POC Requirements for API Management

Evaluating functional and nonfunctional capabilities of an API management solution

Table of Contents

1 Introduction

The platform should be able to:

- Create easy to customize and manage, well-designed APIs from existing services
- Enable API consumers and app developers to build innovative, engaging apps, by providing frictionless self-service onboarding, easy access to API keys and API secrets, and rich interactive API documentation
- Drive developer adoption and help build a community of internal and external (i.e. partners and independent third-party) developers
- Extract operational and business insights from our API and app ecosystem
- Provide the ability to monetize APIs using different rate plans

The solution should sit between back-end services and consumers, providing much-needed flexibility to complement and enhance a variety of business and low-level functionality. This will enable the API team to focus on creating value from core business functionality via APIs and off-load all non-functional aspects of exposing and managing the APIs to the API management solution.

2 Purpose of the proof of concept

This section in the document details the key objectives that drive the evaluation, covering both functional and nonfunctional aspects of the API management product to establish viability and differentiation.

Business need/value alignment

Outlined below are the business drivers for this POC:

<<Replace the following with your specific business needs/value alignment>>

- **Example:** Improve agility of our business units in creating omnichannel ecommerce experience
- **Example:** Reduce amount of time required for partners to start conducting business with us
- Business need #3
- Business need #4
- Business need #5

Key technical drivers

While specific use cases will be defined further in this document, at a high-level the following items are specific technical drivers this proof will validate:
<<Replace the following with your specific key technical drivers>>

- **Example:** Need the ability to provide user and product data to various internal business units responsible for our omnichannel ecommerce strategy
- **Example:** Need to enable our partners to place orders with us in real-time from their mobile apps
- Technical driver #3
- Technical driver #4
- Technical driver #5

3 PoC timeline

	Item	Date Started	Date Completed
●	Sign NDA & evaluation agreement		
●	Define and document POC use cases		
●	Define and document POC success criteria		
●	Define and document POC timeline		
●	Provide details about back-end APIs/endpoints		
●	Validate connectivity with target endpoints		
●	Conduct hands-on workshop with the solution		
●	Conduct architecture and security review		
●	Confirm logistics (venue, badges, start time, point of contact)		
●	Provision API management solution		
●	Conduct POC kickoff meeting		

●	Implement POC use cases		
●	Demonstrate use cases		
●	Conduct POC readout meeting		
●	Define and document next steps		

4 High-level evaluation approach

Perform a cloud or on-premises POC to validate key aspects of API management relevant to our business, including:

- Enterprise-level API management and the ability to expose a uniform façade
- Security aspects of the platform
- Customizability–quick and easy customization of APIs for different development groups
- Easy developer on-boarding and rich documentation for internal and external APIs
- Out-of-the-box metrics and performance and analytics reports
- Monetization capabilities
- Performance and scalability capabilities
- Multi-tenancy and multi-environment
- Fit within DevOps environment

Mutual NDAs must be signed before the evaluation begins.

5 PoC prerequisites

Outlined below are the necessary prerequisites for completion of the POC.

5.1 High-level architecture of backend services/applications

<<include relevant architecture diagram>>

5.2 Infrastructure

<<include relevant infrastructure information>>

5.3 API details

<<include relevant API document and details>>

6 Use Cases

6.1 API management requirements

6.1.1 API lifecycle

No.	Key Focus Area	Description	Priority (H/M/L)
1	Design-first	Demonstrate the support to create a facade and to generate API documentation using OpenAPI (formerly Swagger) specs to support a “design-first” approach	
2	Publish	Demonstrate how resources from one or more APIs can be packaged together and published for easier consumption to serve different API consumers, restrict access to certain resources, and support different quotas	
3	Publish	Demonstrate how to publish APIs for different purposes; sandbox, internal, and production.	

