MuleSoft

WHITEPAPER

Mule 4 performance

Table of Contents

Table of Contents

Introduction

This whitepaper presents a high level overview of the performance of Mule, the runtime engine of Anypoint Platform™. We'll elaborate why integration developers choose Mule runtime over other integration development tools, and provide visibility into benchmarking numbers based on use cases across the MuleSoft customer base. We'll also present metrics for standalone use cases — such as applying policies via API gateways — and metrics for more complex integrated use cases such as for a batch processing application handling Salesforce account records.

The MuleSoft Performance Engineering team collected the numbers presented in this report — you can find details on the methodology in the Appendix. The metrics are collected from sample customer applications in a controlled environment and they should not be explicitly used to size deployments because environment conditions can vary greatly across deployments. If you need help with Mule application sizing, we recommend getting in touch with a MuleSoft representative.

Why Mule 4?

In today's world, the demand is for speed at scale. Everyone is looking for real-time responses from their applications. They want data to be made available now and they want to access it from anywhere, at any scale. Over the past few years, we have seen back-end architectures evolve to cater to front-end responsiveness and drive elasticity and resiliency through event-driven microservices. And, we have seen reporting requirements evolve from entirely retrospective analyses to include real-time predictive analytics made possible by event stream processing.

Mule 4 is the latest version of the Mule runtime engine and is built to satisfy speed and scale needs. The runtime engine adopts a reactive programming model to facilitate non-blocking execution of common domains in integration development such as connectivity, transformation, validation, error handling, and security. Mule developers declaratively define a set of flows within an application, and each flow is composed of a chain of event processors which process data passing through the flow. Mule 4's ability to facilitate non-blocking execution of the event processors allows it to easily scale the amount of events it can handle concurrently.

Mule 4 also features automatic thread pool management to allow event processors to execute in a way that optimizes resource utilization. Threads are dynamically assigned to the application in a way that maximizes concurrency given the application logic. And, memory and disk are utilized in a way that facilitates stream processing of larger than memory payloads. These innovations are the reason the Mule runtime has led, and continues to lead, the integration market when designing for speed and scale.

How to read the benchmarking numbers

The charts used in this report compare different scenarios by displaying the throughput in transactions per second (TPS) and the percentage of CPU usage across varied deployments.

TPS is represented in bars on the Y axis and the CPU usage in trend lines that are connected by the different executions distributed in the X axis, ordered from lowest to highest according to the concurrency or the size of data processed. Three deployments are tested using 0.1 vCore, 1 vCore, and 4 vCore instances. vCore is defined as the number of CPU cores available to a given deployment, and is the unit of compute capacity used in CloudHub, MuleSoft's iPaaS offering. For representing fractional vCores like the 0.1 vCore deployment, CloudHub allocates a limited amount of CPU, I/O, and memory ideal for simpler use cases such as an HTTP proxy or integration application with minimal transformation. In this report, the 0.1 vCore deployment is benchmarked as an AWS EC2 T3 burstable instance, and the throughput numbers presented are under burst conditions. Burstable instance types allow applications to better handle temporary spikes in load by allocating more CPU during periods of high use. The numbers presented in this report are to show a maximum possible throughput for 0.1 vCore applications, not the baseline level of performance. If you are considering sizing applications for fractional vCore sizes, we recommend getting in contact with a MuleSoft representative for assistance.

For each scenario the definition of transaction is slightly different, based on the type of application:

- → **HTTP Proxy, API Gateway, and REST API implementation (APIkit):** A transaction is an HTTP request that passes through the application.
- → **Batch:** In the batch application, each transaction corresponds to a record that is processed from the input file.
- → **Data Transformation (DataWeave):** Each transaction corresponds to one HTTP request with a payload that is transformed into the target format.
- → **Messaging (JMS Connector):** A transaction occurs when a message is successfully transferred from a source queue to a destination queue.

An important term in this report is knee point, which refers to the point at which the throughput of the scenario reaches the maximum value, usually due to the exhaustion of some resource, such as the CPU.

01 Standalone use cases

01. Use case — HTTP Proxy

An application that acts as a proxy is one of the most common use cases for Mule. In this scenario, the application attends to GET requests that pass to a separately hosted Vertx server. This backend server adds a 70 milliseconds delay to emulate more realistic conditions and returns a payload of a given size to the Mule HTTP proxy application.

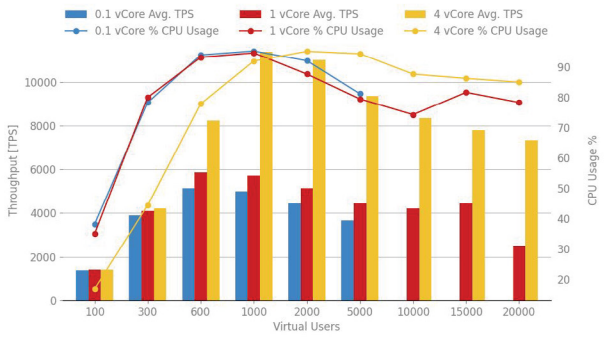
The following charts show the results as the concurrency increases when the payload size requested is 1KB.

