, IdP integration, orgs)* [19]
4. Stoplight Elements – *Open-source interactive API docs components (three-column layout)* [10]
5. DociQL by Wayfair – *GraphQL documentation generator with Stripe-like layout and examples* [11] [12]
6. Cloudflare Blog – *Hono framework overview (fast, runs on Cloudflare Workers)* [26]
7. Blobr.io Blog – *"What Makes a Great API Developer Portal" (design and monetization insights)* [3] [20]

---

[1] [3] [4] [20] [21] [22] [33] What Makes a Great API Developer Portal
https://www.blobr.io/post/great-api-developer-portal

[2] [34] What is an API Developer Portal? The Ultimate Guide | Moesif Blog
https://www.moesif.com/blog/api-strategy/api-development/What-is-an-API-Developer-Portal-The-Ultimate-Guide/

[5] [16] [17] [18] Open-Source Developer Portal | Streamline Developer Experience | Moesif
https://www.moesif.com/solutions/developer-portal

[6] [15] [27] [28] GitHub - Moesif/moesif-developer-portal: An open source developer portal for API productization and monetization
https://github.com/Moesif/moesif-developer-portal

7 8 13 14 30 GitHub - gravitee-io/gravitee-api-management: Gravitee.io - OpenSource API Management
https://github.com/gravitee-io/gravitee-api-management

9 29 Kong Developer Portal with Okta Open ID Connect (OIDC)
https://ritesh-yadav.github.io/tech/kong-dev-portal-okta/

10 31 32 Stoplight Elements: API Documentation Tool
https://stoplight.io/open-source/elements

11 12 GitHub - wayfair/dociql: A beautiful static documentation generator for GraphQL
https://github.com/wayfair/dociql

19 23 24 35 Tyk Enterprise Developer Portal
https://tyk.io/docs/portal/overview/

25 26 The story of web framework Hono, from the creator of Hono
https://blog.cloudflare.com/the-story-of-web-framework-hono-from-the-creator-of-hono/