Nuvalence

ARCHITECTURE GUIDANCE SERIES

The Anatomy of a Digital Platform

WHITEPAPER

Purpose & Audience

When building a digital platform and bringing it to market, just knowing where to start and how to approach the effort can be overwhelming. After reading this paper, the hope is that you will be armed with a practical framework to ask the right questions, make the tough decisions, and execute successfully. The intended audience of this paper is a Chief Digital Officer, Head of Digital Platforms, Chief Technology Officer, Lead Architect, or anyone tasked with building a digital platform.

Understanding the Context

Over the past two decades, consumer expectation has put tremendous pressure on all companies to build and deliver applications. Consumers are no longer content with physical experiences, they demand digital experiences to supplement their physical world (smart homes, wearables, digitally-enabled shopping experiences, etc). To succeed, companies need to have ambitious digital strategies - they can no longer just build apps, they also need to build digital platforms to capture and support an ecosystem. The best and most successful digital organizations start with digital products that nearly always evolve into "digital platforms."

While "platform" is often used to describe things like Kubernetes or AWS, those are horizontal technical platforms. A digital platform is taxonomically best described as a domain-specific platform (e.g. automotive, healthcare, banking, retail) that abstracts common, high-value functions in a technology ecosystem into a shareable set of data, APIs, components, and capabilities. 3rd-party organizations can tap into these domain-specific "digital platforms" and build apps that create value for end-users that is complementary to a core "anchor application."

This benefits:

1. **End-users**, by providing them access to a rapidly evolving value stream.
2. **Internal and external developers**, by reducing technology implementation and market entry costs.
3. **The platform provider**, with a nearly unshakeable market position as its ecosystem of producers and consumers expands.

In some cases, the platform may actually make previously inaccessible markets accessible to 3rd-party organizations (for example, in healthcare or banking where the regulatory hurdles are generally too high for new entrants).

What might be most interesting in all of this is digital platforms in the context of existing enterprises. Existing, large organizations have a unique opportunity to leverage their entrenched market positions and significant assets (data, existing client relationships, access to capital). This advantage can help power a platform strategy with even more disruptive leverage and disintermediation potential than what's available to their nimble startup competitors. Incumbent leaders can extend their leadership power and own next-generation digital ecosystems in their markets.

While it's critical to understand the strategic context, this whitepaper will focus on how to approach the practical implementation of a digital platform. At a cursory level, the concept of a digital platform is best described as a technology layer that displays the following characteristics:

1. Provides tangible, specific value to consumers (both business and end user) by solving a problem in that consumers domain (this is the "killer app" that is at the core of the platform).

2. Accelerates the development of new applications by teams in your organization as well as by 3rd parties.
3. Displays flexibility in the range of use cases to which the platform can be applied to over time. It's often the case that the initial use cases and modalities become less relevant over time, so a platform should morph to fit contemporary use cases.

To start this exercise, we must first define "digital product." Digital products/services tend to be sufficiently defined as a technology where the original developer of that technology has a tight grip over the use cases and is the only party adding features to the technology. Many traditional SaaS offerings fit this static definition. In order for a digital product to become a digital platform, it is necessary to display characteristics 2 and 3. This implies a technical architecture that goes beyond a standard app or technology offering. The architecture needs to provide the ability for 3rd parties to safely drive value creation, and for maximum adaptability to ever-changing use cases.

This paper has two parts: (1) understanding the requirements of a platform and (2) designing an architecture to satisfy them.

Requirements

The components that define a platform aren't as obvious as one might think, primarily because characteristics 2 and 3 are supported by a deep set of subsystems. A requirements gathering exercise will help clarify this. To start, let's look at questions that can help outline the requirements of characteristic 2:

1. Accelerates the Development of New, 3rd-Party Applications that Sit Atop the Platform

This defining characteristic carries meaningful technical implications. Developers of a digital platform need to answer a few questions about the intended business goals and target state of the platform in order to understand how to approach enabling this characteristic:

1. **Does the platform lean more toward being a data platform or a workflow platform?** Nearly all platforms have a data and a workflow/logic component, but tend to lean more toward one or the other when attempting to commoditize patterns and boost developer productivity. Organizations focused on aggregating and correlating market transaction data or collecting high volume data from IoT devices, for example, may deliver the most value by abstracting their data architecture into a platform. Others who focus on payment processing, 3D rendering, or health insurance claims as examples might find that capturing their workflows and commoditizing development around those workflows provide the most value to developers.