HTTP Proxy

As shown in the chart below, for 0.1 vCore and 1 vCore executions, the knee point is 600 virtual users, while 4 vCore reaches the knee point at 1000 virtual users. The three vCore targets show how the CPU usage increases as more virtual users are added, until the knee point is reached. After that, the CPU starts dropping slowly due to garbage collection (GC) pauses. This indicates that this particular scenario is CPU-bound.

For 0.1 vCore, executions were limited to 5000 virtual users because at this point, in such small instances, the resources are very saturated. Running under unstable conditions might lead to unpredictable results, such as the application or the instance crashing.

MULE 4.3.0 - HTTP PROXY 1KB AVG. TPS AND % CPU USAGE PER VCORES



HTTP proxy GET with 1 KB payload for 0.1, 1, and 4 vCores - average throughput and % CPU usage

HTTP Proxy Non-Repeatable-Streaming

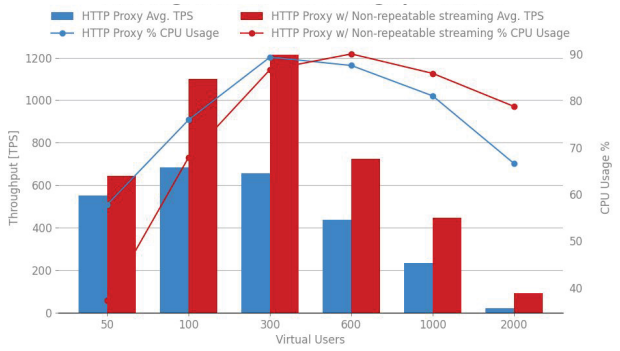
If these types of proxy applications regularly expect a payload that is 500 KB or longer, then they should be configured to use non-repeatable-streaming as part of the HTTP component that handles the payload. For this scenario, we add a new Mule 4 HTTP proxy application that adds this non-repeatable-streaming approach to its HTTP components. The Mule 3 application stays the same because it is non-repeatable-streaming by default.

The following chart shows the results for Mule 4.3.0, comparing both HTTP proxy applications --with and without non-repeatable-streaming. The requested payload size is 1MB.

The non-repeatable-streaming approach adds an 84% improvement at the knee point, which is 300 virtual users.

CPU-wise, non-repeatable-streaming shows less average usage in virtual users groups before the knee point. At the knee point the difference is 2%, while at 50 virtual users, the difference is 36%. After the knee point, the average CPU usage is higher but is a good indicator of resource usage because the TPS is better than the traditional proxy application. Now we can see how the CPU usage continues increasing for non-repeatable-streaming after the knee point is reached, which indicates that the non-repeatable-streaming proxy application is less CPU-bound than the traditional proxy application.

MULE 4.3.0 — HTTP PROXY VS. HTTP PROXY W/ NON-REPEATABLE STREAMING



HTTP proxy and HTTP proxy no-repeatable streaming with 1 MB payload performance for 4.3.0 version with 4 vCores - average throughput and % CPU usage

02. Use case — API Gateway

API Gateway offers the possibility of adding specific policies on top of the HTTP proxy, making it another common scenario for MuleSoft's customers.

Each scenario creates an API instance with API Manager and installs the policy with the specified configuration.The policies tested for these scenarios were:

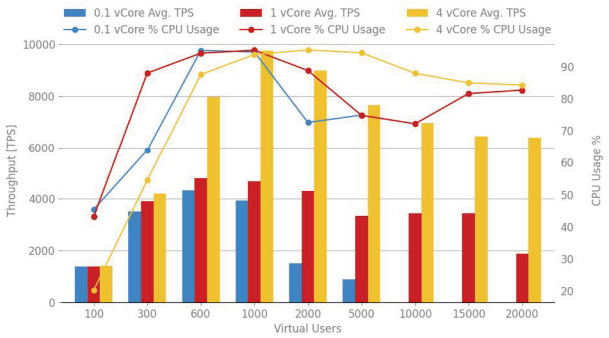
→ **Client ID Enforcement:** One scenario with the client id and client secret passed by the query params on the request.

→ **Rate Limit** (with and without X-RateLimit headers[default configuration]): Two scenarios configured for a maximum of 80,000 requests in a time period of 6 seconds, but one with the headers exposed and other with the default configuration (without headers).

Client ID enforcement

As shown in the next chart, the Client ID Enforcement policy is a CPU bound scenario because in all cases, the application CPU usage is maxed out at its knee point. In the 4 vCores scenario, Mule 4 reaches the knee point at around 1,000 virtual users. For 0.1 and 1 vCores, the knee point is reached at 600 virtual users and only performs 54.26% and 60.26% of the virtual users that 4 vCores can manage.