6.1.2 API Runtime

No	Key Focus Area	Description	Priority (H/M/L)

1	Traffic management/ throttling	Demonstrate out-of-the-box traffic throttling capabilities to protect the back-end systems from unusual spikes	
2	Protect against unusual spikes	Decide on the spike rate based on factors including payload size and type of service it is calling	
3	Control number of open connections from API tier to backends (rate limit)	Demonstrate out-of-the-box traffic throttling capabilities to control number of active/open connections from API layer to our back-end systems	
4		Control/configure the number of open/active connections by backend endpoint type	
5		Limit the total number of connections to our back-end systems across distributed runtime instances of the platform	
6	Quota	Demonstrate different kinds of API access limits (by API key, app, API user, and end user)	
7		Demonstrate SLA capabilities based on the consumer. (partner types, app types, environment, and products)	
8	Caching	Explore some out-of-the-box caching from the API tier covering the scenarios noted below **Caching back-end response**s: the API tier should serve as a caching layer between the apps and the backend on frequently called services/resources. E.g. if a response from a certain service is resource intensive and is not going to change for a relatively long period of time (hours or a day), the API tier should cache it and serve it from the cache for the subsequent responses.	
9		**Caching for state management**: the API tier should be able to do state management across different APIs.	

		E.g. if API “X” is caching a state about the user preferences, retrieval of that stored state (cache) from API “Y” should be enabled.	
10		**Caching content based on HTTP 1.1 caching header directives from back-end target (i.e. origin) servers** Caching headers and directives including max-age, no-cache, s-max age, expires, e-tags, and accept-encoding.	
11		**Caching content by consuming apps**: the ability to cache entries(buckets) by app. E.g. if partner app “A” is calling API “X” since the response will be unique to that app, the cache should as well be applied to that app ONLY.	
12		**Cache content by app user**: the ability to cache entries for different users of the app. E.g. if User A is using an app, and an attribute in the API payload can identify the user, the response should be cached specifically for that user. Each user should have their specific data returned from the cache upon subsequent requests.	
13		**Cache invalidation**: the ability to define a clear way of flushing/clearing out the caches. E.g. when a user logs out, we should be able to release all the cache that was captured during that session for a user. There should also be a clear way of flushing the caches from the back-end responses (see above) if the backend updates before the configured cache expiry.	
14	Config store	Demonstrate the ability to externalize values that can be stored, retrieved, and referenced during API call processing.	

16	Security	Map internal authentication/authorization mechanisms to API keys and OAuth. Should be able to perform Credential Mediation. E.g. Apigee should only validate keys/tokens before or after successful authentication from identity providers.	
17		Back-end systems need base-64 encoded credentials (basic auth). The API platform should encode the credentials coming in plain text from the applications before sending the request to the back-end system.	
18		Authenticate the runtime API calls against existing LDAP system.	
19		SAML is used for SSO across different applications. The API platform must generate as well as validate a SAML assertion, providing flexibility to use it as either a service provider or an identity provider.	
20		A unified solution against SQL injection is required, as is the ability to restrict and control payloads and headers and ensure malicious code does not make it through to the backends. The threat model should be extensible against multiple data types, including XML and JSON.	
21		The ability to block or blacklist traffic coming from certain IPs hitting the API tier is required. Also required: the ability to allow or whitelist only a certain set of IPs hitting the API tier.	
22		Network or transport-level security between the client apps and the API tier is needed. This should be achieved that using standard protocols, including TLS. All the options of client-server hand shaking should be explored.	

23		Network or transport-level security between the API tier and our backend systems is needed. This should be demonstrated using standard protocols like TLS. All the options of client-server hand shaking for this should be explored.	
24		Because the API platform will be integrated with several distinct back-end systems, we need a unified way to throw and handle errors such that it appears as such to our consumers/apps. The API platform should be the place to map all errors from different backends and give one standard error format.	
25		We want to customize errors by consumers (apps). We also need to be able to control the granularity of the error messages by apps. E.g. we want to give out more detailed error messages to our “trusted”/internal apps and less detailed errors to external apps.	
26	Error handling, logging	To troubleshoot or debug an end-to-end use case, we need to log all the responses (success and error) from the API tier to our logging systems (syslog). We also want to tie a unique ID to each message so we can map it pinpoint issues.	
27		Sending logs from the API tier to our logging systems should be done securely.	
28		Logging messages from the API should have zero impact on the performance of an actual API call.	
29		We need to be able to create workflows (tie in a webhook) to certain conditions, including our backend server erroring beyond certain counts (for example) such that we would receive a notification.	
30	Extension	We need to handle business logic in our API tier. Depending on the use case we might want to handle it	

