Migrating away from MuleSoft

This document addresses some common, high-level questions about migrating away from MuleSoft. We explain why companies are migrating away, what the new destination system's capabilities should be, recommend a migration process, and explain how Poly differs from MuleSoft. We intentionally wrote the first three sections of this document to be agnostic of the platform selected to replace MuleSoft. The fourth section is our biased, shameless plug for our platform.

We hope all readers will benefit from the first three questions, regardless of which platform is ultimately selected to replace MuleSoft. All the statements about MuleSoft are obtained from eight current customers of MuleSoft who are either migrating or preparing a plan to relocate and a few customers who have evaluated MuleSoft but ultimately decided not to adopt it. The authors, with their intimate knowledge of MuleSoft, have conducted thorough research to present this document as unbiased as possible and to purely represent the voice of these customers.

As a disclosure, I (Darko Vukovic) am a former employee of MuleSoft with no current ties to the company. Our lead Investor from DIG Ventures is Ross Mason, the founder of MuleSoft. This paper does not represent Ross's views or opinions about MuleSoft in any capacity.

Why migrate from MuleSoft?

Mulesoft emerged as a pivotal early innovator in the Integration Platform as a Service (PaaS) sector. However, as time has progressed, an increasing number of organizations have recognized significant drawbacks to using Mulesoft:

Cost of License

The loudest complaint is around the absolute cost of the licenses and significant price increases at contract renewal time. The customers we spoke with feel they must pay more for MuleSoft and get value to justify the ever-increasing cost. They are eager to look at an alternative that controls license costs and helps them reduce the overall cost of solving their integration and microservices development problems.

Pricing Model

Another common complaint is around the VCore pricing model itself. Customers claim that it's hard to predict the cost over time, and the need to purchase VCores in sets adds unnecessary spending. Customers who start gaining traction with MuleSoft find themselves with runaway costs that sometimes run ahead of the value gain. We have also heard complaints about MuleSoft demanding payment of the entire contract up-front and pushing to longer-term contracts with no options for cancellation.

Cost and Availability of Labor

While some customers who have experienced MuleSoft developers have told us that it's "straightforward to develop in MuleSoft," we have also met several customers who have been unable to build enough critical knowledge in-house to use MuleSoft effectively. This is mainly due to MuleSoft's programming/configuration language, IDE, runtime environment, and operations platform, all proprietary and specialized skills. These customers have had to rely on professional services and SIs to develop relatively simple MuleSoft applications. These Mule "experts" come with the additional overhead of contract management, margins, minimum contract sizes, retainer fees, etc.... which further exacerbates the costs of developing on MuleSoft.

Technical Alignment with Development Processes and Tooling

The more technically savvy customers with in-house development teams also desired more alignment between their core development and approach and that of MuleSoft. MuleSoft should architecturally align with their infrastructure, deployment, delivery models, source management, coding patterns and practices, and operations tooling and

processes. They feel MuleSoft is too much of an island compared to their core development infra and tools. Customers want to embrace more modern and standardized infrastructure, runtime, and development technologies.

Developer Productivity

While not a top complaint, customers also felt that there were some mundane and tedious aspects of development with MuleSoft. A few cited examples are modeling of services for which there are no box connectors (most services); dealing with customized object models in enterprise systems; the ability to log, debug, and monitor services; building multi-tenant integrations with credentials in a vault; and building/managing reusable flows which are not instantiated as a separate service (due to VCore pricing constraints).

PolyAPI

What constitutes a good replacement?

A sound replacement system must be able to implement all of the core integrations/orchestration development/API management use cases and development patterns needed by a modern enterprise. It should be optimized for modern development but must cater to legacy application integration/development. It should align well with modern development infrastructure, tooling, practices, and security standards. Its business model should be transparent, its pricing should be reasonable and predictable and more accurately represent the value delivered by the platform. Below, we outline more details about each essential requirement for a MuleSoft replacement system.

Key Integration, Service Development, API Management Use Cases

In the author's opinion, all eight patterns MUST be addressed as suitable replacements for MuleSoft. An enterprise today may only implement some of them, but in an effort to be futureproofed, we highly recommend that you evaluate a platform's ability to implement all seven patterns. It's entirely possible to select multiple platforms to replace MuleSoft, but the author's STRONG opinion is that there is a massive advantage to using only a single atomic platform. This is because integration, services development, and API management are manifestations of the same underlying capability. To be clear, there are some advantages to buying all the needed platforms from a single vendor. However, we strongly recommend that the customer seek out a SINGLE PLATFORM, not just a single vendor. We give credit to MuleSoft for understanding this and realizing the Anypoint Platform. Below are all the core patterns that the new platform, in our opinion, **must** implement. Note: some use cases may require multiple patterns as part of the solution.