MULE 4.3.0 — CLIENT ID ENFORCEMENT 1KB AVG. TPS AND % CPU USAGE PER VCORES



Client Id enforcement API gateway policy (GET) with 1KB payload for 0.1, 1 and 4 vCores - average throughput and % CPU usage

Rate limit

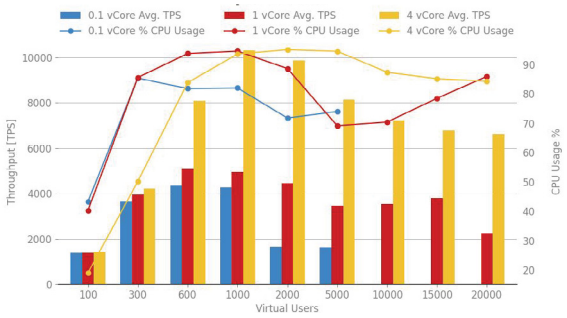
The rate limit policy has two configurations that involve the headers. Both scenarios are CPU-bound because, in all cases, the knee point is reached when the CPU usage is close to 100%.

RATE LIMIT WITHOUT HEADERS: DEFAULT CONFIGURATION

As shown in the next chart, in the 4 vCores scenario Mule 4 reaches the knee point at around 1,000 virtual users. For 0.1 and 1 vCore, the knee point is reached at 600

virtual users and only performs 42.03% and 49.40% of what 4 vCore can manage.

MULE 4.3.0 — RATELIMIT W/O HEADERS 1KB AVG. TPS AND % CPU USAGE PER VCORES

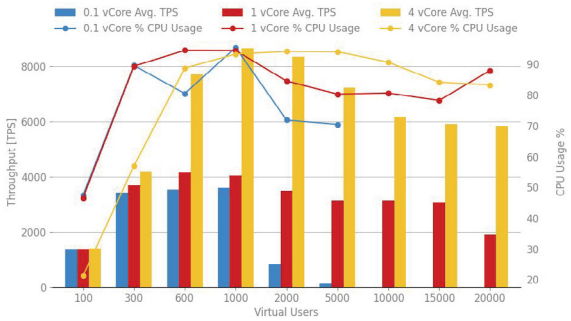


Rate Limit without headers API Gateway policy (GET) with 1KB payload for 0.1, 1 and 4 vCores - Average Throughput and % CPU Usage

RATE LIMIT WITH HEADERS

As shown in the next chart, in the 4 vCores scenario Mule 4 reaches the knee point at around 1,000 virtual users. For 0.1 and 1 vCore, the knee point is reached at 600 virtual users and only performs 45.92% and 54% of what 4 vCores can manage.

MULE 4.3.0 — RATELIMIT W/ HEADERS 1KB AVG. TPS AND % CPU USAGE PER VCORES

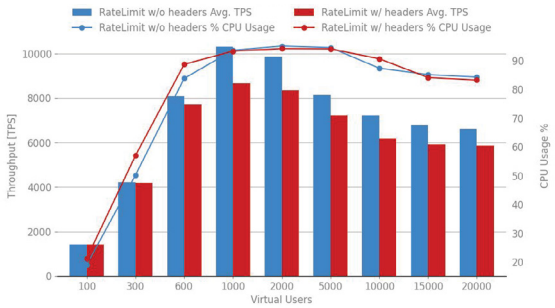


Rate limit with headers API Gateway policy (GET) with 1KB payload for 0.1, 1 and 4 vCores - average Throughput and % CPU usage

RATE LIMIT — COMPARING HEADERS VERSUS NO HEADERS INJECTION

One of the improvements in the Rate Limit policy is that the headers with information about the current status of the limits are disabled by default as a security enhancement, which limits the exposure of the usage information of the API. When the header injection is enabled, the performance drops significantly as shown in the chart below, where the throughput with no headers injections is about 12.21% and 19.03% better for 600 and 1,000 virtual users respectively. Also, with no headers injections, the performance is close to Mule 3, with a 0.83% improvement.

MULE 4.3.0 — RATELIMIT W/ HEADERS VS RATELIMIT W/O HEADERS



Rate Limit with and without headers API gateway policy (GET) with 1KB payload for 4.3.0 version with 4 vCores - average throughput and % CPU usage

→ **Note:** It is worth noticing that with both policy configurations, the 0.1 vCore instance drops the CPU usage after 1,000 virtual users, which suggests that 0.1 vCore is not adequate for high concurrency applications. Also note that the application fails from 5,000 virtual users onwards.

03. Use case — REST API implementation (APIkit)

REST APIs have emerged as the dominant architectural style for APIs, and implementing them is easier when using the APIkit Mule plugin.

