Benchmarking Integration Pattern Implementations

Daniel Ritter, Norman May, Kai Sachs
SAP SE
Dietmar-Hopp-Allee 16
Walldorf, Germany
{first-name.last-name}@sap.com

Stefanie Rinderle-Ma
University of Vienna
Währingerstrasse 29
Vienna, Austria
stefanie.rinderle-ma@univie.ac.at

ABSTRACT

The integration of a growing number of distributed, heterogeneous applications is one of the main challenges of enterprise data management. Through the advent of cloud and mobile application integration, higher volumes of messages have to be processed, compared to common enterprise computing scenarios, while guaranteeing high throughput. However, no previous study has analyzed the impact on message throughput for Enterprise Integration Patterns (EIPs) (e.g., channel creation, routing and transformation).

Acknowledging this void, we propose EIPBench, a comprehensive micro-benchmark design for evaluating the message throughput of frequently implemented EIPs and message delivery semantics in productive cloud scenarios. For that, these scenarios are collected and described in a process-driven, TPC-C-like taxonomy, from which the most relevant patterns, message formats, and scale factors are derived as foundation for the benchmark. To prove its applicability, we describe an EIPBench reference implementation and discuss the results of its application to an open source integration system that implements the selected patterns.

1. INTRODUCTION

Integration systems have become ubiquitous in enterprise computing environments, since they address the need for (business) application integration by acting as a messaging hub [5]. The Enterprise Integration Patterns (EIPs) like message channel creation, routing, and transformation [12], as well as message delivery semantics (e.g., At-least Once, Exactly-Once) [26] constitute the building blocks of integration systems. Through a growing number of cloud applications, microservice architectures [8] and the rapidly growing amount of data from the Internet of Things (IoT) domain, integration systems gain even more importance.

These new cloud and mobile applications challenge classical integration systems because massive numbers of concurrent users, devices (i.e., message sources) and messages – with message sizes up to several hundred megabytes [30] – have to be processed. The EIP operations like complex routing patterns have to process the messages of diverse and complex data formats (i.e., nested, multi-format), and constitute a critical performance aspect of integration systems, which we showed in previous work on “data-aware” message processing [25]. The guarantee of reliable messaging, expressed through configurable message delivery semantics, adds a non-functional complexity to the message processing.

Given these new, challenging requirements, researchers and practitioners in related areas have defined more “data-aware” benchmarks that fostered novel solutions and allow for comparing them. For instance, in the area of data integration TPC-DI [19] was recently standardized. For analytical and (business) application processing, e.g., BigBench [6, 21] targets end-to-end analytics processing, however, underrepresents the integration aspect. Complementary, complex event- and stream processing benchmarks have been defined (e.g., [2, 16]). They focus on small portions of frequent data and analytical (stream) queries on actual and historic data. Recently developed IoT and cyber-physical system benchmarks [14] add new notions and can be seen as variants of the existing benchmarks. They specifically target the analytical processing of data within these applications (e.g., mostly event processing). On the application integration side, some efforts were made as part of the SOA benchmark [31], from which we take the ideas for the macroscale factors concurrent client and flexible payload size.

Despite the importance of application integration, many functional- and non-functional, performance-related questions cannot be answered today, due to the absence of a benchmark (cf. [33]). To close this gap, this work focuses on the following questions (Qx):

- What is the impact of complex routing conditions (Q1), multiple route-branchings (Q2) and message delivery semantics [26] (Q3) for “data-aware” scenarios?
- What is the impact of message sizes (Q4), e.g., large messages, and concurrent users (Q5), e.g., for a growing amount of IoT devices?
- What is the potential of new message processing approaches (e.g., “micro-batching” [25]) and how can they be compared (Q6)?

To answer these questions and to measure enhancements of current pattern implementations, we define a micro-benchmark for EIPs (without integration adapter processing) including functional and non-functional aspects with a strong focus on message throughput for “data-aware” integration scenarios.

DOI: 10.1145/1235

doi:2933267.2933269
Following some ideas from the SPEC SOA initiative [31], we base the design of the benchmark on flexible, configurable scale-factors on (i) a pattern or micro level and (ii) a general, macro benchmark level. The contributions of the paper are:

1. We analyze and classify common, “data-aware” integration scenarios to derive relevant patterns.
2. We define an EIP micro-benchmark that covers testable patterns, relevant for cloud applications, and specifies microscale factors for each pattern (incl. message delivery semantics) to answer questions Q1–Q3.
3. We specify macroscale factors that address the aspects of large messages and concurrent users to answer questions Q4–Q6.
4. We present a reference implementation of the benchmark to show its applicability and conduct experiments by example of the questions (Q1–Q6).

Our contributions are set into context to related work in Sect. 2. In Sect. 3 cloud integration scenarios of different integration styles from the SAP HANA Cloud Integration platform [30] are analyzed and classified. The general design choices of the benchmark are discussed in Sect. 4 (incl. message formats and macroscopic factors). The scenario analysis allows the derivation of relevant EIPs, for which we specify microscale factors in Sect. 5. Furthermore, we discuss the benchmark’s execution schedule and metrics in Sect. 6. In Sect. 7 we present benchmark experiments that answer the questions (Q1–Q6). Section 8 concludes our discussion and gives a brief outlook on future work.

2. RELATED WORK

In this section we survey related benchmarking approaches and analyze to what extent they satisfy core requirements for integration systems and thus help answering the questions Q1–6. Based on the comparison we derive gaps of current benchmarks, which let us define design criteria for EIPBench, see also Tab. 1.

For that, we categorize features of benchmarks in our field by their target system (e.g., Extract/Transform/Load (ETL), Messaging System (MS), Integration System (IS)) and scope (i.e., End-to-End (E2E) or Micro-benchmark (Micro); cf. [21]). We analyze the following benchmark dimensions along important IS tasks (cf. [5, 12]): (a) we conducted an evaluation about the message format definitions, leading to a differentiation between multi-format (MF), nested (NE) and simple messages. Then we checked (b) how well the related work supports EIP operations on these messages (e.g., content-based routing (CBR), message transformation (MT)) and (c) message delivery semantics in general. Hereby, format conversions on a message level (i.e., usually done by integration adapters [26]) are distinguished from those on the content level. The scale factors for (d) concurrent user measurements (Conc. Users) are specified as either configurable or static (i.e., cannot be changed), and (e) additional factors (SF) are shown separately. Since the EIPBench micro-benchmark, considers the operations in the integration process, integration adapter and transport protocol related topics are out of scope. These categories are discussed subsequently for each related field or target system and compared in Tab. 1 for their major representatives. To rate the maturity of a benchmark, the discussions contain hints on how recently the benchmarks were published and whether they are still actively maintained.