Managed Connection - aka "API gateway" pattern

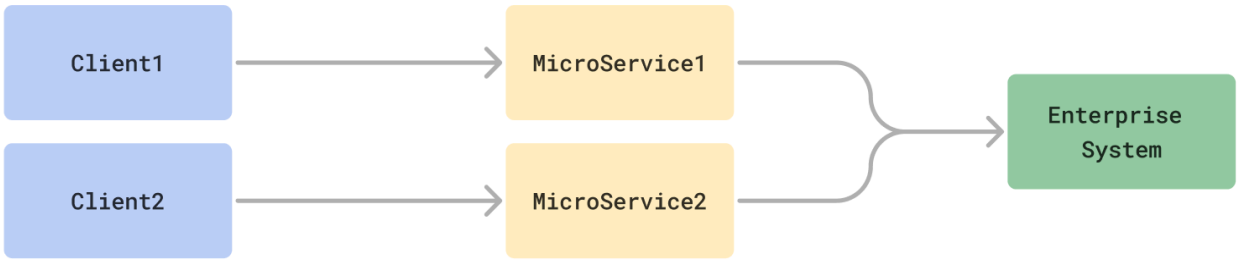
An approach to develop **loose coupling**, manage access, standardize authentication and authorization, observe usage, detect operational issues, and meter access.

Backend Orchestration - aka "orchestration" or "composite App" or "legacy modernization" pattern

They are used when a **new interface or additional logic** is needed, generally due to transformation or orchestration across multiple backend systems, enterprise-specific

business logic, or mediation of protocols/data formats. It also scopes down data access, masks fields, or programmatically inserts additional data.

Frontend MicroServices - aka "presentation APIs" or "single purpose APIs" pattern
They are **built for specific clients** needing specialized APIs to be adequately constructed and agnostic of backend systems. For example, a backend to a mobile application requires a standard data model, authorization model, resource structure, etc... to be easy to develop and maintain over time. Generally, front end developers or partner developers are not experienced and productive when working directly with most enterprise backend systems. This approach can also be used to build specific APIs for important partners where they drive the interface contract, which needs to be adhered to.

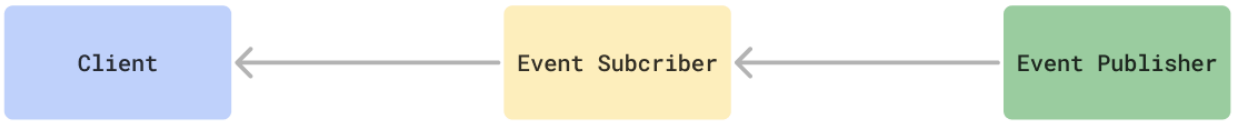


Event Stream Integration - aka “event-based sync” or “async API” pattern
These integrations are triggered by events originating in other systems, such as webhooks, Kafka messages, or GraphQL Subscriptions. These events are listened for and configured in the source system to be sent to the IPaaS, where they must be captured, acknowledged, and processed. These events come in the structure of the origin system and generally must be converted to an internal data model.

PolyAPI

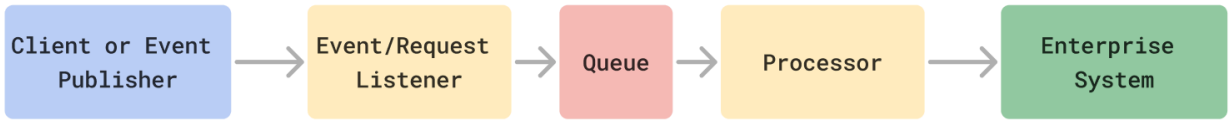
Event Subscription Pattern - aka “webhook” or “callback” pattern

Clients sometimes want realtime delivery of events from backend systems when states change. The archaic way of this is via polling. A platform must be able to provide clients a way to subscribe to published events occurring in the enterprise systems. How the platform gets the events may vary from webhooks, Kafka events, GraphQL subscriptions or other protocols. The delivery of those events by the platform to clients should ideally be both easy and consistent regardless of what the event origins within the enterprise systems are.