		either in JavaScript, Java, Python or, for some IO-based utilities, we might want to handle that with Node.js	
31		We need to integrate some non-http endpoints, adapters, or jars through our API tier.	
32	Custom Policies	We want to be able to create our own features that aren't covered in the product.	
33	Debugging	We want to be able to debug issues in different ways. **Run time**: We want some mechanism to debug the API at runtime and get insights on what's coming in and what's going out, and pinpoint the exact point of problem.	
34		**After the fact:** for some difficult to reproduce scenarios, we want the ability to analyze the API traffic for debugging purposes.	
35		We want to plug in our existing monitoring technologies with the API platform.	
36		We want to mask sensitive personally identifiable health information attributes within the payload during debug sessions	
37	Orchestrations	We want to use multiple target backends in our APIs, and do load-balancing and build conditional backends.	
38	Mashups	We want to build mashups, conditionally calling services for validation and/or data enrichment.	

6.1.3 API Backend-as-a-Service

Because APIs are stateless and used to build modern apps, they require a Backend-as-a-Service that provides the ability to do state management, provide server-side support to manage and authenticate app users, and the ability to create APIs for sending in-app push notifications or issue geolocation queries. These capabilities make it easier for API consumers to build robust interactions and applications.

No.	Key Focus Area	Description	Interest
1	API-driven scalable data store	The API-driven, highly performant NoSQL data store should be easily scalable to our growing needs. This data should be fetchable in a performant way through runtime RESTful APIs.	
2	Security	The data should be protected and secured allowing access to only authenticated entities (APIs and apps) by standards like OAuth.	
3	Granular data access controls	We need access control to configure which users are allowed to execute the RESTful API associated with the data.	
4	Special data formats	We want the ability to store images, videos, and audio within the datastore and provide RESTful APIs to handle CRUD operations on that data.	
5	Data querying	We want the ability to retrieve data from the NoSQL datastore using RESTful APIs that understand SQL-like queries.	

6	Device/user specific data store	We want the ability to store and create a relationship between users and their mobile devices and need the ability to query data for a specific user and/or device.	
7	Geolocation driven querying	We need a way to store and query data based on a particular geolocation. E.g. we need to be able to fetch all our users within a certain range of distance from a particular geolocation	
8	Referenceable data sets	We the ability to create reference relationships between different data sets. E.g. if we have a customer data set containing each of our customer entries and a product data set containing our product information, we require a way to create some link between the two in a way that produces a list of all the customers who viewed or purchased a particular product.	
9	Pagination support	We need out-of-the-box support for pagination that would be controlled by data querying. We need the ability to control how much data is returned in a query and iterate through the data on a per call basis.	
10	Push notification support	For mobile apps, we want the ability to send in-app push	

		notifications to users. We want the ability to leverage Apple, Google, and Windows push notification service notifiers.	

6.1.4 Analytics

Robust analytics provides complete insight and visibility from the developer apps that are using the APIs, the APIs themselves—their traffic, performance, success rate—right down to the target endpoints that the APIs hit. The solution must provide:

- complete activity, performance, and error/alert reporting
- API segmentation by traffic, performance, success rate, and a host of other metrics
- a fine-grained view of how APIs are being used by the consuming apps and usage by API method to know which APIs to scale
- assistance in troubleshooting anomalies and errors
- the ability to create custom reports on both operational and business-level information; as data passes through the API management layer, default types of information should be collected, including URLs and IPs for API call information, and latency and error data

Besides out-of-the-box information that's collected, the solution must also provide the ability to easily configure extraction of data from the XML or JSON request or response and make it available for analysis. All data should be pushed to analytics where it can be aggregated and leveraged by built-in or custom reports. Analytics should also provide fundamental administration services, including user and role management.