2.1 Integration System

The only known, public integration system benchmark is the ESB Performance benchmark [1], which was last executed in the year 2013. The benchmark defines E2E integration scenario performance measurements. The number of concurrent users is defined between 20 and 2,560 users, with a simple, flat XML-based payload embedded in a SOAP envelope. The test cases contain content-based routing on the SOAP header and the body with one simple string-equal routing condition using XPath, and XSLT-based format conversions (e.g., XML to CSV). Besides concurrent users, the benchmark defines a static scale level for message sizes (i.e., from 512 B to 100 KB). In contrast, EIPBench exclusively focuses on the performance (i.e., throughput) of EIP implementations, which requires more complex message formats and more elaborate EIP operation definitions that target only the message payload (currently defined by example in JSON format). In addition EIPBench defines tests for message sizes up to 500 MB and reliable messaging (message retry, idempotency repository, resequencing). As transport protocol, the ESB benchmark [1] uses HTTP only, while EIPBench measures the performance of EIPs in an integration process without protocol adapters (cf. Tab. 1).

2.2 Messaging System

The complementary field of Messaging System (MS) bench-
marks targets point-to-point message queuing and topic-based, publish-subscribe tests. The most prominent and still active representative is the SPECjms2007 benchmark [29], on which the.jms2009-PS [28] publish-subscribe benchmark is based. Although it addresses JMS implementations only, it defines an E2E benchmark for concurrent users (i.e., connections, sessions), scale-levels in the numbers of destinations and messages, and reliable, durable and persistent message queuing (cf. Tab. 1). The latter feature is similar to the reliable messaging in integration systems, which uses messaging systems for that purpose. However, SPECjms2007 does not define an EIP benchmark.

2.3 Data Integration / ETL

The work on data integration and ETL benchmarks can be considered conceptually related from a message transformation point of view. For instance, the recently released TPC-DI benchmark [19] defines an E2E data acquisition from multiple source systems with simple CSV, XML and TXT file data sets, and a data size scale factor for the import into multiple target systems (e.g., data warehouse). Similar to TPC-DI, the EIPBench uses the TPC-H data generator [23, 20], however, EIPBench constructs more complex message transformations (e.g., multi-format, nesting). The TPC-DI message transformations are format conversions as conducted by integration adapters [26] (e.g., XML or CSV to DB), which are different from the message transformations defined by the EIPs. The quality of service patterns in EIP are not in the focus of TPC-DI. On the other hand, the TPC-DI data quality checks are not in the EIPs, thus out of scope of the EIPBench (cf. Tab. 1).

The E2E Data-Intensive Integration Processes (DIPBench) [3] benchmark is positioned as hybrid, conceptual framework for ETL and integration system performance measurements. This discontinued benchmark targets the physical data integration within the context of ETL processes. Compared to EIPBench it does not specify EIP operations on the messages, works only with a simple XML-based message format, and neglects the message delivery semantics aspects of integration systems. Similar to our benchmark, DIPBench specifies several scale factors for data size, time and data distribution and allows to conduct concurrent user tests (i.e., parallel streams). The provided DIPBench tool suite [4] focuses on the flexible configuration and pluggability of integration adapters (cf. Tab. 1).

2.4 Stream / Event Processing

Benchmarks like FINCoS [17, 16] target the identification of performance bottlenecks in event processing systems, by measuring event throughput and the scalability of engines when increasing the throughput of small event messages and continuous streams. Similar to EIPBench different load conditions can be configured, however, messages sizes and format complexities are static. Although the defined operators (e.g., join, select, project) are similar to the operations in integration systems, the definition does not target the EIPs.

2.5 EIPBench

Summarizing our analysis in Tab. 1, none of the benchmarks covers all relevant aspects for the evaluation of integration processing in the context of “data-aware” scenarios. EIPBench fills this void and addresses the following aspects:

- the analysis and classification of common and new integration scenarios and the required patterns.
- representative message models used in these integration scenarios (cf. message format and conversion).
- the definition of a message throughput, micro-benchmark for EIPs (e.g., from [12]) that covers all requirements and specifies microscale factors for each pattern (incl. message delivery semantics; cf. Q1–3).
- the specification of macroscale factors that address the aspects of large messages (cf. Q1), concurrent user (cf. Q5) and micro-batching (cf. Q6).

3. INTEGRATION SCENARIOS

The EIPs from [12] are the building blocks for implementing integration systems. In this section, we set their usage in real-world cloud integration scenarios into context to generally known integration types and styles. The analysis is based on several cloud solutions, productively running on the SAP HANA Cloud Integration platform [30]. Therefore, more than 148 distinct integration scenarios with 934 common EIP usages out of 1429 were analyzed (w/o adapters).

![Overview of integration scenario types ST1–ST6; dashed lines mark aspect is out of scope.](image)

To derive the most relevant patterns of these scenarios, they are categorized according to their location in current enterprise architectures, their integration style and scenario type. Figure 1 shows the scenario types (ST) which are relevant for the message exchange between applications, users and devices to chain business processes of current enterprise integration architectures. Similar to [12], we define an integration style according to its purpose of message exchange (e.g., invoking business functions, synchronizing data), and we distinguish six scenario types, ST1–ST6; each of which denotes the type of endpoints that participate in the exchange (e.g., cloud application, device). The scenario types can follow different integration styles. An integration scenario can be seen as specific description of one type and style, composed of diverse integration patterns.

3.1 Integration Scenario Styles

According to [15] the classical Application-to-Application (A2A) integration styles are: Process Invocation (e.g., communicate creation or status updates of a business object) and Data Movement (i.e., synchronization and replication of a business object record). In particular, scenario type ST1 uses the integration style Data Movement which is typically realized using EIPs like Message Translator (MT). In Tab. 2
we summarize our analysis, and we also mention predominant message formats as well as example applications for each integration style. As scenarios types may use the same EIPs and message formats for different applications, we discuss the EIPs and message formats below.

We continue the analysis of integration scenario types with ST2, another application-to-application type. Unlike ST1, this integration scenario type focuses on the integration applications hosted in the cloud with on-premise applications. This type has become more prominent as applications are moving into the cloud, but they still need to be integrated with legacy on-premise applications. Furthermore, we identify ST3 which deals with the integration of different cloud applications. As indicated in Tab. 2 all three scenario types share the same integration styles: Process Invocation and Data Movement.

In addition, integration systems are often used in the area of User-centric Application Integration [10] (e.g., display customer financial status) for the consumption of business data by users. We call this integration style User-Centric Consumption, and it maps to scenario type ST4.