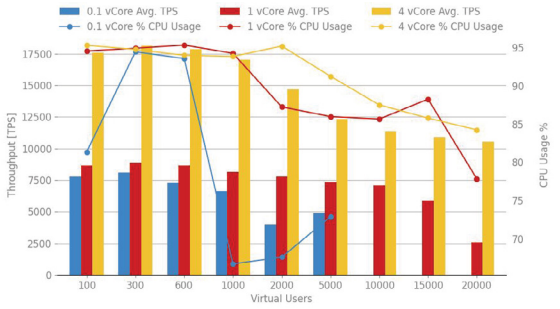
For this scenario, we have two versions of the same application, one which applies validations on the received payload and another which skips those validations. Both applications attend POST requests that emulate the creation of a given record. In contrast to the HTTP Proxy scenario, the application doesn't interact with any backend-server, then no delay is added either.

The following charts show the results as the concurrency increases when the payload size is 1KB. For 0.1 vCore, executions were limited to 5000 virtual users because at this point, in such small instances, the resources are very saturated. Running under unstable conditions might lead to unpredictable results, such as the application or the instance crashing.

As shown in both charts, for all vCores, the knee point is 300 virtual users. At the knee point, 0.1 and 1 vCore performs a 55% and 51% of what 4 vCores can manage. For 0.1 vCore, after 1000 virtual users, it starts showing unstable behavior due to back pressure mechanisms but still maintains a fair amount of TPS.

APIkit with Payload Validations

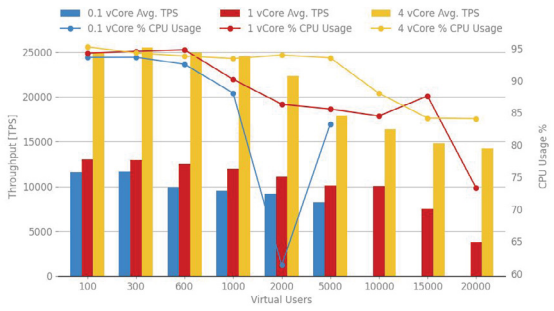
MULE 4.3.0 — API KIT W/ VALIDATIONS AVG. TPS AND % CPU USAGE PER VCORES



APIkit POST with validations enabled. 1 KB payload for 0.1, 1, and 4 vCores - average throughput and % CPU usageEped

APIkit without Validations

MULE 4.3.0 — API KIT W/O VALIDATIONS AVG. TPS AND % CPU USAGE PER VCORES



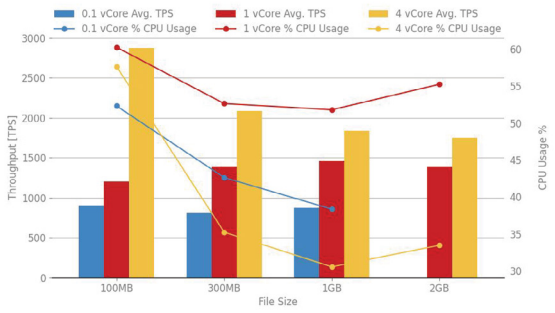
APIkit POST with validations disabled.1 KB payload for 0.1, 1, and 4 vCores - average throughput and % CPU usage

04. Use case — batch

The batch component in Mule allows users to process records in batches, performs actions upon each record, reports the results, and potentially pushes the processed output to other systems or queues.

For this report, a complex ETL scenario was chosen for benchmark purposes. To make the benchmark result realistic, the workflow was modeled following a common customer setup with three Batch Step components. The input payload is a CSV file that is read, and a DataWeave transformation to a POJO is applied for each record. The second step has an I/O operation that writes part of the payload to a file. In the last step, the records are inserted with a bulk operation in groups of 100 to a MySQL Database.

MULE 4.3.0 — BATCH 1KB AVG. TPS AND % CPU USAGE PER VCORES



Batch performance for 0.1, 1 and 4 vCores - average throughput and % CPU usage

It is important to understand that the performance of a batch application will be directly affected by the combinations of connectors and components used in each of its steps.

Batch

The next chart shows the results of the test as the file size increases. It compares CPU usage and average throughput in machines with 0.1, 1, and 4 vCores.

As shown in the figure, the application performance improves about 3 times when the resources of the instance are increased from 0.1 to 4 vCores. However, in this type of application, it is important to understand the I/O usage as one of the factors that limits the performance: as the amount of data processed is bigger, the I/O activity grows higher, causing a drop in the CPU usage.

For the 0.1 vCore instance type, note that a 2GB payload cannot be processed. The record size has a huge impact on memory usage (500MB heap) because batch requires having enough available memory to move records from persistent storage into memory and to process them in parallel with multiple threads.

05. Use case — data transformation (DataWeave)

DataWeave is the MuleSoft expression language for accessing and transforming data that travels through a Mule application.

