Accepted version. The published version is available here: http://dl.acm.org/citation.cfm?doid=2933267.2933269

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, under-represents 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.

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

Table 1: EIPBench in the context of related work.

Benchmarks	Category	Transport protocol	Message format / conv.	EIPs (cf. *Q1*,*Q2*)	Message delivery semantics (cf. *Q3*)	Scale factors (cf. *Q4*,*Q6*)	Concurrent users (cf. *Q5*)
ESB Perf. [1]	IS, (EIP) [E2E]	HTTP	simple (SOAP-XML)	CBR, MT (partially)	–	msg sizes, concurrent users	static
SPECjms2007 [29], jms2009-PS [28]	MS, JMS [E2E]	JMS, AMQP	n/a	–	reliable, transactional	#dest, #msgs	user sessions
TPC-DI [19]	DI, ETL [E2E]	FILE, DB	simple (CSV, XML, TXT)	–	–	data (incr. load)	multiple sources
DIPBench [3, 4]	DI, ETL, IS [E2E]	configurable	simple (XML)	–	–	data size, time, distribution	parallel streams
FINCoS [16, 17]	CEP, Stream [E2E]	FILE, JMS	simple (CSV)	–	–	#msgs	–
EIPBench	IS, **EIP [Micro]**	n/a	complex **MF, NE (JSON)**	covered	**reliable messaging**	**concurrent users, micro-batching, message sizes**	configurable scale factor

Category: Integration System (IS), Messaging System (MS), Java Message Service (JMS), Extract/Transform/Load (ETL), Data Integration (DI), Complex Event Processing (CEP); Enterprise Integration Patterns (EIPs): Content-based Routing (CBR), Message Transform. (MT); Format: Multi-Format (MF), NEsted (NE).

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 macroscale 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 protocol level (i. e., usually done by integration adapters [26]) are distinguished from those on the message 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) benchmarks 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 formats (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. *Q4*), 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).

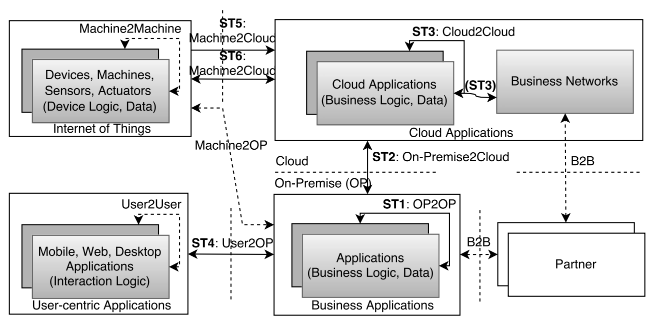


Figure 1: Overview of integration scenario types *ST1–ST6*; dashed lines mark aspect is out of scope.

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

Table 2: Integration scenarios grouped by their integration styles and examples from SAP, Ariba and Success Factors (SFSF) applications. Usual pattern occurrences are marked by √.

Integration Style	Scenario Type	Msg. Format	Patterns										Example Applications
			CBR	MF	MC	SP	AGG	MT	CF	CE	CM	UDF	
Process Invocation, Data Movement	ST1: OP2OP	XML	√	-	-	-	-	√	-	-	-	-	SAP ERP / CRM
	ST2: OP2C	XML, JSON	√	√	-	√	-	√	-	-	√	-	SAP C4C, SAP S/4 HANA
	ST3: C2C	JSON, XML, (binary)	√	√	-	√	-	√	-	√	√	√	SFSF Employee Central, Ariba Quadrum Network, SAP S/4 HANA
User-centric consumption	ST4: User2OP, User2C	XML, JSON	-	-	√	√	√	√	-	√	-	-	Hybris Social Marketing
Device Data Movement	ST5: M2C	JSON, CSV	√	-	-	-	-	√	-	√	√	-	Vehicle Logistics, Connected Cars
Device Invocation, Data Processing	ST6: M2C	JSON, CSV	√	-	-	√	√	√	√	√	√	√	Sports Management, SAP Convergent Invoicing

Content-based Router (CBR) and Message Filter (MF) (↦ 11.35% of 934), Multicast (MC ↦ 0.75%), Splitter (SP ↦ 8.14%), Aggregator (AGG ↦ 0.32%), Message Translator (MT ↦ 32.86%), Content Filter (CF ↦ 2.99%), Content Enricher (CE) and Content Modifier (CM) (↦ 42.93%), user defined functions (UDF ↦ 0.64%).