Furthermore, integrating physical devices with business applications becomes more important (e.g., medical [32] or connected car device integration). Since the term “Device Integration” is still not consistently defined, we apply the classical styles process invocation and data movement to the devices and call them Device Data Movement for ST5 and Device Invocation for ST6. Additionally, for scenario type ST6 we include a new scenario style, Data Processing, which is a combination of message processing and exchange as it is motivated by the related field of data analytics.

Again, Fig. 1 shows these six integration scenario types (ST1-ST6) considered in this work. Although technically covered by other scenario types, the cases of cross-partner (B2B), User-to-User and Machine-to-OP message exchange are out of scope of this work, and thus they are depicted by “dashed-lines”.

### 3.2 Analysis of Real-World Applications

For every scenario type discussed above, we now analyze real-world applications on how they realize integration scenarios using certain integration patterns and message formats, also see Tab. 2.

<table>
<thead>
<tr>
<th>Integration Style</th>
<th>Scenario Type</th>
<th>Msg. Format</th>
<th>CBR</th>
<th>MF</th>
<th>MC</th>
<th>SP</th>
<th>AGG</th>
<th>MT</th>
<th>CP</th>
<th>CE</th>
<th>CM</th>
<th>UDF</th>
<th>Example Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process Invocation, Data Movement</td>
<td>ST1: OP2OP</td>
<td>XML</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>ST2: OP2C</td>
<td>XML, JSON</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>ST3: C2C</td>
<td>JSON, XML</td>
<td>√</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>User-centric consumption</td>
<td>ST4: User2OP, User2C</td>
<td>XML, JSON</td>
<td>-</td>
<td>-</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>Device Data Movement</td>
<td>ST5: M2C</td>
<td>JSON, CSV</td>
<td>√</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>Device Data Movement</td>
<td>ST6: M2C</td>
<td>JSON, CSV</td>
<td>√</td>
<td>-</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+</td>
</tr>
</tbody>
</table>

In addition, as illustrated in Fig. 1, the integration scenarios in this work are grouped by their integration styles and example applications from SAP, Ariba and Success Factors (SFSF) applications. The most important integration patterns for each integration style are summarized in Tab. 2.

### 3.2.1 Analysis of Real-World Applications

**On-Premise-to-Cloud Integration (OP2C):** Through the trend of building cloud applications or moving existing applications to cloud environments, there is a growing need for communication with on-premise applications (ST2). For instance, SAP ERP / CRM on Demand and SAP S/4 HANA applications require status changes of on-premise applications as well as data replication, while existing on-premise applications tend to delegate integration with governmental organizations and institutions, e.g., for legal aspects, to cloud environments. In addition to XML, JSON gains importance for those scenarios that reach peak throughput of up to several 10,000 msg/s per sec. The message formats are mostly XML-based. In Tab. 2 we summarize the study of real-world scenarios from different integration styles in SAP HANA [30], setting them into context to the used integration patterns. Accordingly, the classical OP2OP scenarios mostly use Content-based Router (CBR) and Message Translator (MT) patterns.

**Cloud-to-Cloud Integration (C2C):** The fast-growing field of Cloud-to-Cloud (Cloud2Cloud) integration (ST7) includes micro-services [18], connects all kinds of business (e.g., Success Factors, Salesforce), social media (e.g., Twitter, Facebook), and business network applications (e.g., Ariba). Depending on the application domain, the message formats are mostly JSON-based, and the scenarios reach an even higher throughput, e.g., LinkedIn generates 100’s of GB of new data in the form of one billion messages per day. Facebook generates 6 TB of user activity data per data store. Especially in cloud scenarios there are several auxiliary patterns like encoders or decoders, signer, verifier,
decrypted or encrypt, which are mainly handled by integration adapters, e.g., WS-Security, thus out of scope for this work.

**User-to-On-Premise (User2OP) and Cloud (User2C) Integration:** The user-centric scenarios (ST4) are mostly about scheduled or ad-hoc, message-based queries that gather data from different data sources according to a user context and report back to the user. Thereby, the queries are latency- and message throughput bound (e.g., usually less than two seconds). To reach these requirements, a combination of MC and CE patterns are used to gather data in parallel and enrich the response message. The message transformation pattern is required in case of different source and target formats.

**Machine-to-On-Premise and Cloud (M2C) Integration:** Recently, the case of device invocation, data processing (ST6), and data movement (ST5) gained more importance. Scenarios like the convergent invoicing and vehicle integration:

Recently, the case of device invocation, data processing (ST6), and data movement (ST5) gained more importance. Scenarios like the convergent invoicing and vehicle integration:

- **4.1 Data Set and Message Creation**

  An important aspect from Tab. 1 is the message format. The analysis of scenario types in Tab. 2 indicates that mostly textual message formats are used (e.g., XML, JSON, CSV), while binary data (e.g., images, videos) is currently limited to few social media applications (cf. Cloud2Cloud). For the textual formats, there seems to be a move from XML to JSON, YAML\(^2\) (and CSV) formats. Hence, we define the message body format as textual, JSON and specify the integration pattern content accordingly.

  **4.1.1 Data Set**

  The messages can have an arbitrary format, however, current business application data and even social media data look similar to existing TPC data sets. Hence, we decided to start with a standard, PDGF-generated [20] TPC-H data set that provides different scale levels and – similar to BigBench [22, 24] – extended the generation for our purpose. The TPC-H data describes business object formats, which can be found within exchanged messages (i.e., less conversions). Although the generated data sets cannot be directly used as messages for the benchmark, they provide basic business objects such as ORDERS, CUSTOMER and do not require further explanation in the benchmark community. The TPC-H scale-level one generates 1.5 million ORDERS, 150k CUSTOMER, 25 NATION and 5 REGION records as CSV files.

  **4.1.2 Message Models**

  The message models are generated from the data sets using the following operations: join, union, append (\(\oplus\)), and scale. Following the edges, Figure 2 shows from left to right how the generated TPC-H source relations are combined to message formats. Subsequently a message \(MSG\) is defined as \(MSG := (B, H, A)\), with an arbitrary message content or body \(B\), an optional list of name-value pairs denoting the message header \(H\), describing the content, and a list of name-binary value pairs for the attachments \(A\) (optional). Usually the format of \(B\) is typed to one message model (e.g., TPC-H ORDERS). For our (source) message model \(MM\) we focus on the TPC-H order to customer processing. We selected the foreign key related relations ORDERS, CUSTOMER, NATION and REGION in the CSV message protocol, transform the single records to JSON and add two additional columns:

  **Figure 2:** Extended PDGF-based message creation.

---

\(^2\)YAML, visited 02/2016: http://yaml.org/
a unique message identifier and type information that specifies
the name of the source relation. The single JSON objects
are combined to one JSON array and stored as source model
for the benchmark execution. To sufficiently support “data-
aware” scenarios, the focus of EIPBench lies on the message
body and not the header. The source order messages are
defined as \( \text{MSG}_{\text{Ord}, B} := \{ \text{msgId}, \text{type} \} \oplus \text{OBJ}_{\text{Ord}, \text{fields}} \),
while \( \text{MSG}_{\text{Ord}, \text{H}[\text{A}]} := \emptyset \), where \( \text{OBJ}_{\text{Ord}, \text{fields}} \) are the
fields of the order object. Analogously, customer \( \text{MSG}_{\text{Cust}, \text{ation}} \),
nation \( \text{MSG}_{\text{Nat}} \), and region \( \text{MSG}_{\text{Reg}} \) are defined.