   When defining a platform, is it more data-centric or more workflow-centric? This centricity decision will influence many details of the platform implementation.
2. **What are the platform's execution models?** Execution models are defined via two dimensions: (1) the interaction model (API, plugin, etc.) and (2) where a client application is hosted (off-platform or on-platform). A platform can be built to have one or more of these execution models, but it must have at least one:

API Platform	Domain-Specific Framework	Plugin	Rapid Application Development IDE
Off-platform/External	On-platform	On-platform	On-platform
The platform exposes functionality to 3rd-party apps via APIs. App developers need to decide where these apps will run since the platform does not provide a hosting capability. The platform markets itself as a metaphorical "hub" and acts as a critical data and workflow broker between app providers and end-users. Having APIs and taking a market position as a "broker" is the bare minimum expected of any platform.	The platform selects/creates and then layers a domain-specific framework atop a well-accepted platform stack (e.g. Kubernetes, Lambda), offering it as an integrated cloud to their ecosystem. The intent is to afford developers the flexibility of a general purpose programming language model (e.g. Java, JavaScript, C#, etc.) while providing domain-specific runtime services (e.g. big data access, regulated data access, massively parallel execution systems). Domain-specific platforms host apps because accessing the data or the programming model is impractical off-platform.	3rd-party apps may need to replace or modify standard behavior in the anchor app or in the platform. This is done via a plugin model. The platform provides an embedded code execution model where developers can write plugins and extensions to the anchor tenant application via an extensibility framework. This framework has to ensure runtime safety for the app so that errant plugins don't ruin the user experience. End users can select to install 3rd-party plugins from a marketplace. Plugins may be partially (e.g. front-end) or wholly (e.g. front-end + back-end logic) hosted by the platform provider.	The platform provider not only hosts guest apps, but also requires the use of proprietary development frameworks and programming languages. Generally, this requirement is in an attempt to maximize productivity, even if at the expense of flexibility. These platforms tend to focus on technical "power users" rather than developers, but often provide a way for more sophisticated code to be written via a plugin model. This execution model generally includes a powerful development environment and shouldn't be confused with highly flexible "power user" focused features.

In designing a digital platform, this trade-off will be one of the most important design considerations. A platform provider may choose to align with a particular model for strategic reasons, while in some cases, may be forced into a model out of practicality. For example, a platform in the banking or healthcare space may require that apps execute on-platform for regulatory reasons, or a platform in the 3D rendering space may host apps on-platform since pulling terabytes of data across a REST API is highly inefficient. Platforms often have multiple integration models depending upon the use case so multiple strategies can be employed.

3. **How is an additional layer of tenancy going to be handled?** Multi-tenancy is old news. Most digital offerings have a logical customer definition (e.g. tenant) and we know how to architecturally handle tenancy (despite it still being complex). Depending on how (1) is answered, a platform provider may need to now define the execution context as a pairing of the end user customer tenant context coupled with the context of the 3rd-party application provider. For example, imagine customer 1 is a user of the digital product at the center (the "core app") of the platform, but also a user of two ecosystem apps: app A and app B. The core app may have some sort of report or component injected into its runtime by app A, but not by app B (because the user may not have authorization to access some widget in app B).

The tenancy model now becomes a tenancy matrix, significantly increasing architectural complexity, but potentially delivering massive end user value.