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*. Additonally, 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.

On-Premise Integration (OP2OP): The application-to-application message exchange between business applications (*ST1*) within one corporate network, referred to as *On-Premise* (OP), denotes the classical integration case (e. g., between SAP ERP and CRM solutions) with moderate message throughput requirements per integration scenario up to several 10,000 msgs/sec. The message formats are still mostly XML-based. In Tab. 2 we summarize the study of real-world scenarios from different integration styles in SAP HCI [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.

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 100,000 msgs/sec, depending on the integration style. The patterns used in these scenarios (cf. Tab. 2) are again mostly MT, CBR, but also *Message Filter* (MF), *Splitter* (SP) and *User-defined Functions* (UDFs).

Cloud-to-Cloud Integration (C2C): The fast-growing field of *Cloud-to-Cloud* (Cloud2Cloud) integration (*ST3*; incl. 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 day[1]. Besides the previously discussed patterns, the most important integration patterns are *Content Modifier* (CM), *Multicast* (MC), e. g., for parallel message processing, and *Content Enricher* (CE), e. g., adding additional data to the message from a data store. Especially in cloud scenarios there are several auxiliary patterns like encoders or decoders, signer, verifier,

[1]Log processing metrics, relevant for message-based integration, visited 02/2016; last update 2012: http://www.solacesystems.com/techblog/deconstructing-kafka

decrypt 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 logistics produce large amounts of messages, while requiring the *Aggregator* (AGG) pattern in addition to the previously discussed patterns to form common, map-reduce-like patterns, such as *Scatter-Gather* (i. e., MC, AGG) and *Composite Message Processor* (i. e., SP, AGG) [13].

General: The scenarios of all discussed integration styles potentially require reliable messaging with service quality guarantees [26], called message delivery semantics. The most common message delivery semantics are *At-least-Once* (i. e., ALO; *Message Redelivery* pattern [26]), *Exactly-Once* (i. e., EO; ALO with *Idempotent Receiver* pattern [12]), and *EO-In-Order* (i. e., EOIO; EO with *Resequencer* pattern [12]).

3.3 Summary

The study shows the relevance of integration patterns along classical and new application integration styles and scenario types. From the currently known routing and transformation patterns, the studied cloud integration scenarios mainly use the content-based router, message translator, splitter, content modifier, and content enricher. Consequently, these patterns are considered as relevant for EIPBench. In addition to these patterns, new integration scenarios require even more "data-aware" operations (e. g., CM, CE) and patterns for parallel, map-reduce-style processing (e. g., MC, SP, AGG). Furthermore, we include other branching patterns (i. e., multicast and recipient list) for variety and scratch the content modifier, due to its similarity to the message translator. Finally, we include all patterns from the message delivery semantics discussion. In addition, many scenarios use user-defined functions (UDF) which indicates that current patterns do not satisfy all requirements.

Especially new domain-operations (e. g., arithmetic operations) and the flexibility required for more diverse message formats and new scenarios introduce new challenges. Notably, less verbose message formats like JSON are used, thus influencing the selection of the EIPBench message format.

4. BENCHMARK DESIGN

In this section we discuss general EIPBench design choices, for the message format and macroscale factor criteria (incl. concurrent users) from Tab. 1. We use configurable scale factors for the EIP benchmark definitions to allow the specification of a "data-aware" message processing benchmark. Subsequently, the message generation and general, macroscale factors are discussed. The EIP microscale factors are discussed in the next section.