While the first, changes of messages results in a message
body \( B \) with a single message model, some scenarios require
Multi-format (MF) messages (e.g., convergent invoicing
requires additional information added to the message in
a different format). A MF message (\( \text{MSG}_{\text{MF}} \)) is defined
as a list of potentially different message models \( \text{MM} \). For
EIPBench \( \text{MSG}_{\text{MF}} \) messages with one \text{CUSTOMER} record
and all \text{NATION} and \text{REGION} records are created. Hence,
the source messages are defined as \( \text{MSG}_{\text{MF}} := \{ \text{msgId}, \text{type} \}
\oplus \text{OBJ}_{\text{Cust}, \text{fields}} \oplus \text{OBJ}_{\text{Nat}, \text{fields}} \oplus \text{OBJ}_{\text{Reg}, \text{fields}} \). Multi-
format messages transport additional, joinable information
as message content, while cyclic dependencies are allowed.

In addition, tree-like messages play a role, e.g., for OP2Cloud
scenarios. Hereby the foreign key relations between \text{CUSTOMER}
and \text{NATION} relation are replaced within the customer
record beforehand, leading to named message structures \( N \).
For instance, SAP Intermediate Document (IDoc) Types allow
the definition of segments, which are a parent-child-like
structure\(^3\). The nested source messages \( N \) of customer and
nation objects are defined as \( \text{MSG}_{\text{NC}, \text{n}} \) := \{ \text{msgId}, \text{type} \}
\oplus \text{OBJ}_{\text{Cust}, \text{fields}} \oplus \text{OBJ}_{\text{Nat}, \text{fields}} \\oplus \text{OBJ}_{\text{Reg}, \text{fields}} \). Multi-
format messages transport additional, joinable information
as message content, while cyclic dependencies are allowed.

To support a message scaling over orders with a list of
nested customer records \( \text{MSG}_{\text{NC}, \text{n}} \), we define \( \text{MSG}_{\text{NC}} := \text{MSG}_{\text{NC}, \text{m}} \) with \( m > 1 \). Messages of several hundred
MB, e.g., as required for Financial Service Network Cloud-
lapping, are not only “fast”, but “big” data.

Each single message model can be stashed into a mes-
sage collection \( \text{Col}_{\lambda} (\text{MSG}) \), where \( \text{MSG} := \{ \text{MSG}_{\text{Ord}}, \text{MSG}_{\text{MF}}, \text{MSG}_{\text{N}}, \text{MSG}_{\text{Scale}} \} \) and collection size \( \lambda \),
which specifies the number of messages within the collection.

4.2 Macroscale Factors
The integration patterns need to scale along different di-

censions. Consequently, we define the following macroscale
factors: (i) messages with different user contexts (i.e., con-
current users), (ii) micro-batching, and (iii) message size
(implicit and explicit).

The scale factor \text{concurrent users} (i) tests the ability
of pattern implementations to handle concurrent requests. The
generic, concurrent user load pattern for a particular scale
level can be freely configured and is defined as:

\[
scale_{\text{cu}}(\omega) = 2^\omega
\]  

(1)

For example, when transferring the settings of the ESB Per-
formance benchmark [1], \( \omega \) varies between 0 and 11. In our

\(^3\)SAP IDoc structure, visited 02/2016: http://
help.sap.de/saphelp_46e/helpdata/en/de/6b824843d711d1893e000e8328c4f/content.htm

experiments, we use \( 0 \leq \omega \leq 6 \), which already sufficiently
shows the impact of this scale factor to answer question \( Q5 \).

Furthermore, we define the \text{micro-batching} (ii). With this
parameter we intend to show the benefit of batched pro-
cessing for the message throughput in integrations systems (cf.
\( Q6 \)). In this context, the “data-aware” processing approach
is a newly developed mechanism that allows to send collect-
ions of messages \( \text{Col}_{\lambda} (\text{MSG}) \) instead of single messages
[25], called \text{micro-batching}. Currently only the patterns dis-
cussed in [25] are micro-batch enabled, e.g., message trans-
formation. The batch scale levels \( \beta \), with \( 0 \leq \beta \), denote the
number of distinct messages in one message collection
\( \text{Col}_{\lambda} (\text{MSG}) \) as defined:

\[
\lambda := scale_{\text{batch}}(\beta) = 2^\beta
\]  

(2)

The ESB Performance benchmark [1] does not specify such a
test. In EIPBench \( \beta \) is configurable, and we choose \( \beta \) as
\( 0 \leq \beta \leq 10 \) to show the general impact of micro-batching.

Especially in cloud-to-cloud integration scenarios and also
business network solutions we observe that message sizes
(iii) of various sizes are used. EIPBench addresses this chal-
lege by constructing larger message sizes (cf. \( Q5 \)) of multi-
format \( \text{MSG}_{\text{MF}} \) and nested \( \text{MSG}_{\text{NC}} \) messages, as they are
used by various applications. For a given message of type
\( \theta \) we define a function \( \text{size}(\{ \text{msgId}, \text{obj} \} ) \), which determines
the message size in kB. For instance, the size of \( \text{MSG}_{\text{Ord}} \) is
approximately 0.354 kB and for the nested customer object
\( \text{size} (\text{OBJ}_{\text{Cust}}) \approx 0.293 \) kB. For the size of nested messages,
the type of the nested business object \( \text{obj} \) can be specified.