The tests covered the most used transformations executing the following flow:

1. **An HTTP Listener** receives a payload in the source format.
2. **DataWeave** transforms the input payload into the target format.
3. **The payload** is returned.

The three transformations selected were:

→ **CSV to JSON**

→ **JSON to POJO**

→ **XML to JSON**

Since DataWeave transformations are CPU-intensive operations, the knee point is expected to be reached earlier. Therefore, these scenarios were tested with fewer virtual users compared to the other scenarios, such as the proxy that had executions up to 15,000 and 20,000 virtual users.

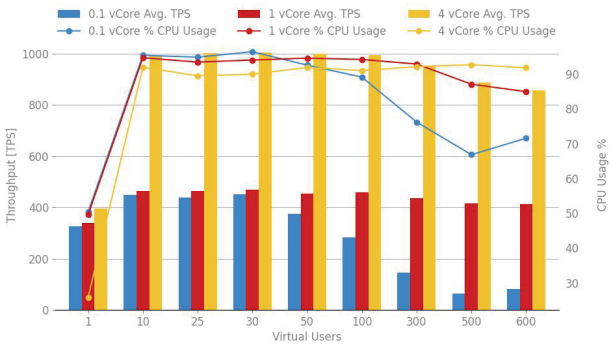
For all the transformations tested, the maximum throughput is also achieved at 10 virtual users for all the machine configurations: 0.1, 1, and 4 vCores. The throughput scalability ratio from 1 vCore to 4 vCores is about 2x, which is approximately a 0.25x increase per core.

CSV to JSON

As expected, this transformation shows clearly that DataWeave is a CPU-bound application because the maximum throughput is reached quickly at 10 virtual users for all the machine configurations: 0.1, 1, and 4 vCores.

As explained in the HTTP Proxy scenario, moving from 0.1 to 1 vCore does not have much impact on this transformation for low-concurrency levels, but when virtual users exceeded 25,

MULE 4.3.0 — CSV TO JSON AVG. TPS AND % CPU USAGE PER VCORES



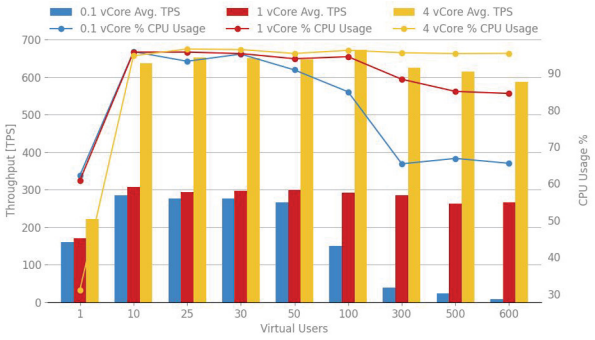
DataWeave (csv2json) performance for 0.1, 1 and 4 vCores - average throughput and % CPU usage

the 1-vCore instance kept the throughput stable, while 0.1 got degraded and sometimes left the machine in an unstable state.

By contrast, the machine with 4 vCores delivered about twice the TPS of the 1 vCore, which shows how this transformation scales when more CPU resources are added.

JSON to POJO

MULE 4.3.0 — JSON TO POJOAVG. TPS AND % CPU USAGE PER VCORES



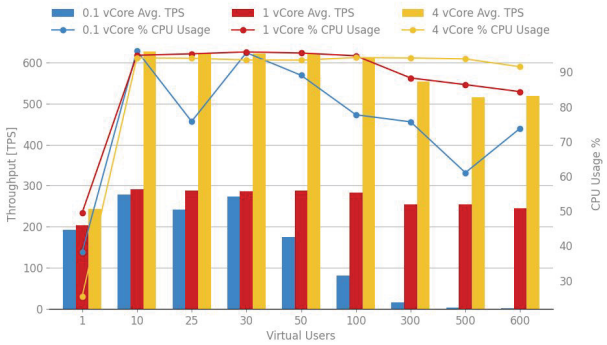
DataWeave (json2pojo) performance for 0.1, 1 and 4 vCores - average throughput and % CPU usage

The transformation shows a similar behavior as the previous one: The knee point is 10 virtual users in all instances, and 4 vCPUs delivers about twice the TPS of the other instances.

XML to JSON

Following the same pattern of DataWeave transformations, this scenario has its knee point at 10 virtual threads with similar behavior.0.1 and 1 vCore instances show a difference on high concurrency executions, and the 4 vCore instances deliver about twice the throughput of the other instances.