No.	Key Focus Area	Description	Interest
1	Real-time dashboards	The API platform should provide out-of-the-box, real-time dashboards with dynamic drill-down and exploratory visualization features that highlighting overall traffic trends, error rates, and API usage. Separate dashboards should be provided to view metrics from different deployment	

		environments including test and production.	
2	Drill-down and time range	Dashboards provide the ability to drill-down based on various dimensions and time ranges. Users should be able to zoom into a specific time period across which they wish to view metrics.	
3	API performance metrics	The API platform should provide dashboards that measure API performance including response times, latency, and error rates. Users should be able to view metrics across different dimensions such as API, application, developer, response status, and client identifier.	
4	API target back-end metrics	The API platform should provide dashboards that measure trends in the responses from target back-end services that the APIs invoke. These should include metrics such as target service response time and target response errors.	
5	Usage metrics	The API platform should provide real-time dashboards that measure API usage distributed across developers, apps, and client devices. It should provide metrics including the volume of traffic generated for a given API or by a specific developer app and the number of successful and failed API calls.	
6		The API platform should provide real-time dashboards that measure	

		trends in API usage over specific time periods, as well as API transaction rates at specific times. The platform should also provide real-time trends such as top APIs and top apps, based on API usage over time, such that business users can measure the adoption of APIs.	
7	Custom dashboards	The API platform should provide the ability to create custom dashboards by customizing any parameter, preferably through the UI. Users should be able to create reports that measure different metrics across various dimensions. Users should also be able to extract any custom metric from any part of the API request or response—URIs, query parameters, headers, and payloads—and use them to construct a real-time custom dashboard.	
8	Custom dashboards: analytics data filtering	While creating custom analytics dashboards, users should be able to set filters on the data to be collected for the dashboard. E.g. a dashboard can be created to measure a metric only when a specific value occurs for a given query parameter.	
9	Analytics data export	A user should be able to export data for external analytics and reporting, and apply powerful filters to ensure the export of only relevant information.	
10	Analytics tools	Demonstrate troubleshooting tools available to debug any anomaly in traffic.	

11	Developer analytics	Demonstrate how to support reports for developers on their own API usage.	

6.1.5 Developer portal

The API management solution should provide a developer portal, which should have out-of-the-box community features including blogs, forums, and FAQs that will help build a developer ecosystem for internal developers or externally exposed to partners and third-party developers. It should be easily customizable and rebranded, and should include mechanisms for secure self-service registration and developer onboarding(whether internal, partner, or external).

The portal should also include the ability to create intuitive interactive documentation that can be annotated by each developer and used to test and view API results in real time. Apart from content management, the portal should offer features for community management such as manual/automatic user registration and moderating user comments. It should offer a Role Based Access Control (RBAC) model that controls access to portal features (for example, it should be able to control whether registered users can create forum posts or use test consoles).

No.	Key Focus Area	Description	Interest
1	Developer registration	Demonstrate self-service developer account registration with developer email, username, and password. On the management side, the publisher should be able to activate a developer account easily.	
2		Demonstrate how developer registration and application registration workflows can be customized according to the API publisher's specific processes and policies.	
3	Developer account management	Demonstrate how developers can manage their accounts. E.g. show how developers can login and edit their account settings, view and manage applications, and subscribe to APIs.	
4	Application management	Demonstrate how developers can register the applications that they build against their	

		developer account, to access APIs provided on the portal.	
5	Developer roles	Demonstrate the ability to assign different roles to developers and portal admins.	
6		Demonstrate how parts of the portal can be restricted to specific developer and admin roles. Make API documentation and API product access role-dependent.	
7	Developer terms and conditions	Demonstrate how to make sure that all developers have agreed to terms and conditions during registration.	
8	Developer authentication	Demonstrate how various authentication mechanisms can be used (local credential store, social login) for developers to access the portal.	
9	App/API authentication	Demonstrate how developers can generate and manage credentials to access APIs from applications that they create, to access APIs in both sandbox and production environments.	
10	API documentation	Demonstrate the capability to host and manage API documentation, provide rich, interactive model-based documentation that's easy to edit and publish.	
11		The interactive documentation should have the ability to test the APIs that are deployed in a sandbox environment directly from the portal.	
12		Provide ability to host the documentation pages offline.	