The exteded function \( \text{size}(\text{msg}, \theta, \tau) \) calculates the size in-
clusive the nested object. Since the nesting is calculated, by
a foreign key \( \text{fk} \) relation between the business objects, a
function \( \text{size}(\{ \text{msgId}, \text{obj}, \text{fk} \} ) \) returns the size of a message or
object without the foreign key field. The generic size
calulation of objects and messages is defined as:

\[
\text{size}(\text{MSG}_0) = \text{size}(\text{MSG}_0, \text{fk})
\]
\[
= \text{size}(\text{MSG}_0) - \text{size}(\text{fk})
\]
\[
\text{size}(\text{OBJ}_0) = \text{size}(\text{OBJ}_0, \text{fk})
\]
\[
= \text{size}(\text{OBJ}_0) - \text{size}(\text{fk})
\]  

(3)

Now, the size of these messages is scaled through parameter \( \eta \),
with \( 1 \leq \eta \leq 20 \). For example, for \( \eta = 20 \) we generate
messages of approximately 512 MB in size. In comparison,
the ESB Performance benchmark [1] specifies messages up
to 100 kB. In addition to the generic scale factor \( \eta \), there is
another message size factor \( \gamma \), which helps to increase the
number of business objects of a scale level: Equation (4)
brings all previous pieces together and shows the generic
calulation of the message size of a scaled message \( \text{MSG}_{\text{Scale}} \)
for a particular scale level \( \eta \).

\[
\text{MSG}_{\text{Scale}} = \text{size}(\text{MSG}_0) + \eta \cdot \gamma \cdot \text{size}(\text{OBJ}_0)
\]  

(4)

For instance, the concrete scale factor constant in EIPBench
is \( \gamma = 6 \). For the nested messages \( \text{MSG}_{\text{NC}} \), \( n \) is defined
as \( n := \gamma \cdot \eta \). Concrete values, e.g., for simple custom-
object messages in EIPBench with \( \theta, \tau := \text{Cust} \), range between
approximately 256 B for \( \eta = 0 \) up to 256 MB and 512 MB.

4.3 Summary
Based on the integration scenario analysis and classifica-

tion, we identified appropriate message formats and macroscale
factors for EIPBench. Considering the focus on “data-aware”
5. PATTERN DESIGN CHOICES

In this section, we define microscale factors for the patterns to be tested based on the categories of message routing, transformation patterns, and message delivery semantics from the integration scenario analysis. Each pattern represents an operation on one or multiple of the defined message models and defines its own microscale factors. The microscale factors describe and test the complete characteristics of the patterns. Subsequently, the scale levels and variations for the different benchmarks are enumerated alphabetically, while A usually denotes the normal or simple case and the cases (B, C, . . . ) represent (scale) variants.

5.1 Message Routing Patterns

The message routing patterns decouple the message sender from its receiver(s). We focus on content-based routing capabilities (i.e., no header), which are mainly used in practice and especially relevant for the evaluation of “data-aware” processing. Table 3 lists the relevant routing patterns (RT) from [12], which are subsequently discussed.

<table>
<thead>
<tr>
<th>Label</th>
<th>Patterns</th>
<th>Description</th>
<th>Scale</th>
</tr>
</thead>
<tbody>
<tr>
<td>RT-1</td>
<td>CBR, MF</td>
<td>Channel cardinality 1:[1</td>
<td>n], n ∈ N outgoing channels, m ∈ N (dis-) conjunctive conditions w/ increasing complexity same as RT-1 on multi-format message with k ∈ N optics</td>
</tr>
<tr>
<td>RT-2</td>
<td>CBR, MF</td>
<td>Multicast (MC)</td>
<td>D: k &gt; 1</td>
</tr>
<tr>
<td>RT-3</td>
<td>Recipient List (RL)</td>
<td>Message cardinality 1:n, n ∈ N outgoing channels, parallel processing, stop on exception same as RT-3 with n receiver determinations</td>
<td>A: n = 1, B: n &gt; 1, and variations</td>
</tr>
<tr>
<td>RT-4</td>
<td>Splitter (SP)</td>
<td>Message cardinality 1:i, i ∈ N outgoing messages, parallel processing, stop on exception message cardinality 1:i, i ∈ N incoming messages</td>
<td>A: n = 1, B: n &gt; 1, and variations</td>
</tr>
<tr>
<td>RT-5</td>
<td>Aggregator (AGG)</td>
<td>Content-based Filter (CF)</td>
<td>i &gt; 1, split cardinality, variations</td>
</tr>
</tbody>
</table>

The multicast (MC) describes the statically configured serial or parallel sending of n copies of the same message to n receivers, while the recipient list (RL) dynamically computes the receivers from the original message through a receiver determination function. Technically, both patterns create message channels (i.e., threads) for each outgoing message.

Example: Copy one order message to several (parallel) channels statically for further processing with a MC, or select or calculate the message channel from the body of the message with the RL (e.g., orders with different priorities).

Scale/Variations: through branches (tests threading, branching), parallel branching vs. sequential processing; on exception.

Implementation: EIPBench scales multiple outbound message channels for the MC and configures RL to use one outbound route per order priority on MSGOrd, which tests the channel branching behavior (i.e., channel creation).

5.2 Message Transformation Patterns

The message transformation covers an important aspect of integration systems that contain the translation of one format into another one (Message Translator (MT) [12]), the enrichment of additional information to a message (Content Enricher (CE) [12]) and the filtering of content (Content Filter (CF) [12]). In more practical realizations, the Message Mapper [12] is used to convert from the message’s format to a Canonical Data Model [12]. In addition, new patterns can be found for executing arbitrary scripts on the message (Script pattern) [30, 13] and for the more guided modification of the content using expression editors in form of the Content Modifier (CM) pattern [30].

In this work, we focus on the standard MT, (transient, internal) CE and CF patterns (MT-1–3) as shown in Tab. 4.
Table 4: Message Transformation (MT) patterns with microscale factors.

<table>
<thead>
<tr>
<th>Label</th>
<th>Patterns</th>
<th>Description</th>
<th>Scale</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-1</td>
<td>MT</td>
<td>program with ( n ) distinct field mappings, each with a directed operator tree of size ( i ) ( n, m \leq 10 ) and ( i = 1 )</td>
<td>(A) ( n, m \leq 10 ) and ( i = 1 )</td>
</tr>
<tr>
<td>MT-2</td>
<td>CF</td>
<td>enrich message with ( j ) new fields or complete structures ( j &gt; 0 ); nesting and multi-format variables ( o \geq 1 )</td>
<td></td>
</tr>
<tr>
<td>MT-3</td>
<td>CE</td>
<td>enrich message with ( j ) new fields or complete structures ( j &gt; 0 ); nesting and multi-format variables ( o \geq 1 )</td>
<td></td>
</tr>
</tbody>
</table>

For all of these patterns the channel- and message cardinalities are 1:1, i.e., they are non-message generating, and we only consider the stateless cases here due to lack of space.