4. **Are per tenant operational controls and diagnostics needed, or are those needs exclusively at the application layer?** Depending on the use case, a digital platform may need to intersect application and tenant context with operational controls and diagnostic needs. For example, if a tenant is using an application on the platform and is also using a 3rd-party extension to the app, does the platform need to provide tracing and debugging information on a per request basis with contextual information related to the tenant and application(s)? If so, will the platform trace requests across application boundaries and microservices, maintaining diagnostic context throughout?
5. **Where is the security boundary for data and for entitled access to workflows?** A platform may need to provide 3rd parties access to some data, or all data. If the answer is some data, the platform will have to provide a means to grant rights to 3rd-party apps to access data and workflows. Data access rights may be coarse or fine-grained in nature, with the platform bearing responsibility for balancing appropriate access with use-case enablement.
6. **Is the platform responsible for aggregating data sources and/or providing data lineage and data quality guarantees?** Coupled with the question of data access and security, if the platform is aggregating multiple data sources, the platform may need to provide tenant apps with guarantees regarding data quality, data lineage, etc. Without guarantees, tenant app developers may not be able to pursue certain use cases but liability may be lower for the platform provider. Guarantees may maximize platform utility but could increase liability for the platform provider.
7. **Is the platform going to provide a comprehensive SDK, or instead, a collection of APIs?** This is part driven by necessity, part driven by intended value. The platform, by its very nature, may require custom testing frameworks, simulators, command line tools, etc., especially if its goal is to accelerate development. On the other hand, that may create too much complexity (for both the platform creator and the 3rd-party developer) if it isn't required and where a set of APIs will be sufficient.
8. **How will the application handle inorganic increases in load?** Exposing an app to value extension by 3rd parties may cause meaningful, rapid increases or fluctuations in load. Where a digital product may be able to have a quasi-predictable scale model in place, once demand is democratized, that predictability is lost. This puts pressure on the platform provider to tightly control resource allocation, which can be handled through simple planning or may require an extensible, custom scheduler (which partly depends on decision around "execution model").
9. **Who is responsible for facilitating the monetization, quality, and trust of newly created applications?** In an ecosystem where a digital platform sits at the center, end-users identify themselves as customers of the platform rather than customers of each individual offering they subscribe to. The platform provider owns the relationship with the customer, and the customer views the provider as the vendor of record.

   Once 3rd-party applications are introduced, some customers might expect that paying for 3rd-party value happens through the digital platform provider, and not directly with the 3rd parties themselves. This is particularly true if end-users view the platform as an arbiter of trust and quality for 3rd-party apps.

A digital platform provider needs to determine if it will: (a) supply its ecosystem of 3rd-party apps with billing and payment facilities and simply "cut a check" to 3rd-party app providers on a regular basis for any sales or (b) if its expected that those 3rd-party app providers deal with billing themselves and choose from a best-of-breed billing provider.

This list is nowhere near exhaustive, but one can easily imagine that the answers to these questions have a huge impact on the architecture. Now onto the next definitional characteristic:

2. Displays flexibility in the range of use cases to which that technology can be applied

This characteristic is more nuanced but equally important. A platform provider needs to determine how flexible it wants the platform to be. Flexibility can be measured through how open the platform is to non-standard use. For example, take Excel and how broadly its flexibility has been exploited across use cases. A digital platform provider needs to answer some key questions related to what it means by "use case" when designing the platform's architecture:

1. **Should the platform accumulate data of indeterminate value to enable yet to be understood use cases?** A platform's central nature allows it to have access to broad user data and behavior. Some platforms may choose to collect the minimum necessary data for powering known use cases. Other platforms may want to broadly accumulate data and telemetry from end-users despite not having an immediate use for it yet. For example, a digital office suite such as Google Workspace or Microsoft 365 may be interested in tracking how many times I misspelled and backspaced my mistakes in this whitepaper, which could be used to power future capabilities and features built by 3rd parties. The scope of data collection intersected with who that data is made available to impacts customer privacy and use case expansion potential:

Availability	Collect the Minimum Necessary Data	Broadly Collect Data & Telemetry
Not Made Available to App Developers on the Platform	• Most Private, No Use Case Expansion Potential	• Somewhat Private, • Some Use Case Expansion Potential
Made Available to App Developers on the Platform	• Less Private, • Better Use Case Expansion Potential	• Least Private, • Most Use Case Expansion Potential

If this sort of data is collected, the platform provider needs to balance privacy concerns with maximizing flexibility.

2. **Should the platform consider future form factors?** While 15-20 years ago, all we really had was the browser, now, new form factors are being introduced regularly. We added mobile, wearables, voice systems (think Alexa, Siri, or Google Assistant), VR and AR, and a number of yet to be discovered modes of interaction. A platform needs to understand how it believes its value will be exercised in the future and prepare for it (e.g. how to store data so it's more readily useful in other form factors, or how to organize its APIs to account for differences in potential API consumption).
3. **How much configurability should the platform offer and what boundaries should it set?** A platform needs to decide what level of configurability it will afford to its applications, and what boundaries, if any, it will impose on an application's ability to execute based on this configuration. If the platform is highly configurable, it will demonstrate flexibility in the app's ability to declare needs and expectations: it may allow the application to request elevated access rights to resources, leverage a broader set of communications protocols, specify what happens in error cases, or decide what sort of versioning approach it wants to use. This comes at a cost of architectural complexity to still ensure safety and isolation despite the increased configurability.