MULE 4.3.0 — XML TO JSON AVG. TPS AND % CPU USAGE PER VCORES



DataWeave (xml2json) performance for 0.1, 1 and 4 vCores - average throughput and % CPU usage

06. Use case — Messaging (JMS Connector)

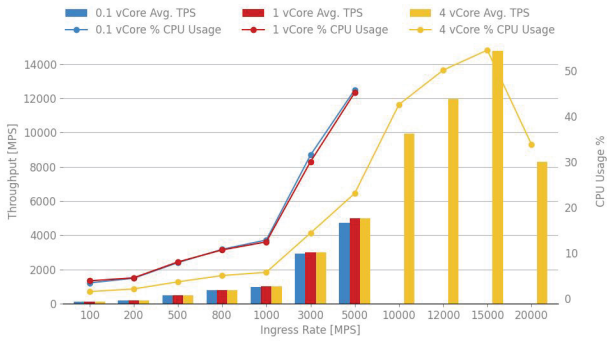
The JMS (Java Message Service) Connector enables the different components of an application to communicate through the exchange of messages, in an asynchronous ways. The JMS Bridge implementation is a software architecture composed of two queues, that integrate an ActiveMQ Service Provider. The Runtime performs the exchange of messages using four producers and four consumers. The source queue is populated by a number of JMeter threads that are created proportionally to the number of messages to be transferred. The throughput of the application is measured in messages per second transferred from a source queue to a destination queue or MPS. This metric is captured at the moment the messages enter the destination queue.

For benchmark purposes, the number of messages to be transferred per second go from 100 up to 20,000 messages. A JMS bridge implementation was tested along with a JMS bridge transactional implementation (or JMS bridge XA for brevity). The XA (eXtended Architecture) is a Two-Phase Commit protocol - 2PC - that provides atomicity, consistency, isolation, and durability properties for distributed transaction processing. Therefore, for the JMS Bridge XA implementation, this standard helps guarantee completion in the transference of a message, which prevents losses in the process. In this type of transaction, a message is retained in the source queue until a confirmation of a successful receipt is submitted. The tests performed aim to determine 1) the application's behavior when it is exposed to increasing stress and 2) to validate the impact of the transactional implementation over the performance of JMS Bridge.

JMS Bridge

The graph below shows the JMS Bridge application performance when tested with three different AWS instance types, namely 4v Cores, 1v Core, and 0.1v Core. The x-axis displays the number of messages per second that were delivered from the source queue in a specific period of time, while the y-axis displays the measured average throughput

MULE 4.3.0 — JMS BRIDGE AVG. TPS AND % CPU USAGE PER VCORES



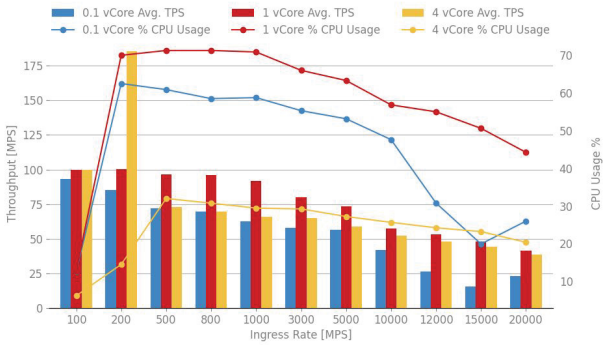
JMS Bridge performance for 0.1, 1 and 4 vCores - average throughput and % CPU usage

of the application, that is, the number of messages per second successfully transferred to the destination queue. The 4v Cores architecture displays a rising tendency, reaching its knee point at 15,000 messages with a completion rate above 98% and a 55% CPU usage. As displayed in the graph, at 5,000 MPS, the 4 vCore architecture uses up to 50% less CPU than its counterparts. For ingress rates higher than 5,000 MPS, the 0.1vCore and 1vCore architectures cannot keep up with the load, and the application becomes highly unstable, which severely affects the throughput. The identified behavior reveals an architectural limitation for the JMS Bridge application to escalate: Transfers that surpass 5,000 messages per second for 0.1 vCore and 1 vCore lead to a throughput instability and therefore untrustworthy outcomes.

JMS Bridge XA

The next graph shows the JMS Bridge XA application performance for Runtime 4.3.0 when tested with three different AWS instance types, namely 4vCores, 1vCores, and 0.1vCores. When the Two-Phase Commit protocol is introduced, the knee point is reached at an ingress rate value of 200 MPS, which is significantly lower from its JMS bridge counterpart. At knee point, the completion rate is 93% with 4 vCores. This throughput is accompanied by a 15% CPU usage. As displayed in the graph, restricting the CPU resources drastically affects the throughput of the application, which drops to 50% for 1vCores and to 42% for 0.1vCores. This throughput diminishment for 1 vCores and 0.1 vCores is attached to a significant CPU usage increment in both cases. Nonetheless, when comparing 0.1 vCores to 1 vCores, the latter shows an improvement of 17.6% with respect to the former.