**Message Translator (MT-1).**

Message translators (MT) transform the structure and values of an incoming message. The mapping program has \( n \):\( m \) distinct field mappings, where the incoming message has \( n \) and the outgoing message \( m \) fields. Each field mapping can be expressed with a directed operator tree of size \( i \). For instance, \( i = 1 \) means one operation is used to transform one field into another one. The single operations intersect with some of the information integration queries defined in [11] (e.g., Query-2 “Mathematical operations”, Query-3 “String contains”). According to the study of [30], MT for integration programs is more complex and can be summarized to arbitrary combinations of the following \( n \)\( \)-ary operations:

- value assignments/mappings: e.g., default values, constants, copy.
- type-specific operations: e.g., String concat, numeric subtract, addition.
- conditions: e.g., equals, greater than, contains.
- external scripts/functions: e.g., value mapping lookups, user-defined functions, external service calls.

**Example:** Transform the mandatory ORDERKEY field to one or many fields of the target structure. For instance, the distinct mapping from source field \( A \) to target field \( B \) checks that the ORDERKEY is not null, has a certain length and only then assigns its value: checkNotNull(A) \( \rightarrow \) checkLength(A) \( \rightarrow \) assign(A, B).

**Scale/Variations:** variations of increasing numbers of \( n \), \( m \) and the size of \( i \), as well as the complexity of the operations (e.g., iterative calculations).

**Implementation:** Since the operator-tree processing of the MT is similar to complex conditions that are already checked in the benchmark, only a simple mapping program (A) is benchmarked in EIPBench.

**Content Filter (MT-2).**

Content filters (CF) remove \( o \) fields and values from a message.

**Example:** the message receiver only requires ORDERKEY, CUSTKEY and ORDERPRICE, and all other fields and values are removed.

**Scale/Variations:** increase number of filtered fields \( o \); more complex filter conditions.

**Implementation:** EIPBench filters fields of ORDER messages.

Table 5: Message Delivery Semantics (MDS) with microscale factors.

<table>
<thead>
<tr>
<th>Label</th>
<th>Patterns</th>
<th>Description</th>
<th>Scale</th>
</tr>
</thead>
<tbody>
<tr>
<td>MDS-1</td>
<td>MRoE</td>
<td>redeliver message on failure ( o \in N ) times, (non-) original message ( A: o = 1 ), ( B: 32 \geq o &gt; 1 ); ( F: ) variants ( n \geq 1 ) ( A: n = ) 10</td>
<td></td>
</tr>
<tr>
<td>MDS-2</td>
<td>RS</td>
<td>sequence of ( n \in N ) messages, sequence identifier ( m \in N ) “in-memory” ( m \in ) “in-memory” ( m \in ) “in-memory” ( A: m = 0 ), ( B: 100,000 \geq m &gt; 0 )</td>
<td></td>
</tr>
</tbody>
</table>

**Content Enricher (MT-3).**

Content enrichers (CE) add \( j \) new fields and values to a simple message, or build a nested or multi-format message structure.

**Example:** the message requires additional master data for the credit check of a customer, which is added to the current message.

**Scale/Variations:** increase number of added fields \( j \); build more complex structures.

**Implementation:** EIPBench focuses on the “in-memory” enrichment of message content (i.e., leaves out external calls).

5.3 Message Delivery Semantics

For reliable messaging, integration scenarios require different levels of message delivery semantics: best effort (BE), at least once (ALO), exactly once (EO), and exactly once in order (EO/O) [26], which can be composed through the standard idempotency repository (IR) and resequencer (RS) patterns from [12], and the message redelivery on exception (MRoE) pattern from [27].

The previously discussed benchmarks assume no reliability, which either means message redelivery on exception by the applications or devices in case of synchronous messaging, or message-loss after an unforeseen event during asynchronous BE delivery. If the message shall be delivered ALO, a MRoE pattern is required, which might lead to duplicate messages exchange. To avoid that EO combines ALO with an IR pattern, which filters out duplicate messages. When a special sequence of messages shall be preserved (e.g., create before update operation), then EO is combined with a RS pattern. The microscale value domains are configurable. However, for the our experiments with EIPBench they are set to values, which show the general impact on message processing. Table 5 lists the relevant message delivery semantics, which are subsequently discussed.

**Message Redelivery on Exception (MDS-1).**

Redeliver messages on exception (transient) to receiver \( o \) times.

**Example:** The creation of an order fails due to a temporary network outage and will be immediately re-delivered to make sure that the order will reach its destination as soon as the issue is solved.

**Scale/Variations:** increase number of redeliveries \( o \); send original or modified message

**Implementation:** EIPBench configures the MRoE pattern with \( A \) no redelivery, \( B-F \) with \( o := \{ 1, 2, 4, 8, 16, 32 \} \) on MSGOrd messages.

**Resequencer (MDS-2).**

Receive sequence of messages (transitively), correlate us-
ing a sequence identifier [12] and re-order, when the sequence is complete.

**Example:** The creation of a customer has to happen before the update of the same customer (i.e., sequence of operations) or before the creation of a referenced order (i.e., sequence of operations) or before the creation of a referenced order (i.e., sequence of operations) or before the creation of a referenced order (i.e., sequence of operations) or before the creation of a referenced order (i.e., sequence of operations).

**Scale/Variations:** number of entries per sequence

**Implementation:** EIPBench varies the number of sequence entries, with \( n := \{10, 100, 1000, 10000, 100000\} \); implemented for (A) \( n = 10 \) and resequencing messages according to their **TOTALPRICE** on MGSOrd messages.

**Idempotent Receiver (MDS-3).**

Filter duplicate messages using (transient) memory.

**Example:** The message source sends the same order twice for creation in another application (same ORDERKEY).

**Scale/Variations:** increasing \# duplicates leads to more main memory consumption due to transient and more frequent lookups or scans

**Implementation:** EIPBench configures the IR for (A) no duplicates, (B) duplicates after 100,000 checked for msgId on MGSOrd messages.

6. BENCHMARK IMPLEMENTATION

The EIPBench is executed close to the pattern implementations, potentially even within the same process. Our reference implementation uses JMH⁴, a Java harness for running benchmarks on the JVM, which factors out JVM side-effects (e.g., on stack replacement) through code generation and allows to configure warmups, iterations and the number of isolated JVM instances. Based on JMH, a tool suite is provided that contains:

**Initializer:** for generating the data and creating the messages in the preparation (pre) phase.

**Client:** that selects the benchmarks in the preparation (pre) phase and uses JMH to schedule the execution of message processors for the different integration scenarios in the work phase.

**Monitor:** collects the statistics, calculates performance metrics and plots the results in the post-processing (post) phase (not shown).