   If the platform is less configurable, it may strictly govern surface area and optionality, creating constraints for developers but reducing architecture complexity as a result.

This list of questions should help support a fundamental understanding that when moving from product to platform, there is a significant increase in architectural and implementation complexity. It's very easy to answer "yes" to all of the questions posed above, but you don't have to build your end state platform in the first iteration. Trying to do so dramatically increases the chances of failure. It's important to note that the two best approaches to building a digital platform help reduce this increased complexity by relying on proper sequencing and iteration:

1. **Build then Extract** - Build the digital product first and extract the platform from the product. This requires carefully crafting the product to be "platform aware." Platform awareness requires that the applications architecture be built with specific cutpoints and abstractions in mind, and that the initial implementation can be extracted into a separate, independent architectural layer that will serve as the foundation to the platform after the product is released.
2. **Build in Parallel** - Build the platform and product side-by-side, using the product as a first reference application and "customer" of the platform. In this case, the application is designed against a series of platform contracts from the beginning, and the work to implement the various supporting platform systems is completed at the same time. This leads to both the app and platform being released at the same time, which may be critical in certain competitive situations and go-to-market scenarios.

"Build then extract" may be less risk but more work in the long run, while *"Build in Parallel"* may be more risk but less work given that it doesn't require an intermediary extraction project and context switch. Generally, the recommended approach is the "Build then Extract" approach defined in (1) but there are circumstances where (2) makes the most sense. When using the "Build then Extract" approach, the risk and additional work profile is defined by how much energy was invested in building the initial offering in a "platform aware" way.

Irrespective of the approach, the architecture should converge to the same end-state model. To help better shape an understanding of this model, the remainder of the white paper will outline a reference architecture and high-level example implementation.

A Generalized Reference Architecture

Most digital platforms will have a somewhat common set of use-cases and supporting architecture despite differences in technology selection and implementation. This allows for the definition of a reference substrate (a "digital platform DNA") that can be shared across specific digital platforms and captured as a reference architecture. To help frame this reference architecture, let's define specific feature expectations for a fictitious reference platform named "Hub." Hub will be a platform created with a definitional, anchor B2B application at its core. The anchor application will be extended to rely on platform features, making it the platform's first meaningful consumer. Hub will generalize and abstract key functional and architecture patterns related to this anchor application so that 3rd-party partner developers can now build and deploy client apps to Hub. Hub provides:

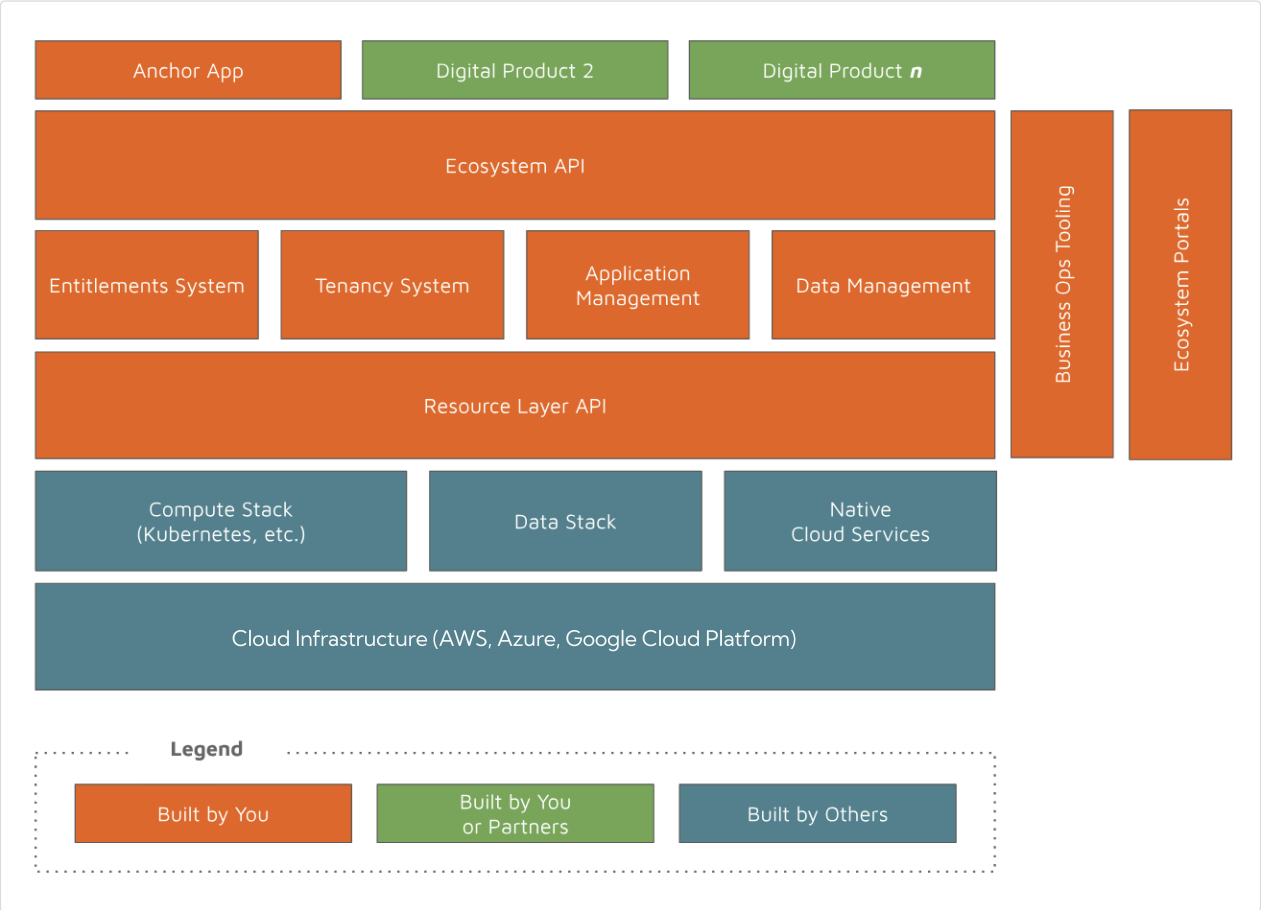
1. APIs, giving client apps access to core workflows and functionality.
2. The ability to host microservices that power the 3rd-party apps.
3. A data layer with access to data critical to the use cases supported by Hub.
4. A frontend framework and execution context, allowing 3rd parties to write Javascript/Typescript based interfaces that extend the core interface of the anchor app or as standalone Hub UIs.
5. An app publishing system, so apps can be registered in the ecosystem such that end users can install those apps to their Hub accounts and start using the applications.
6. Workflows, allowing end users to self-service their interest and entitlement to the client applications.

As discussed earlier, a digital platform is a domain-specific platform that tends to target the functional needs of an industry. An infrastructure software layer like Kubernetes may power the app-hosting and compute needs of a digital platform, but that's only a small part of the broader functional requirements. Multiple other open source and cloud technologies, along with significant bespoke software development, would surround something like Kubernetes to actually turn it into a digital platform. To hone in on what technologies we need to use alongside a cluster/hosting layer and what sort of custom development we need to undertake to create Hub, we'd have to drill in a bit further and outline definitional architecture "characteristics":

Characteristic	Architecture Choice	Details
Platform Focus	Data & Workflow	Hub doesn't optimize around data or workflows. It sees both as equals, so its execution context provides reasonable leverage and value around both application layers.
Execution Model	Plugin	Hub is taking a "plugin" approach to the client app execution model, where client apps run as a "guest" to the platform. Applications built for Hub run embedded in the core app and platform and are designed to implement a narrow interface provided by the platform. The app need not worry about infrastructure, OS', or any scaffolding. The app's code is executed by the Hub platform and is provided the necessary execution resources and hooks. This embedded execution support is for frontend UI extensions and a "lambda-like" architecture for basic data processing. Alternative execution models were considered, but based on customer expectation and lack of practicality, the plugin model offered the best blend of flexibility and productivity. Additionally, Hub also offers an API execution model for externalizing app creation.
Tenancy	Nested Multi-tenancy	Hub has a two-class tenant model: both apps and end-user organizations are tenants (app tenants and org tenants, respectively), each having context scopes. Hub registers and tracks tenants of each class, and allows org tenants to be sub-scopes of app tenants. This allows Hub to affix data to org tenant scopes, app tenant scopes, and to org tenant scopes nested in app tenant scopes. For example, App A may have Tenants 1 and 2, so A-->1 and A → 2 are different tenant scopes, both under the A tenant scope.
Diagnostic Model	Coupled App & Tenant Contexts	Hub provides trace and debugging information consistent with the nested multi-tenancy model that is provided as part of the execution context. Any logging and debug information can be associated with both the app tenant as well as the org tenant whose request was being processed during any diagnostic tracing.
Entitlements	Data, Features	The platform protects data and workflows via app tenant published entitlements and org tenant permissions. That is, app tenants express data and workflow access needs to the platform, and org tenants grant permissions against those requests. When an org tenant requests to use a given application, the platform will publish that applications access requirements to the org tenant, requesting permission to allow the level of access requested by the application
Data Guarantees	Data Lineage, Provenance	Hub can provide app tenants access to data lineage information, allowing it to propagate guarantees to end-users. Additionally, any allowable data modifications are tracked to ensure provenance.