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

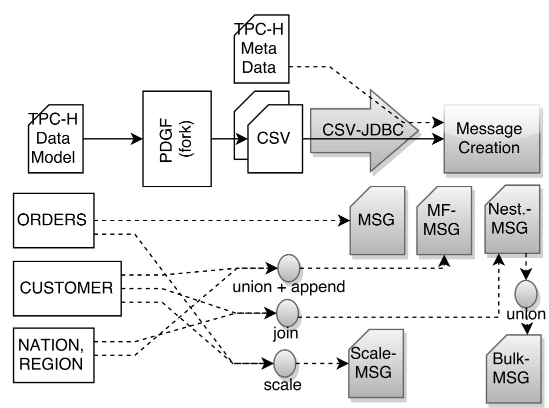


Figure 2: Extended PDGF-based message creation.

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:

[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 $MSG_{Ord}.B := \{msgId, type\} \oplus OBJ_{Ord}.fields$, while $MSG_{Ord}.[H|A] := \emptyset$, where $OBJ_{Ord}.fields$ are the fields of the order object. Analogously, customer MSG_{Cust}, nation MSG_{Nat}, and region MSG_{Reg} are defined.

While the first tranche 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 (MSG_{MF}) is defined as a list of potentially different message models MM. For EIPBench MSG_{MF}^{CNR} messages with one `CUSTOMER` record and all `NATION` and `REGION` records are created. Hence, the source messages are defined as $MSG_{MF}^{CNR} := \{msgId, type\} \oplus OBJ_{Cust}.fields \oplus OBJ_{Nat}.fields \oplus OBJ_{Reg}.fields$. 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 `CUSTOMER` and `NATION` relation are replaced within the customer record beforehand, leading to nested 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 $MSG_{Nest}^{CN}(n) := \{msgId, type\} \bigoplus_1^n (OBJ_{Cust}.fields \backslash C\_NATIONKEY) \oplus (OBJ_{Nat}.fields \setminus N\_NATIONKEY)$, where n defines the number of nested customer entries as message content.

To support a message scaling over orders with a list of nested customer records $MSG_{Nest}^{C}(n)$, we define $MSG_{scale}^{OC} := MSG_{Nest}^{OC}(m)$, with $m > 1$. Messages of several hundred MB, e. g., as required for Financial Service Network Cloud-OP messaging, not only help to test the message throughput but also the capability of an integration system to bigger data volumes (i. e., not only "fast", but "big" data).

Each single message model can be stashed into a message collection $Col_\lambda(MSG)$, where $MSG := \{MSG_{Ord}, MSG_{MF}, MSG_N, MSG_{scale}\}$ and collection size λ, which specifies the number of messages within the collection.

4.2 Macroscale Factors

The integration patterns need to scale along different dimensions. Consequently, we define the following *macroscale factors*: (i) messages with different user contexts (i. e., concurrent users), (ii) micro-batching, and (iii) message size (implicit and explicit).

The scale factor *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:

$$scale_{cu}(\omega) = 2^\omega \tag{1}$$

For example, when transferring the settings of the ESB Performance benchmark [1], ω varies between 0 and 11. In our experiments, we use $0 \le \omega \le 6$, which already sufficiently shows the impact of this scale factor to answer question *Q5*.

Furthermore, we define the *micro-batching* (ii). With this parameter we intend to show the benefit of batched processing 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 collections of messages ($Col_\lambda(MSG)$) instead of single messages [25], called *micro-batching*. Currently only the patterns discussed in [25] are micro-batch enabled, e. g., message transformation. The batch scale levels β, with $0 \le \beta$, denote the number of distinct messages in one message collection $Col_\lambda(MSG)$ as defined:

$$\lambda := scale_{batch}(\beta) = 2^\beta \tag{2}$$

The ESB Performance benchmark [1] does not specify such a test. In EIPBench β is configurable, and we choose β as $0 \le \beta \le 10$ 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 challenge by constructing larger message sizes (cf. *Q5*) of multi-format MSG_{MF} and nested $MSG_{Nest}(n)$ messages, as they are used by various applications. For a given message of type θ we define a function $size([msg|obj]_\theta)$, which determines the message size in kB. For instance, the size of MSG_{Ord} is approximately $0,354$ kB and for the nested customer object $size(OBJ_{cust}) \approx 0,293$ kB. For the size of nested messages, the type of the nested business object obj_τ can be specified. The extended function $size(msg, \theta, \tau)$ calculates the size inclusive the nested object. Since the nesting is calculated, by a foreign key fk relation between the business objects, a function $size([msg|obj]_\theta, fk)$ returns the size of a message or object without the foreign key field. The generic size calculation of objects and messages is defined as:

$$\begin{aligned} size(MSG'_\theta) &= size(MSG_\theta, fk) \\ &= size(MSG_\theta) - size(fk) \\ size(OBJ'_\tau) &= size(OBJ_\tau, fk) \\ &= size(OBJ_\tau) - size(fk) \end{aligned} \tag{3}$$