13		Provide the ability to embed code snippets for each API into the documentation, as references to invoke the API in different programming languages and platforms (Java, JavaScript, Android, and iOS, for example).	
14	API productization and catalogs	Demonstrate how APIs can be offered as products under different packages in an API catalog. Also show developers can subscribe to different API packages under different payment models.	
15	API categorization	Demonstrate how APIs can be categorized and grouped, based on the different solutions that the API publisher wishes to offer. Developers should be able to select different API products based on the solution offered.	
16	Developer community	Demonstrate how a developer community can be fostered via forums and blogs.	
17	Content management	Demonstrate how content can be easily created and managed by the API publisher and developers. Also demonstrate the different types of content that can be hosted (articles and videos, for example). Also, show content viewing can be restricted based on the role based access.	
18	API design	Show how API publishers can design and build APIs using tools following a “design-first” approach, using well-known standards such as OpenAPI (formerly called Swagger) or WADL. Publishers should also be able to auto generate API documentation based on these designs.	

19	Extensibility	The platform should be extendable to accommodate future needs like integration with a ticketing system, JIRA, SFDC, and SSO.	
20	API sandbox	Demonstrate how developers can test APIs in a sandbox environment.	
21	Search	Demonstrate how content on the developer portal, such as API documentation, forums, and blogs can be made searchable. Demonstrate search features for API users non-structured content.	
22	SEO optimization	Demonstrate how we can improve search engine ranking with the developer portal.	
23	Branding and customization	Demonstrate how the developer portal can be customized to brand it to be consistent with the publisher's other IT assets.	

6.1.6 Monetization

The API management solution must provide:

- a flexible out-of-the-box mechanism to monetize APIs
- the means to package groups of APIs for different API consumers
- the ability to apply different ways to meter the usage of those APIs using different rate plans, which should be easy to configure using a web-based console and enable experimentation with different rate plans

No.	Key Focus Area	Description	Interest

1	Basic use case	We want to be able to create several different packages out of our APIs, grouping them by functionalities. Our consumers (developers and partners) should be able to subscribe to product offerings and access those services, which will be mandated by a count (“X” number of calls per month). E.g. we have assets (APIs) A,B,C , and D. We want to create 3 product offerings: Silver, Gold, and Platinum. Silver will have access to A and may make 250,000 API calls a month. Gold will have access to A and B and may make 500,000 API calls a month. Similarly, we want to create a premium product Platinum which will have access to all assets (A, B, C, D) and will be able to make a million API calls a month. The consumers should have the correct access in terms of features and count according to the purchased offering.	
2	Onboarding, self service	The entire subscription experience for our consumers and partners should be seamless and self-service. It should be a one-stop shop experience for them to review, select, subscribe, generate billing document, and reports.	
3	Purchase/subscription options	Our consumers should be able to subscribe and purchase in either a post pay (bill later) manner or in a prepaid (charged in advance) way.	
4	Promotions (trial period before charging)	The platform should enable running a promotion before charging our consumers. E.g. we start a new service, and want our consumers to use it free for the first month,	

		then charge them after the promotion period.	
5	Charging rules	We want to be able to configure different types of charging rules. **Charge a flat fee for all calls** e.g. charge a flat fee of 2 cents per call. **Charge a flat fee by segments** e.g. charge a flat fee of $5 for the first 1,000 calls, and $10 for calls 1001-5000. **Charge a fee based on call volume** e.g. for the first 100 calls charge 5 cents per call, for 101-1,000 charge per 3 cents per call, and 1001-100,000 charge 2 cents per call.	
6	Other charges	We want to be able to configure some one-time fees like initial subscription fees, and recurring fees.	
7	Notification when successfully subscribed	Once consumers have successfully subscribed or purchase a product they should get a notification stating a successful subscription or purchase with the details of the subscription (allowed counts and end date, for example).	
8	Errors when exhausted, notifications before limit	Consumers should not be able to access the services if they have exhausted the account limit. They should receive an alert if they are nearing the usage limit.	