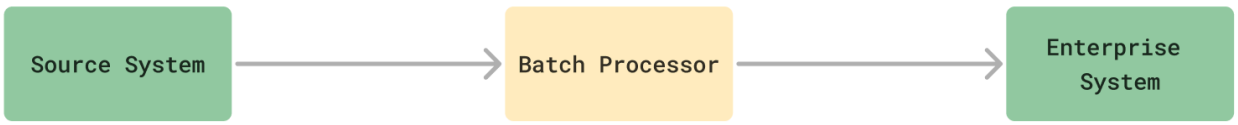
Queue Based Integration

An essential but often overlooked pattern where a queue is used within an integration. These integrations generally have an event or request listener, which is scalable to handle a mass influx of burst traffic and persist it to a scalable queuing technology. On the other side of the queue are one or more separate processors that optimize for batching to enterprise systems and self-throttling to avoid overloading downstream systems.



Batch Data Synchronization - aka “data migration” pattern

They are used for synchronizing large data sets periodically or migrating data from one system or instance to another. It is beneficial as a tool for companies with multiple instances of the same technology, when decommissioning and consolidating systems, acquiring new companies, or to implement processes that involve large data sets with latency affordances. Many times, one or both sides of these integrations are databases; the IPaaS system must be able to support this. Generally, these synchronization processes are scheduled or triggered by an API call or event and can be one or two way.



SDK and Client Proxy - aka “SDK generation” pattern
A pattern to simplify integration and application development where SDKs are provided to client developers to offer a much easier experience in building integrations in their native programming languages. This is especially helpful when the client developers are not experienced in the business domain of the enterprise APIs, the backend systems are very industry-specific and complex, and the backend systems are not well documented. One essential capability here is to develop utility functions that help client developers with bespoke orchestrations and complex non-business logic development.

Catering to Legacy Applications

Legacy applications are, by definition, legacy because the original creators have either placed them into maintenance mode or have altogether stopped investing in them. Vendors may have previously developed these systems, or they may be in-house built. It’s unrealistic to expect new vendors to provide first-class support for them on new integration platforms. However, these systems still play a vital role in operations for many companies and will continue to do so for a long time; therefore, as a prospective customer of an integration platform, it’s perfectly reasonable to demand support for integrating with and orchestrating against these systems with some affordance on the expected experience to do so. We believe that a comprehensive IPaaS must seamlessly accommodate both legacy systems and modern tools within a single platform. The approach of using separate integration systems — one for legacy systems and another for new technologies — does not constitute an acceptable solution in our opinion, as it does not generate the cost and labor savings customers deserve. This stance applies to any scenario involving multiple integration implementations.

PolyAPI

Alignment with Modern Development Infra, Tools, Processes

An IPaas should align with modern development tools, infrastructure, and processes. The degree to which it does should be used as an evaluation criterion in selecting a winner. Below are the critical categories by which a vendor should be evaluated.

Infrastructure

The entirety of the platform, runtime and management services, should be Kubernetes native, and a customer should have a choice in either running it entirely in the customer's Kubernetes environment or using it as a Platform as a Service (PaaS). If the service is not Kubernetes native, the service should at least be hybrid where the runtime is decoupled from the management service. We believe any service that is neither Kubernetes Native nor hybrid should be disqualified.

Runtime Technology

The platform runtime should be built on proven open-source technologies with a large developer community. The vendor must articulate strictly which technologies are used and in which capacity they are utilized. Ideally, the runtimes are offered in all three formats: Kubernetes native services, as standalone applications, and as a PaaS offering. In our opinion, proprietary runtime should be avoided if possible.

Infra Provisioning and Upgrades

If the customer runs the whole service within their cloud, the service should be easy to maintain and operate. There should be Helm charts, or equivalent scripts, to make deployment and upgrade easy and seamless. The application should be cloud-agnostic and easy to run on AWS, Azure, and GCP. Ideally, there should not be any cloud-specific services used. If there are, the vendor should provide a roadmap to eliminate those dependencies or a thorough justification for why they are needed. The vendor should offer an affordable, fixed bid for managing the service on behalf of the customer with a detailed list of the permissions necessary to operate.