Now, the size of these messages is scaled through parameter η, with $1 \le \eta \le 20$. 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 η, there is another message size factor γ, which helps to increase the number of business objects of a scale level: Equation (4) brings all previous pieces together and shows the generic calculation of the message size of a scaled message MSG_{scale} for a particular scale level η.

$$MSG_{scale}^\eta = size(MSG'_\theta) + \eta \cdot \gamma \cdot size(OBJ'_\tau) \tag{4}$$

For instance, the concrete scale factor constant in EIPBench is $\gamma = 6$. For the nested messages $MSG_{Nest}(n)$, n is defined as $n := \gamma \cdot \eta$. Concrete values, e. g., for simple customer messages in EIPBench with $\theta, \tau := Cust$ 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 classification, we identified appropriate message formats and macroscale factors for EIPBench. Considering the focus on "data-aware"

[3]SAP IDoc structure, visited 02/2016: http://help.sap.de/saphelp_46c/helpdata/en/dc/6b824843d711d1893e0000e8323c4f/content.htm

pattern processing the definitions allow for the specification of a comprehensive benchmark. Subsequently the tested patterns are introduced and defined by their microscale factors.

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 $\{B, C, \dots\}$ 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 3: Message Routing (RT) patterns with microscale factors.

Label	Patterns	Description	Scale
RT-1	CBR, MF	Channel cardinality 1:[1|n], $n \in N$ outgoing channels, $m \in N$ (dis-) conjunctive conditions w/ increasing complexity	A: normal, B: $n > 1$, C: $m > 2$
RT-2	CBR, MF	same as *RT-1* on multi-format message with $k \in N$ entries	D: $k > 1$
RT-3	Multicast (MC)	Channel cardinality 1:n, $n \in N$ outgoing channels, parallel processing, stop on exception	A: $n = 1$, B: $n > 1$, and variations
RT-4	Recipient List (RL)	same as *RT-3* with n receiver determinations	A: $n = 1$, B: $n > 1$, and variations
RT-5	Splitter (SP)	message cardinality 1:i, $i \in N$ outgoing messages, parallel processing, stop on exception	$i > 1$, split cardinality, variations
RT-6	Aggregator (AGG)	message cardinality i:1, $i \in N$ incoming messages	completion sizes, aggr. strategies

*Content-based Router, Message Filter (*RT-1, RT-2*).*

The content-based router (CBR) routes one incoming message to exactly one of the n outgoing channels according to the ordered evaluation of n-1 channel conditions that read the message's content. The first condition that evaluates to `true` decides on the outgoing channel, else the message is routed to the nth channel (default channel). The message filter (MF) is a special case of the CBR with a channel cardinality of 1:1, resulting in a "pass or no-pass" decision.
Example: Use different processing for an order with higher `ORDERPRIORITY` and higher prize `TOTALPRICE` with the CBR or filter out messages with `ORDERSTATUS` of "F".
Scale/Variations: number of complex conditions and branches
Implementation: EIPBench evaluates the different micro- and macroscale factors, conditions on MSG_{Ord} for (A–C) and MSG^{CN}_{Nest} for (D).

*Multicast, Recipient List (*RT-3, RT-4*).*

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 MSG_{Ord}, which tests the channel branching behavior (i. e., channel creation).