Characteristic	Architecture Choice	Details
Developer Surface Area	App APIs, Management Portals	Nearly all of Hub's functionality can be accessed via an API, which is useful for code executing outside of Hub. Given that Hub can host applications, however, it must choose a language/stack that will be considered "Hub Native." Hub will allow developers to leverage Javascript & Typescript as first-class languages for 3rd-party app development, and those apps can be run directly on the platform. Hub will "sandbox" those apps for execution, giving the app a runtime context. No specialized development tools, emulation layers, etc. are needed, so Hub provides more of a collection of APIs surrounding this sandbox rather than a heavier SDK. Hub also provides developers access to portals and APIs for app publishing and management.
Scalability	Elastic	As load increases, Hub can rely on autonomic scale-out of underlying infrastructure. Consequently, it can scale-in on diminished load.
Monetization	External	Hub does not provide any "app store" or transaction processing capabilities. Any monetization needs are left to the 3rd-party developer.
Data Collection	Central, Peripheral	Hub's developer expects that its users will evolve their usage over time. Additionally, Hub's developer anticipates that advancements in machine learning and data analysis will lead to significant future increases in value. As a result, Hub not only tracks data central to the core use case of its anchor tenant app, but also tracks as much peripheral data as possible since it may prove valuable to both Hub's developer and to end-users in the future.
Form Factor Tuning	Web, Mobile	Hub will be focused on current form factors, providing APIs and documentation tuned toward Web & Mobile. Optimizations will be made for payload size and reducing call frequency so that mobile experiences can be optimized for any plugins built for the platform.
Configurability/ Boundaries	Low/Strict	Hub will enable developers to write plug-in code, but won't afford much configurability outside of what can happen within a plugin's memory space. This reduces the architectural complexity associated with policy driven flexibility. Hub's primary goal is to allow plugins to influence the anchor app's data and workflows, expose UI controls to end users, and enrich data via external HTTP data sources. Outside of this scope, Hub intends to provide tight boundaries and very little configuration.

Hub's architecture needs to enable these characteristics in addition to the fundamental requirement of hosting 3rd-party workloads. Some of these characteristics will wholly exist in a single subsystem while others are manifested across multiple components and don't exist as a specific architectural component. Given the requirements for Hub so far, we arrive at a logical architecture that looks like this:



To better understand this logical architecture, it's important to understand the role of each component, starting from the bottom of the diagram and working up. For each, we provide a general description and a non-exhaustive list of technology options:

1. **Cloud Infrastructure** - For Hub, there is little value in leveraging any infrastructure other than cloud-based (or possibly hybrid) infrastructure. Hub's use case allows for app-data to be loaded into the selected cloud providers data offerings, skirting one of the common blockers that would prevent a new platform from leveraging cloud.
   a. **Technology Options:** AWS, Azure, or Google Cloud Platform
   b. **Enabled Characteristics:** Elastic Scalability
2. **Compute Stack** - Hub will leverage a container orchestration service at its core. The choice is primarily defined by Hub's need for a dynamic, programmable infrastructure. This gives Hub access to the necessary declarative primitives and APIs to shape infrastructure according to the needs of the anchor application and of 3rd-party client applications. Plugins will run in isolated execution runtimes (e.g. lambdas) that will be referred to as "Backend Isolation Units" or BIUs.
   a. **Technology Options:** Kubernetes, AWS ECS, Azure CS, Lambda
   b. **Enabled Characteristics:** Elastic Scalability, Embedded Execution Context
3. **Data Stack** - Hub apps rely on a combination of real time data streams from endpoints around the globe, infrequently updated unstructured data and relational data provided by end users. To handle this, Hub leverages a streaming data ingest framework, loading and transformation service, a high performance cache layer, an unstructured document store, and a relational database.