9	Notifications, branding, logos	The notification should be personalized with the name and account ID of the recipient, and company branding.	
10	Grouping consumers/partners into a single entity	Each of our partner firms has several consumers within them. We want to design monetization with this aspect in mind. E.g. we should be able to have a partner subscribe to a plan and all the consumers within that partner firm should get subscribed to that plan	
11	Reporting	The platform should enable comprehensive reporting on the usage of assets—their consumption, consumers, charge, and balance.	
12	Bill generation	The platform should generate and publish a document outlining the bill including factors like usage, fees, and charges. Marketing content should be configured on this document.	
13	Credit and refunds	We want the ability to configure and control other aspects like credit adjustment and refunds.	

6.1.7 Operations and architecture

Alignment with our operations and architectural principles is an important aspect to evaluate as part of this POC. The volume of API calls in a successful API program requires tremendous scalability. The solution needs to fit in with existing tools and best practices, and must also be able to integrate with existing monitoring, CI, and software configuration management (SCM) tools.

No.	Key Focus Area	Description	Interest
1	Operations	We should be able to integrate API development into our internal SDLC without changing internal processes so that we can: ● continue using our code version control system ● continue using our existing infrastructure for CI/CD for build automation and deployment across our SDLC ● sync API definitions with the documentation	
2		We should be able to get real time statistics on traffic and operational metrics including: ● latency ● usage ● throughput so that: ● our operation teams can rationalize our backend infrastructure ● we can anticipate scaling up/down our API infrastructure ● isolate bottlenecks and fix issues	
3		We should be able to pull audit logs into our central monitoring tool so that we can rapidly respond to access and RBAC breaches.	

4		We should be able to integrate with our internal logging system(like Splunk) so that we can rationalize the API traffic with back-end traffic and requests. We would like to integrate API message logging and operational logging.	
5		We should be able to automate setup of two-way MSSL (TLS) between the API platform and target machines so that we can automate loading of certs and determine cert expiry in real-time and notify (of expiry) the responsible people within our organization.	
6		We should be able to monitor API health to rapidly respond to API traffic disruption. The health applications should enable functional monitoring, including tests for existence of specific payload information. We would like to conduct: ● health checks based on response time SLA ● health checks based on custom payload information ● health checks based target availability Failure notifications should be sent through different channels (email, Slack, and other channels we support) and to different people in the organization.	
7		The platform should support: ● federated identity through SAML ● two-factor authentication for internal API developers	

		● internal active directories for API developers	
8		The platform should support identity federation for app developers and support SAML-based IDP.	
9	Architecture and scalability	The software should be multi-tenant to ensure our distributed teams can satisfy their internal SDLC requirements.	
10		The software should be distributed across data centers so that management console user information, API consuming developer information, cache, keys, and quota, are available in multiple regions without our operational teams having to waste cycles deploying these capabilities.	
11		For an on-premises deployment, the software should be able to scale heterogeneously. We should be able to scale individual components depending on what is under load at any given point in time. Our DevOps team should be able to automate provisioning of components at will.	
12		The software must enable upgrades of runtime and non runtime components while continuing to serve APIs to consumers.	
13		We should be able to shut down different platform components to study the impact on API runtime traffic.	
14		Running performance test should not affect production environments.	

15		The platform should enable deployment of components across security zones so as to comply with our security requirements.	
16		For an on-premises deployment, we should be able to monitor all individual components of the software, so that we can isolate problems when they occur.	
17		For an on-premises deployment, we would like to understand how we could reduce the cost of bringing up new environments in our SDLC. Would adding a QA environment entail adding additional hardware to support the environment, or can we leverage existing infrastructure?	
18	DevOps	We should be able to create different roles for different teams and different team members. The platform must support granular RBAC for the API team	
19		We should be able to support audit logs for user access.	
20		All the functions available in the UI should also be available via APIs, which should be easily accessible.	
21	Multi-tenancy	Demonstrate the platform's multi-tenant capabilities by creating multiple tenants within the same deployment. Demonstrate the platform's ability to support multiple deployment environments for different tenants.	

22	Performance	Demonstrate the performance characteristics of the platform for different concurrency, different payload sizes, and different policy management scenarios. <<Detailed scenarios to be provided>>	