Development and Operations Tools

Ideally, the platform should be lined up with the development tools loved by developers who have emerged as winners. Ideally, developers get to use the tools they love, and the customer benefits from more productive and happier developers. Customers will in turn experience easier recruiting and onboarding of developers for their integration projects. For development, these tools notably include, but are not limited to, VS Code, IntelliJ, Postman, NPM, PIP, Maven, and Gradle. And for operations, these tools include KubeCTL, Helm, Prometheus, Grafana, Terraform, Loki, and equivalents.

PolyAPI

Development Languages
In terms of community and enterprise adoption, the four proven winning languages are TypeScript (Node), Python, Java, and C# (.NET). The IPaaS should support all four languages to develop new runtime microservices and support client developers consuming APIs produced and managed by the platform. If all four are not supported, at least Java or Javascript should be supported, given their wide popularity within enterprises. Custom and configuration driven languages should be highly avoided. Beyond the argument of developer productivity, availability, and happiness, custom languages will face a significant disadvantage with the rise of AI. AI is trained in open-source languages and exhibits ever-increasing accuracy and competence and in our opinion will eclipse proprietary languages.

The advent of advanced tools like Github Copilot and Tabnine underscores the significant edge that proficiency in programming languages familiar to Large Language Model (LLM) AI systems offers. Developers skilled in mainstream languages such as Java and Python, as opposed to proprietary languages like Mulesoft's DataWeave, position themselves at a considerable advantage. LLMs demonstrate an exceptional understanding of these widely used languages and are continuously improving in their ability to suggest accurate and efficient code snippets, making fluency in these languages increasingly beneficial.

Source Management
All coding performed to build integrations or orchestrations should be subject to the same development procedures as regular application development. This means that a developer should be able to commit code to source management, perform code reviews in collaboration with other developers, handle merge requests, manage contributors, run linting rules, develop and run unit tests, etc... on any code they produce. At a minimum, the IPaaS should allow any programmatic or configuration-driven code to be managed with a GIT-based source management system.

Deployment Automation
When microservices are developed, a developer should be able to test them locally in their IDE and manually deploy them to dev/test environments. Beyond that, the customer should use a CI/CD pipeline to build from source and deploy to their UAT and Production environments. An IPaaS should support a seamless deployment model for all three stages: local testing, manual deployment to dev environments, and automated deployment to production.

Credentials Management
At a minimum, the IPaaS should either include an industry-recognized vault as part of their service or be able to integrate with one offered by the customer (ideally both). In our opinion, credentials used by integrations must be stored in an encrypted vault managed by a DevOps or SecOps team. Ideally, the IPaaS offers features to allow for implicit isolation of credentials across and a feature to pass credentials by reference, where a developer or the client application never handles credentials.

API First
We believe the IPaaS should have an API for every capability of the IPaaS, and the APIs should be top-notch in design and architecture. Any vendor with poor quality or missing APIs should be doubted on their ability to execute and fundamental understanding of the domain. Ideally, the vendor should also supply SDKs for their APIs.

Transparent Business Model and Predictable Pricing

We believe it's disrespectful to employ the "land and expand" business model where prices used to win an initial contract are not then honored for many years. It's one thing to list the list price and offer a heavy, stated discount, and another to provide list pricing, which increases by a double-digit percentage at renewal. Price increases should trend with inflation, and value increases in the product should be offered as add-ons for those customers who prefer to keep their costs as flat as possible.

Pricing should be usage based, or at least approximate usage or value delivered. Any metric used to set price should be intuitive and aligned with a customer's business metrics. In an ideal world, customers should be able to pick their own metrics and vendors should set the price per unit of that metric.

In our opinion, technology prices should trend lower over time as the customer base grows and the same IP is amortized over a larger customer base. Additionally, any vendor that has a track record of raising prices, of not offering new capabilities as add-ons, that took on large VC rounds, that expends a great deal on sales commissions, marketing, and facilities should be viewed cautiously as their likelihood of raising prices will be far more significant, especially in a challenging economic climate, than a vendor that does not exhibit those behaviors.

Customers should be able to model out pricing for a very long time, potentially a decade, using a number close to the Consumer Price Index (CPI) as a cost increase benchmark. Vendors should be willing to contractually include maximum price increases for existing

functionality and options to renew at pre-negotiated prices. Infrastructure costs should be made clear to customers by vendors and not represent a hidden markup. Lastly, Vendors should ideally be willing to use a metric set by the customer for establishing pricing, assuming it's easy to obtain and verify.