a. **Technology Options:**

   i. Streaming Data Ingest/Distribution: AWS Kinesis, Kafka, Microsoft Event Hub, Google Pub/Sub

   ii. Relational Database: Postgres, MariaDB (managed cloud instances preferred)

   iii. Document Database: AWS DynamoDB, Microsoft CosmosDB, Cassandra

b. **Enabled Characteristics:** Elastic Scalability, Data Lineage & Provenance, Central & Peripheral Data Collection

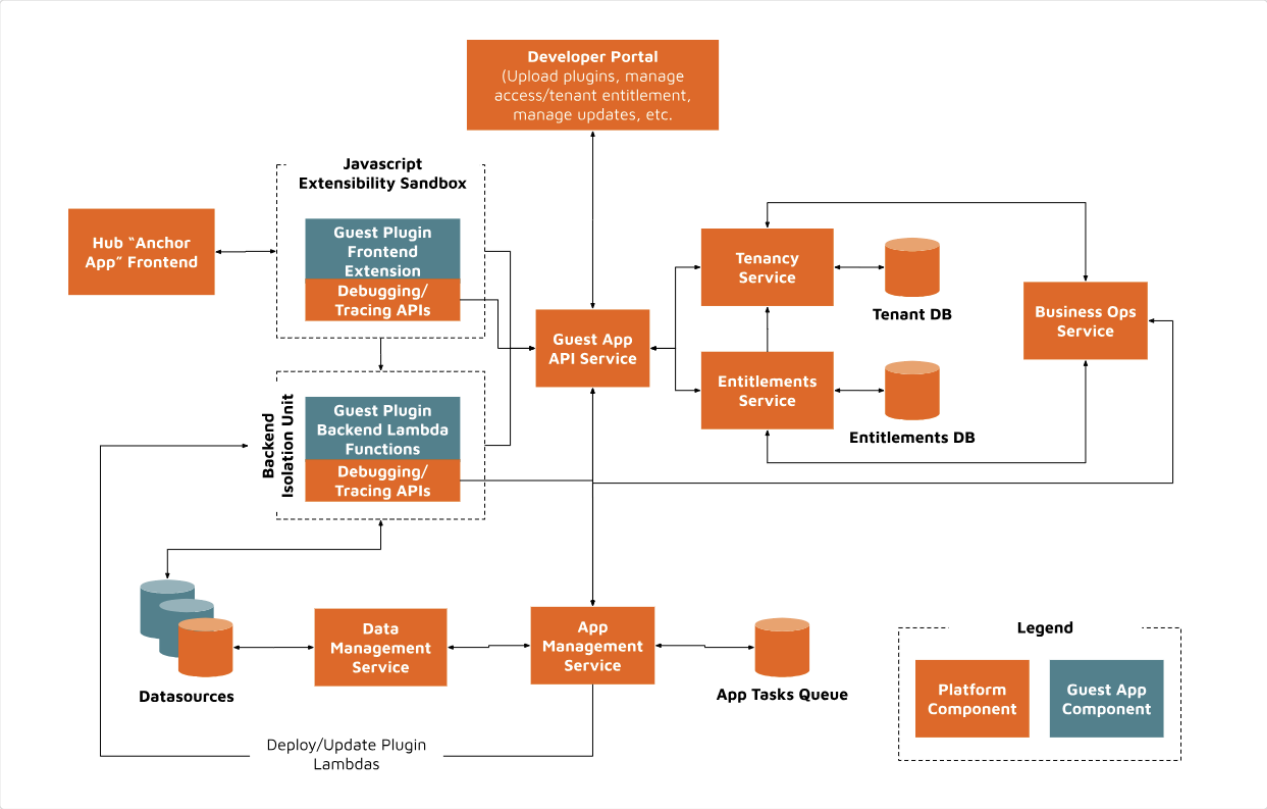
4. **Native Cloud Services** - Hub will rely on necessary peripheral services to access any commodity functionality it may need, including but not limited to global load balancing, CDN, and authentication. This will be unique by use case and deployment model.
5. **Resource Layer API (RLA)** - Hub will have its own API for managing resources in a manner consistent with its architecture expectations. For example, Hub will have a specific expectation of what infrastructure manipulation needs to occur to support a newly published plugin. Rather than expose raw cloud and infrastructure APIs to upper portions of the stack, the RLA will expose resource management via functions such as ConfigureAppInfrastructure (appName:string, ...).
6. **Tenant Management** - The Tenant Management System (TMS) is responsible for storing and manipulating tenant meta-data. It defines first class app and org tenants, and manages the relationship mapping between them. Its duties include managing tenant onboarding and offboarding, tenant context establishment and manipulation, and tenant record maintenance.
7. **Entitlements System** - The Entitlements System provides intra and inter app entitlements management. Through this system, the platform tracks what tenants are allowed to use what apps, what apps can communicate with what apps, and what features within an app each tenant user has access to. Additionally, the system provides data entitlement registration mechanisms to describe what data each application has access to.