As illustrated in Fig. 3, the benchmark realization is divided into three main phases: initialization (pre), execution (work), and verification (post). The time of the pre phase \( T_{pre} \) consists of the creation of a fork \( T_{fork} \), the loading of all messages required by the current benchmark \( T_{load} \), and the preparation of the start of the benchmark \( T_{start} \) (cf. Eq. (5)).

\[
T_{pre} = t_1 - t_0
= T_{fork} + T_{start} + T_{load}
\]

During the work phase, the client executes the defined pattern benchmarks on a specified number of isolated and freshly initialized JVM instances, called forks \( \zeta \), for a configurable amount of warmup and main iterations. The execution time of this phase \( T_{work} \) mainly adds up the warmup \( T_{warmup} \) and the actual evaluation time \( T_{eval} \) (cf. Eq. (6)).

\[
T_{work} = t_2 - t_1
= T_{warmup}(\Phi) + T_{eval}(\Phi)
\]

\[
= m \cdot \text{eval}(\phi) + n \cdot \text{eval}(\phi)
\]

During the evaluation, the selected benchmark is executed, and the discrete throughput values \( \phi \) are collected. Each fork accesses the created message files \( (T_{load}) \) and sends (collections of) messages to the message channel with the tested patterns. Hence the overall runtime of the whole benchmark is \( T_{Bench} = \zeta \cdot (T_{pre} + T_{work} + T_{post}) \). To measure \( T_{work} \), the message scenarios are synchronous and have a void receiver adapter, which immediately returns to the sender. Then, cleanup and verification are performed (cf. Eq. (7)).

\[
T_{post} = t_3 - t_2
= T_{clean} + T_{verify}
\]

When a complete scale factor run is finished, the results are serialized to disk in a raw format, containing all captured measurements. The monitor parses the data and creates plots for all tested patterns and scale factors.

The relevant metrics for EIPBench is the discrete throughput measures \( \phi \) of a tested pattern (i.e., \( T_{eval} \)). More precisely, \( T_{eval} \) is the calculated mean of the individual evaluations \( \text{eval}(\phi_i) \), with \( i \in I \), for the number of iterations \( I \) within one fork (cf. Eq. (8)).

\[
T_{eval} = \frac{T_{mean/fork}}{n}
\]

For reproducible results the whole test instance will be cleared after one fork and initialized. The benchmark will be executed for the number of forks \( \zeta \). Equation (9) shows the calculation of the mean for multiple forks. While higher number of forks (i.e., \( \zeta = >> 10 \)) leads to increasing overall execution times, the results become more reproducible.

\[
T_{mean} = \frac{\sum_{j=1}^{\zeta} T_{mean/fork}(j)}{\zeta}
\]

\[
T_\alpha = \sqrt{\frac{\sum_{i=1}^{\zeta}(T_{eval} - T_{mean})^2}{\zeta}}
\]

In addition to the mean, EIPBench measures a confidence value for the result with a confidence level \( \alpha \) of 99% (i.e., confidence interval \( \alpha \)). The confidence interval is calculated once for all forks based on the observed mean throughput values and the standard deviation. Equation (10) shows the upper and lower bound calculation of \( T_\alpha \).

\[
T_\alpha = \begin{cases} T_{mean} - \alpha \cdot \frac{T_\alpha}{\sqrt{\zeta}}, & \text{lower} \\ T_{mean} + \alpha \cdot \frac{T_\alpha}{\sqrt{\zeta}}, & \text{upper} \end{cases}
\]

---

7. EXPERIMENTS

In this section we briefly describe the setup of the benchmark and share results running the benchmark to answer our guiding questions and discuss lessons learned, e.g., including “deficits” found in the pattern implementations.

7.1 Benchmark Setup

All measurements are conducted on a HP Z600 work station, equipped with two Intel X5650 processors clocked at 2.67GHz with a 12 cores, 24GB of main memory, running a 64-bit Windows 7 SP1 and a JDK version 1.7.0 with 2GB heap space.

For our experiments we used the test harness described in Section 6. As first system under test, we decided to use the open-source integration system Apache Camel [13] implemented in Java, referred to as Java/AC, since it provides implementations for all discussed patterns and is used in SAP HCI [30]. For comparison we have chosen a Java-based, “data-aware” integration pattern implementation [25], which simulates table operations on the message content, unmarshalled to OIC-iterators instead of JSON objects during TLoad. Since the data-aware implementations use Datalog and are embedded into Apache Camel, we subsequently use the term TIP/AC synonymous to Datalog.

7.2 Benchmark Results

For the discussion of the benchmark results, we follow the research questions Q1–Q6, for which we show representative results, instead of discussing each particular result. Subsequently all diagrams show message throughput for different scale levels. Discrete points are calculated mean values $T_{mean}$ (cf. Eq. (9)) according to the metrics, and the error bars denote the precision of the values according to the 99.9% confidence interval $T_{ci}$ (cf. Eq. (10); i.e., small intervals indicate low variance, thus a higher confidence).

Before benchmarking the different patterns, we conducted a “baseline” benchmark using Java/AC without any pattern configurations, which measures the pipeline processing without operations on the message (cf. BL in Tab. 6).

7.2.1 Microscaling

To answer the “microscale” questions Q1 and Q2 about the impact of complex routing conditions and multiple branchings, we benchmarked the routing test description RT-1 (i.e., context-based routing) together with streams of MESSAGEs for the Java/AC and TIP/AC implementations. Conceptually the routing conditions are similar to the examples for the patterns in Sect. 5.

On the impact of complex routing conditions (Q1) and multiple route branchings (Q2): Table 6 shows the results of RT-1 starting with the simple routing condition case RT-1 (A), followed by increased route branchings RT-1 (B), condition complexity RT-1 (C), and complex conditions on multi-format messages. Not surprisingly, the materialization of messages for processing by a pattern implementation results in a significant decrease in the throughput compared to the baseline measurement (cf. BL). The number of route branchings in RT-1 (B) correlates with the number of evaluated conditions (worst case). In our experiments, all conditions are executed. The impact of an increasing branch-

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Scale</th>
<th>Java/AC (early-out)</th>
<th>Java/AC</th>
<th>TIP/AC</th>
</tr>
</thead>
<tbody>
<tr>
<td>BL</td>
<td>n/a</td>
<td>300.837 +/-.070</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>RT-1</td>
<td>A (simple)</td>
<td>174.795 +/-.115</td>
<td>176.319 +/-.926</td>
<td>179.528 +/-.545</td>
</tr>
<tr>
<td></td>
<td>B (branching)</td>
<td>158.838 +/-.132</td>
<td>100.070 +/-.859</td>
<td>163.672 +/-.417</td>
</tr>
<tr>
<td></td>
<td>C (complex)</td>
<td>3.002</td>
<td>2.635</td>
<td>4.186</td>
</tr>
<tr>
<td></td>
<td>D (join)</td>
<td>115.599 +/-.261</td>
<td>98.237</td>
<td>115.859 +/-.3.417</td>
</tr>
<tr>
<td></td>
<td></td>
<td>165.644 +/-.312</td>
<td>2.261</td>
<td>176.926 +/-.6.514</td>
</tr>
<tr>
<td>MT-1</td>
<td>A (medium)</td>
<td>193.545 +/-.672</td>
<td>7.612</td>
<td>4.407</td>
</tr>
</tbody>
</table>

Throughput denoted by $T_{mean}$ (cf. Eq. (9)) and $T_{ci}$ (cf. Eq. (10)).