Ability to Execute the Migration

The selected partner should be able to own and deliver the migration from MuleSoft to their new platform. At the very least, the vendor should foster introductions to partners who can deliver migration projects. Not all customers may need this service, but for those without inhouse development teams, this offering is essential. There are a few core skills a customer should ensure the vendor or their recommended partners MUST have.

Strategic Project Planning

An ability to create project phases, goals, deliverables, and estimations. A coherent understanding of each of the steps involved and deliverables at each step. We recommend that the customer interview the delivery team and ask for a high level plan for all the phases and a detailed plan for the execution of a single phase. There are a lot of people who can do either of those things, but few who can do both really well.

Ability to Understand MuleSoft Implementations

Since the migration will be from MuleSoft, an ability to understand in detail what has been done is required. This skill will benefit the customer to ensure continued operation as well as reducing the cost to rebuild on a new platform. MuleSoft being proprietary means that the work done on MuleSoft cannot be ported over, and hence will need to be re-developed. An ideal partner can ensure continued operation of the flows, while proposing opportunities to improve the integration implementation while doing the rewrite. We believe that the largest potential cost variance will come from a partner's ability to easily understand what is already working in production and how it works. Partners that fail to do so will inherently produce lower quality work. We recommend looking for partners with either ex-MuleSoft services employees or ex-MuleSoft developers who are also proficient in the new technology.

Ability to Operate and Develop on the New Platform

Any new platform chosen will also have an operational need. We recommend that customers test the vendor or delivery partners for an understanding of what it takes to run the new platform in production, common failure points, and the ability to own the operation of the platform. This is essential as someone developing for a platform that they don’t know how to operate is a recipe for poor monitoring, troubleshooting, and incident resolution.

PolyAPI

What does the migration process look like?

There are five high-level phases we recommend in replacing MuleSoft:

1. Select a replacement
2. Replace a few representative integrations
3. Use replacement for all net new integrations
4. Migrate remaining projects
5. Decommission MuleSoft

Each of these five phases has a clear and stated goal, and they should be done sequentially to minimize disruption and cost to the customer. It's essential to remember that MuleSoft overall is a good product that serves an essential purpose, and that new products that replace it implicitly introduce a degree of risk that needs to be carefully accessed. We believe that a customer should not rush this process and that a steady, methodical approach is required.

Phase 1 - Select a Replacement

The key outcome of this phase is to identify one or more suitable replacements for MuleSoft. An acceptable outcome is simply verifying the non-existence of a suitable replacement for the customer's needs.

Using the criteria outlined in the previous section, an enterprise should assemble a team of at least a developer, enterprise architect, security architect, and business stakeholder to identify the set of vendors which will be considered as a potential replacement product. Each enterprise will have its own team composition, but we recommend that at least those four skills be represented in the evaluation team to make a good decision. This team should first assemble a list of known leaders and up-and-coming vendors. The known leaders will be documented in Analyst reports, but the up-and-coming vendors will require more work to identify.

We recommend contacting reputable VC firms, specifically those founded by Ex-Executives of the top IPaaS vendors, such as DIG Ventures, or ones that led the investment in the current winners, such as LightSpeed Ventures. Additionally, we recommend contacting leading analysts at firms like Gartner and asking them for their opinions on upcoming vendors catering to the use cases described in the previous section. Some notable analysts include Mark O'Neill, Gary Olliffe, and Max van den Berk, and the cost of the consultation will be money well spent.

We recommend that the qualification criteria outlined in the previous section be used as a benchmark for shortlists that should be invited for an RFP. We also recommend that a "Do it yourself" option is considered, while keeping in mind that DIYS alternatives generally tend to be severely underestimated in terms of cost and effort due to human biases. Additionally DIYS approaches tend to lead to additional maintainability and visibility issues as they tend to, over time, outgrow the original design which was focused on a specific problem.

In the RFP, we recommend that vendors are given a few exported MuleSoft projects, with the request to replicate the flows within the vendor's products in a demo instance. The vendor should be granted access to a UAT environment where the MuleSoft flow can be executed. In return, the vendor should share an environment where the customer can see the demo in action. Ideally, this would be a very low-cost, paid engagement so that customers can assess the vendor's ability to quickly turn around contracting, onboarding, and other business support functions.