   The system exposes APIs to manipulate entitlement mappings as well as to compute entitlement claims and grants. This gives app developers the power to weave entitlement declarations and logic into their app stack.
8. **Application Management** - This system provides app deployment, registration, and manipulation functionality for tenant applications. Hub relies on this system for app management and monitoring needs. The system provides (1) abstract information about app status and debugging/tracing information and (2) exposes controls for registering the application in an "app store" like context so they can control how end-users get access to and consume the application.
9. **Data Management** - Applications tend to rely on multiple data sources. Hub provides core "domain specific" data to the app developers (e.g. banking or healthcare data if Hub were the core platform for a company in that space).

   The Data Management system allows developers to describe their data needs, giving the platform information it can use to properly curate and entitle access to data needed by the application. This layer interacts with the entitlement system as a mechanism to guarantee proper access rights. Additionally, apps can register for custom data streams so the app receives only the data it needs for its use cases.

10. **Ecosystem API** - Rather than exposing subsystems directly to plugin app tenants, apps interact with the Hub via this curated and controlled API layer. The Ecosystem API aggregates and exposes functions to app tenants via a single surface area that's organized functionally and well documented. The API is not public, but instead, is available only to authorized plugins.
11. **Business Ops Tooling** - Hub recognizes that a number of the people who manage 3rd-party app interactions are not technical (e.g. customer service staff, sales staff) and provides portals (and APIs where appropriate) exposing high level business controls to those individuals. This is useful for managing support desks, investigating app status in the context of business workflows (e.g. activating access to a deployed app when end users manually submit payment for an app, deactivating on lapsed payment, etc.).
12. **Ecosystem Portals** - Hub may provide end users with a portal that contains self help guides, generalized Hub support desk functionality, etc. These portals can be as shallow or as dense as required by the Hub business model, and may or may not include surface area related to specific 3rd-party applications.

This logical architecture provides comprehensive coverage for the general expectations and behaviors for Hub. While useful, setting the direction for Hub's implementation requires a finer grained reference component & systems architecture:



Implementing a systems architecture similar to this one will likely be an exercise that combines developing some of this architecture "from scratch" with incorporating some best-in-class OSS components and cloud services. In practice, an architecture like the one proposed for Hub is just a starting point.

The most important takeaway is that moving from product to platform requires an architecture similar to this one as an addition to the apps domain specific architecture. A number of domain

and model specific components would expand this architecture significantly, resulting in a final architecture that is well-aligned with the platform outcome necessary for winning over the target ecosystem.

Conclusion

As you initiate your digital platform journey, it's important to take stock of the goals that your partners and end users have, and use those goals to shape the platform functionality needed to allow for rapid innovation around your core offering. There is no "one size fits all" approach to building digital platforms. Every business and every team is unique with its own set of priorities, constraints, skill sets and timelines. Hopefully this paper has given you a framework for getting started.

ABOUT NUVALENCE

Nuvalence is a next-generation consulting firm specializing in mission-critical, intelligent platforms for the world's most ambitious organizations.

Using our product-driven, AI-centric approach, we empower organizations to build for the intelligent digital future. Our elite team of product leaders, data scientists, designers, and software engineers enables our clients to solve their most complex technology product challenges and positively impact people and the world.

We don't just deliver software, we deliver outcomes.