MULE 4.3.0 — JMS BRIDGE XA AVG. TPS AND % CPU USAGE PER VCORES



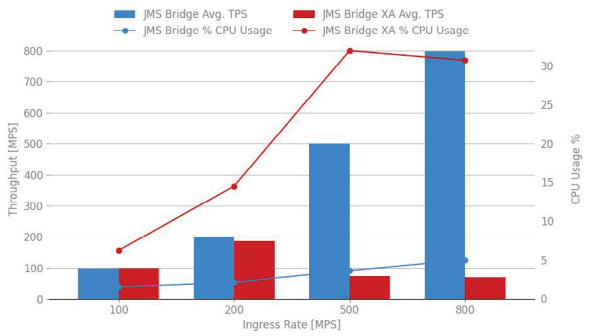
JMS Bridge XA performance for 0.1, 1 and 4 vCores - average throughput and % CPU usage

JMS Bridge vs. JMS Bridge XA

The following chart displays the average throughput of both JMS Bridge and JMS Bridge XA when exposed to an increasing number of messages per second to be transferred. The side-by-side bar comparison evidences a conclusive performance degradation when Extended Architecture Transactions are introduced. The JMS Bridge XA implementation reaches the knee point at 200 number of messages, 98% below its counterpart with an average successful TPS of 93%.. These results expose a conundrum when it comes to two desirable properties, such as reliability and performance. Though Extended Architecture Transactions guarantee the integrity and completion of the transaction operation, the benefit comes with the tradeoff of a major throughput degradation.

It is outside the scope of this report to discuss the number of strategies that can be implemented to overcome these limitations, which are specific to the XA protocol and not the runtime engine. However, it is important to highlight that there are other approaches that can ensure integrity without compromising performance, such as JMS queues with Dead Letter Queues, among others.

MULE 4.3.0 — JMS BRIDGE VS. JMS BRIDGE XA AVG. TPS AND % CPU USAGE PER VCORES



JMS Bridge vs JMS Bridge XA performance comparison for 4.3.0 version with 4 vCores - average throughput and % CPU usage

02 Complex customer scenarios

01. Batch Salesforce processing (customer app)

02. Streaming to Kafka (customer app)

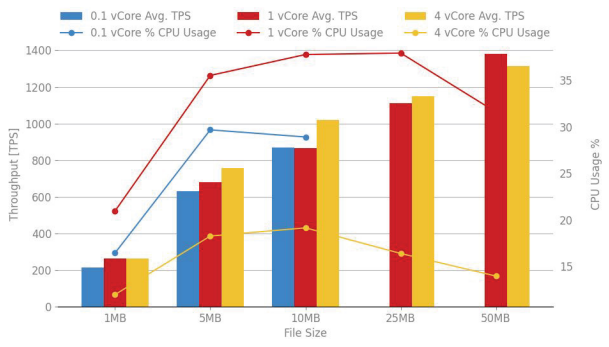
01. Batch Salesforce processing (customer app)

For benchmark purposes, a complex template was chosen from a customer setup that allows users to insert records on a Salesforce account in batches.

This application also integrates the APIkit, Choice, and VM components. The workflow is triggered by a request with the file name. Then a subflow executes a DataWeave transformation to convert the CSV input file into a Java payload. Payload is then fed to the Batch Step, which splits the data into groups of 50 records for the Upsert Salesforce operation. After batch processing is complete, the Publish VM queue receives the data, and the subflow finishes. The VM Consumer queue waits for the response, and a Choice Flow validates if the process has failed records or not. It is important to understand that the performance of the application will be directly affected by the combinations of connectors and components used in each of its steps and flows. Performance is also bound to the Salesforce account configuration. The performance results measure the Batch process with the Salesforce upsert process, and upsert process is limited for the Salesforce account storage. This Salesforce account was configured as a developer account with a limit of 5MB data storage and 20MB file storage.

As shown in the next chart, the 4 vCores scenario does not reach the knee point because one of the limits of the application is the Salesforce account, which throws the **STORAGE_LIMIT_EXCEEDED** error when the number of records exceeds the data storage. For 1 and 0.1 vCore, the throughput presents a degradation of 9% and 20% against 4 vCores, and the 0.1 vCore does not support a payload larger than 10MB. However, in this type of application, which uses the batch component, it is important to understand the I/O usage as a variable that limits the performance: as the amount of data processed gets larger, the IO activity is higher, which causes a drop in the CPU usage.

MULE 4.3.0 — BATCH SFDC CUSTOMER APP 1KB AVG. TPS AND % CPU USAGE



Batch Salesforce performance for 0.1, 1 and 4 vCores - average throughput and % CPU usage

02. Streaming to Kafka (customer app)

The Anypoint Connector for Apache Kafka (Kafka Connector) enables you to interact with the Apache Kafka messaging system and achieve seamless integration between your Mule application and an Apache Kafka cluster, using Mule.