We recommend that a vendor be judged on their ability to work off Mule projects as inputs, whether the implementation met or exceeded the capabilities of the Mule projects, how quickly the vendor was able to accomplish it and provision an environment for the customer, the vendor's ability to point to an articulate the advantages of their solution, and lastly the ability of the vendor to engage their business teams to handle contracting and invoicing. Vendors who are unwilling or unable to execute a simple exercise will either be plagued with the same organizational inefficiencies as MuleSoft or will not give the customers the proper respect they deserve.

If the customer is impressed with multiple vendors, there is likely a good business case to progress them all to the next stage. To reiterate, replacing MuleSoft is a huge decision and a big undertaking. Investing a little more in the evaluation process will be time and money well spent to further distinguish the capabilities of the promising vendors that conducted themselves well in the first phase. Generally, this phase may take 3-6 months to complete from beginning to end and should require part-time involvement from the team members conducting the RFP, plus the time to agree to terms and go through procurement for the next phase. We recommend that the customer identify the critical metric that pricing will be based on and structure a contract that includes predictable and favorable terms for the remaining phases. The customer should have a contractual option that allows them to withdraw from the contract or to mitigate the loss if the vendor's platform fails to deliver in phase 2.

Phase 2 - Replace a few representative flows

In this phase, the key is to verify in production that the selected vendor is suitable to replace MuleSoft. For each vendor chosen, the project's scope should be to implement 3-5 representative Mule projects. The internal team should supply the vendor with the representative Mule flows and an explanation of their purpose. Each of the projects should have been selected carefully so that successful implementation of the chosen projects constitutes technical proof without a doubt of the suitability of the preferred vendor as a replacement for MuleSoft.

Per the customer's preference, a production instance of the platform should be instantiated as either a PaaS tenant, managed instance, or Kubernetes deployment. Also, per the customer's preference, either an internal person or team or a team hired from the vendor should be tasked with implementing the new integrations, testing them within UAT environments, and deploying them to production. Subject to a good experience of the progression and a successful project outcome in production, with the opportunity to address any customer questions, this should constitute proof that the vendor's solution was the appropriate decision. We also recommend that you be wary of any vendor who's own professional services team cannot deliver time and cost effective integrations on their platform.

Phase 3 - Build net new services

Assuming a successful phase 2, in phase 3 the customer should stop investing in developing new integrations on MuleSoft. All new developments should happen on the new platform. A successful outcome of this phase should be that no new investments are made in development onMuleSoft. There may still be maintenance, bug fixes, etc... which are needed to keep the current implementations healthy. There could be cases where, for some reason or another, new integrations or services must be developed on MuleSoft for potentially valid reasons like: team capacity, a defect in the new platform that is waiting to be addressed, or a missing feature. These are understandable, and the customer should understand that this can happen to a degree, especially if the customer chooses an up-and-coming platform. However, each occurrence should be taken as a severe warning and add weight to potentially abandoning the migration. The customer should be cautious not to fall into the sunk cost fallacy and brute force the migration to continue despite clear warning signs.

If successful, the new platform should demonstrate its ability as a significantly superior solution, and the consensus by any generally unbiased team member should be that they "don't want to go back to the old way of doing things." This stage could take as long as

months or could take years. There is no sense in rushing this phase. We recommend that the cost licensing and operating MuleSoft be the primary forcing function to move faster and that the customer does not create a synthetic motivation to move faster than needed.

Phase 4 - Migration from remaining MuleSoft projects

At this stage, a clear decision should be made to migrate any remaining MuleSoft projects to the new platform. This should be a focused effort and executed as fast as possible, ideally by a dedicated person or team to ensure high productivity from being in the zone. There is a major productivity benefit from being in the zone when replicating so much work, and there is a cost to running two platforms in parallel. A company may consider outsourcing this work to accelerate MuleSoft's decommission, as it will likely be an excellent ROI. Hiring someone experienced with MuleSoft and proficient in the programming language of the new platform may take much work based on the platform chosen. Ideally, the new platform requires only knowledge of Java, TypeScript, Python, or C#, in which case finding developers within the customer organization may be possible. We recommend iteratively building and launching the replacements to de-risk the migration further as much as possible. This stage will take a variable amount of time, depending on the number of flows to be re-written. However, it should take an estimated two days of development and one day to deploy per Mule flow, assuming the developer is proficient in both MuleSoft and in the new platform.