Further message routing impact factors: During the implementation of the benchmark, the “early-out” capability of implementations (i.e., filter can return halfway during the scanning (for row filter) [9]) turned to another important factor of routing throughput. The Java/AC “early-out” implementations are comparable to the corresponding TIP/AC implementations. However, the non-“early-out” Java/AC implementation performs even worse apart from RT-1 (A), which is conceptually equal to the “early-out” variant.

The microscale factor (D) for cross-relation operations requires a multi-format message $MSG_{CNR}$. Therefore a cross-relation operation is used for TIP/AC, which is represented by a join over the CUSTOMER and NATION relations with several conditions. For the TIP/AC implementation these operations seem more natural than for the AC/Java implementations, thus show slightly better results.

On the impact of complex message transformations: The results for the benchmark of MT-1 message transformation of simple (A) mapping programs are shown in Tab. 6. In this case the TIP/AC implementation outperforms the Java/AC approach, which is designed for “data-aware” operations on messages. Again, message transformation operations seem more natural for a “data-aware” implementation. Hence, further investigations on an extension or refinement of the EIP semantics for data-aware processing could be preferable.

On the impact of message delivery semantics (Q3): The study of the impact of the message delivery semantics (cf. Q3) touches the inner workings of the integration pipeline system, thus are only executed for Java/AC. Table 7 shows the microscaling of MESSAGEs for an increasing number of retries starting with $1 \leq o \leq 32$. The variant “use-original message” (not shown) does not show a significantly different throughput behavior. Since MESSAGEs MRoE is a “loop” pattern, this test allows insight in the loop-
processing capabilities of the runtime system. The redelivery delay penalty (without exponential backoff) becomes notable in the results for an increasing amount of redeliveries. This raises questions for future work like “Could a more scalable implementation keep up the general message throughput of the system and deliver messages in redelivery separately?”.

For the resequencer pattern, Tab. 7 shows case MDS-2 (A), which measures the throughput of a resequencer with a sequence size of $n = 10$. That means, after the reception of 10 unordered messages, the messages are ordered and resumed. The relatively low impact on the throughput is a result of not persisting the sequences in an operational datastore.

Conceptually, the (transient) idempotent receiver and the message filter patterns are comparable. This is supported by the similar message throughput as shown in Tab. 7 MDS-3 (A) with a duplication factor of $m = 100,000$ messages.

7.2.2 Macroscaling

To answer the “macroscale” questions Q4–Q6 about the impact of message sizes, concurrent users and micro-batching, we benchmarked the routing test description RT-1 (i.e., content-based routing) together with streams of $MSG_{ord}$ messages for the Java/AC and TIP/AC implementations. Conceptually the routing conditions are similar to the examples for the patterns in Sect. 5.

On the impact of increasing message sizes (Q4): The “data-aware” messaging question Q4 about increasing message sizes for content-based routing leverages RT-1 together with messages of type $MSG_{scale}$. Figure 4 shows the immense impact of big messages for RT-1 (A) and RT-1 (B). Notably, the data-aware implementation performs slightly better for messages bigger than 64 MB. Especially for the TIP/AC approach, handling bigger amounts of “data-aware” data similar to “in-memory” database table processing should be further studied.

On the impact of concurrent users (Q5): Especially for Machine2Cloud (cf. ST6) integration scenarios, “concurrent user” cases are common, which we formulated in question Q5. Figure 5 shows the “multi-threading” scaling capabilities of AC for the routing cases RT-1 (A) and RT-1 (B) showing an early saturation after $scale_{\omega}$ with $\omega = 3$.

The results indicate a non-optimal usage of hardware resources through the Camel threading model [13], used by the EIP implementations. For instance, a thread pool can be configured for the Multicast [13], but not for the router pattern. However, even with a sufficiently configured threading, the multicast implementation does not reach a message throughput comparable to the router (cf. Sect. 7.2.3). This observation and further measurements indicate an impact on composed patterns like scatter-gather implementation [12] (i.e., multicast and aggregator).

On the impact of micro-batching (Q6): For integration scenarios that trade the single message processing latency for message throughput and the overall latency (e.g., especially data movement and data processing ST5, ST6 as well as process invocation scenarios ST1–3), the processing of collection of messages Q6, called “micro-batching”, seems to be beneficial. Figure 6 shows a good scaling behavior of the “data-aware” TIP/AC implementation, which is able to process several messages in ONC-format with one operation. The scalability outperforms event “multi-threading” by factors. To fully leverage “micro-batching” within integration systems, the EIP semantics [12] have to be re-visited in future work.

7.2.3 General Aspects and Deficits

The benchmark results show general integration system aspects, which are important for the message throughput. Besides the routing and transformation, the system is re-
responsible for the message and channel creation [12]. For instance, the creation of messages is part of the RT-5 and RT-6 benchmarks, while channel creation is covered by RT-3 and RT-4 (not shown).

The results indicate that the message creation involves time consuming operations (e.g., message ID generation, message model creation, format transformations), thus lower the throughput of those patterns. The creation of channels requires thread management (e.g., thread creation, pooling), which has an even bigger effect on the message throughput, thus making patterns like the “machine-local” load balancer [13], practically unusable in “data-aware” scenarios.

8. SUMMARY AND OUTLOOK

With EIPBench we specify the first benchmark for integration patterns, which play a crucial role for the message throughput of integration systems. The benchmark definitions put emphasis on the identified micro- and macroscale factors, for which we provided a reference implementation. Based on that, we experimentally evaluated the benchmark definitions along the discussed research questions (Q1–Q6). Besides the benchmark results, the analysis brought up several areas for future research in the area of the benchmark (e.g., extend the benchmark for pattern composition and integration adapter processing) and more efficient message processing (e.g., routing selectivity and re-ordering, more efficient “in-memory” TIP/AC processing). To fully leverage “micro-batching” within integration systems, the EIP definitions [12] might be extended. In this context, the system aspects message and channel creation have to be re-visited.

9. REFERENCES