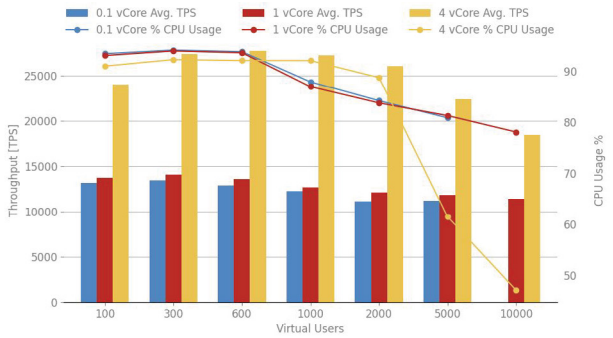
This customer application allows users to insert a message record into a topic in the Kafka cluster, which acts as a backend server. In this scenario, the application attends to GET requests. Those requests are passed to a separately hosted Kafka server (deployed in a Docker container) that inserts a message of a given size.

The following chart shows the results of different thread groups when the requested payload size (message record size) is <1 KB.

The next graph shows the Kafka application's performance. The 4 vCore architecture displays a rising tendency, reaching its knee point at 1000 virtual users with a completion throughput rate above ~27K TPS and a 92% CPU usage capacity. The 1 vCore and 0.1 vCore architecture display a constant tendency, reaching its knee point at 300 virtual users with a completion throughput rate above ~13K TPS and a 94% CPU usage capacity.

The throughput scalability ratio from 1 vCore to 4 vCores is ~2x, which is approximately a 0.25x increase per core.

MULE 4.3.0 — KAFKA PUBLISHER AVG. TPS AND % CPU USAGE PER VCORES

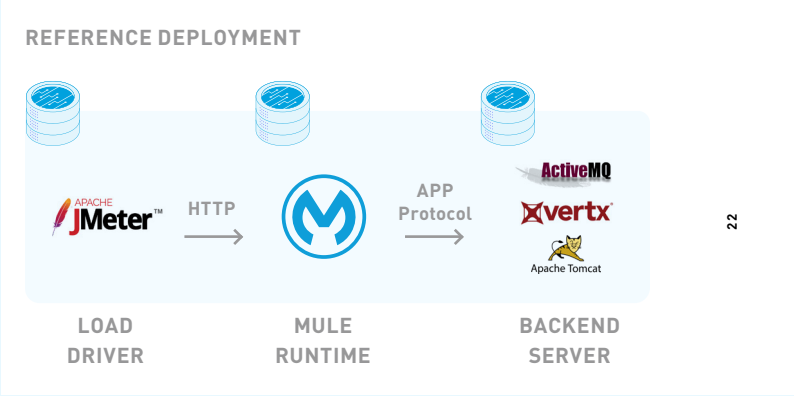


Kafka performance for 0.1, 1 and 4 vCores - average throughput and % CPU usage

Appendix

Most of the applications tested had a setup running in AWS with a structure similar to the image below:

A load driver running Apache JMeter generates load against a Mule deployed in a separate instance. In cases when a backend is required, for example, to retrieve a payload, back-end is deployed in a third instance that is used by Mule. The backend server used for HTTP Proxy and API Gateway was Eclipse VertX; for the batch scenario, it was a MySQL database running in a Docker container.



Mule runtime engine ran in three types of AWS instances to emulate different environments (see details in the table below, including the heap assigned to the JVM process of the Runtime).

The operating system of these instances was Red Hat Enterprise version 7.6. Mule runtime running over the Java 8, Oracle Hotspot version. Additionally, some of the tests were also run over Adopt OpenJDK 8 Hotspot to confirm that the results are similar in both versions.

Environment (to emulate)	AWS instance	Shared	vCPU - Base	Memory (GB)	JVM Heap
On-Premise	C5N.XLARGE	YES	4 VCPU	10.5	2 GB
On-Premise Backend Server	C5N.XLARGE	YES	4 VCPU	10.5	2 GB
On-Premise Load Client	C5N.2XLARGE	YES	8 VCPU	21	16 GB
Cloud-hub 1 vCore	T3.MEDIUM	YES	2 VCPU	4	2 GB
Cloud-hub 0.1 vCore	T3.MICRO	YES	2 VCPU	1	500 MB

MULESOFT, A SALESFORCE COMPANY

MuleSoft, the world's #1 integration and API platform, makes it easy to connect data from any system — no matter where it resides — to create connected experiences, faster. Thousands of organizations across industries rely on MuleSoft to realize speed, agility and innovation at scale. By integrating systems and unifying data with reusable APIs, businesses can easily compose connected experiences while maintaining security and control. Through API-led connectivity, customers unlock business capabilities to build application networks that deliver exponentially increasing value. MuleSoft is the only unified platform for enterprise iPaaS and full lifecycle API management, and can be deployed to any cloud or on-premises with a single runtime.