Phase 5 - Decommissioning MuleSoft

The final stage will be to turn off all the MuleSoft applications and cancel or notify MuleSoft that the customer will not renew. There will be time left on the contract, and we recommend keeping the MuleSoft applications running idle for this duration as a backup to issues discovered in the new integrations. When MuleSoft is finally turned off, we recommend that a party be organized to celebrate and recognize all the people who did the work, as it likely was much work and likely will pay significant dividends for the company.

PolyAPI

How is Poly different from MuleSoft?

We wrote the previous three sections as neutral and unbiased as possible. At this point, we will let our biases come out about why Poly is the best alternative. We sincerely believe each of these points to be true in full and it's why we built Poly to begin with and why DIG Ventures invested in Poly. Still, many of the points are subjective, so please keep in mind that our opinions may not be shared with everyone equally, and we always would love to hear views that counter ours so that we can grow our understanding of the domain landscape.

Here, we aim to outline the key differences between MuleSoft and Poly and list the key benefits of Poly (if any) as we see it.

Consideration	**MuleSoft**	**Poly**	**Poly Key Benefit**
Pricing Model	VCore (unpredictable)	Customer Defined Metric (fixed/known)	Metric lines up with customer definition of value delivered.
Cost of License	"High"	A fraction	Use of OSS and elimination of marketing, sales commissions and facilities are all passed through to customers.
Cost/Availability of Labor	"High"	"Normal"	Developers proficient in TypeScript, Python, Java are productive from day one. No specialized training or "experts".
Dev Tools	Custom	VSCode, Postman, NPM, PIP, Maven	Leveraging standardized developer tools improves productivity, happiness, recruiting and onboarding.
Dev Languages	Custom + Java	TypeScript, Python, Java	Higher reach of developers, better alignment with internal development. C# (.NET) is on the roadmap.
Source Management	Source Persistence + Sharing	Full Benefits	Poly being native TS, Py, Java code allows for code reviews and linting in addition to committing & Sharing like MuleSoft.
Test Automation	Custom	Standard	MuleSoft offers an proprietary automated testing solution while Poly code can be tested with any customer preferred test automation solution.

Deployment Automation	Standard	Standard	Effectively equivalent.
API First	1st Class	1st Class	Both platforms offer comprehensive APIs. MuleSoft also offers a UI while Poly is “headless” at the time of this writing.
Credential Management	Cred Store + Custom	Out of Box HashiCorp Vault	Poly offers first class management and storage of credentials in a vault while MuleSoft requires integration to a Vault. Poly credentials can be used by reference and tenancy is implicit in architecture.
Runtime Engine	Mule (custom)	Node, Python, Java (JVM)	Poly uses open source proven runtimes, which affords flexibility in where the loads are run.
Runtime Infra	AWS (CloudHub) Self Hosted Mule	AWS, Azure, GCP Self Hosted Kubernetes or Node, Python, Java runtimes	Poly is Kubernetes Native and we use KNative for running flows. As a PaaS we offer AWS, GCP, and Azure. Self hosted customers can run loads within Kubernetes using KNative or as standalone Node, Python, Java Apps.
Management Service Infra	AWS or Self-Hosting in other Clouds	Kubernetes Native or as a PaaS in AWS, Azure, GCP	Being Kubernetes Native, Poly’s whole platform can be hosted by customers within their own Kubernetes Cluster or can be hosted in AWS, Azure, GCP as a PaaS.
Legacy Apps	Lots of Connectors	No Connectors, customer connectivity.	Poly falls short compared to MuleSoft when using legacy transports to connect to legacy systems. With Poly, developers can custom build functions which abstract the legacy systems, but the connectivity development is up to the customer to implement.

MuleSoft is a very good platform and has emerged as the top vendor in this domain for several years. We believe that it’s had a good run but that a new paradigm to integration development is now here. One that can make developers more productive, happy and will also save money for enterprises of all sizes. We believe Poly is that product.

Conclusion

We hope this paper provides a high-level understanding of why customers are interested in moving away from MuleSoft and the fundamental patterns any vendor needs to support to be a suitable contender for replacing MuleSoft. We then outlined our recommendations for the high-level phases, their exit condition, and estimates on how long it should take, assuming the new vendor was a suitable replacement. Lastly, we compared Poly to MuleSoft in approach, architecture, and business model. We sincerely hope to be a contender if your organization decides to evaluate alternatives. We also would appreciate any feedback about this writing to ensure we are as accurate about this topic as possible, so please do not hesitate to reach out at info@polyapi.io.

Thank you,

Darko Vukovic
CEO & Founder of PolyAPI