*Splitter (*RT-5*).*

The splitter (SP) splits an incoming message (with repeating elements) into i outgoing messages to the same receiver using a split condition.
Example: The creation of new messages for each part of a multi-format message MSG_{MF} (incl. foreign key creation) or the separation of a message collection $Col_i(MSG_{Ord})$ into single messages.
Scale/Variations: increasing split cardinalities i
Implementation: EIPBench configures the splitter to (A) split collection of messages MSG_{Ord} into single messages (reverse of aggregator for micro-batching), (B) each entry or section of a message into a single message, (C) take first four fields of message (message id, type, orderkey, custkey) as fixed parts, then split next j elements into j messages and add last one element (comment) to the message as footer.

*Aggregator (*RT-6*).*

The aggregator (AGG) combines j incoming message into one outgoing message, sent to the same receiver using correlation, completion conditions (e. g., size, time) and an aggregation strategy.
Example: The union of two relations to one multi-format message MSG_{MF} or the stashing of messages a message collection.
Scale/Variations: increasing completion size or time.
Implementation: EIPBench configures the aggregator to merge different numbers j of MSG_{Ord} messages.

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.

Label	Patterns	Description	Scale
MT-1	MT	program with n:m distinct field mappings, each with a directed operator tree of size i	(A) $n, m \leq 10$ and $i = 1$
MT-2	CF	filter o fields	$o \geq 1$
MT-3	CE	enrich message with j new fields or complete structures	$j > 0$; nesting and multi-format variants.

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*).*
Massage 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.

Label	Patterns	Description	Scale
MDS-1	MRoE	redeliver message on failure $o \in N$ times, (non-) original message	A: $o = 1$, B-F: $32 \geq o > 1$; variants
MDS-2	RS	sequence of $n \in N$ messages, sequence identifier	$n \geq 1$; A: $n = 10$
MDS-3	IP	filter duplicate messages $m \in N$ "in-memory"	A: $m = 0$, B: $100,000 \geq m > 0$

*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 (EOIO) [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 MSG_{Ord} messages.

*Resequencer (*MDS-2*).*
Receive sequence of messages (transitively), correlate us-

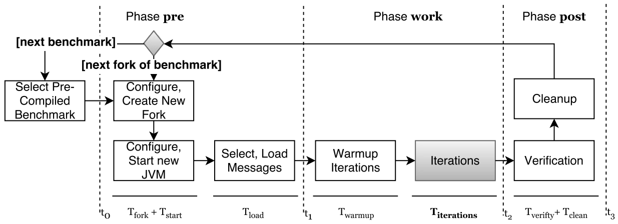


Figure 3: EIPBench execution phases

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 business object creation).
Scale/Variations: number of entries per sequence
Implementation: EIPBench varies the number of sequence entries n, with $n := \{10, 100, 1000, 10000, 100000\}$; implemented for (A) $n = 10$ and resequencing messages according to their `TOTALPRICE` on MSG_{Ord} 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 MSG_{Ord} messages.

6. BENCHMARK IMPLEMENTATION

The EIPBench is executed close to the pattern implementations, potentially even within the same process. Our reference implementation uses JMH[4], 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 producers 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)).

$$\begin{aligned} T_{pre} &= t_1 - t_0 \\ &= T_{fork} + T_{start} + T_{load} \end{aligned} \tag{5}$$

During the `work` phase, the client executes the defined pattern benchmarks on a specified number of isolated and freshly initialized JVM instances, called forks ζ, 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)).

$$\begin{aligned} T_{work} &= t_2 - t_1 \\ &= T_{warmup}(\Phi) + T_{eval}(\Phi) \\ &= m \cdot eval(\varphi) + n \cdot eval(\varphi) \end{aligned} \tag{6}$$

During the evaluation, the selected benchmark is executed, and the discrete throughput values φ 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)).

$$\begin{aligned} T_{post} &= t_3 - t_2 \\ &= T_{clean} + T_{verify} \end{aligned} \tag{7}$$

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 φ of a tested pattern (i. e., T_{eval}). More precisely, T_{eval} is the calculated mean of the individual evaluations $eval(\varphi_i)$, with $i \in I$, for the number of iterations I within one fork (cf. Eq. (8)).

$$\begin{aligned} &T_{eval} = \\ &T_{mean/fork} = \frac{\sum_{i=1}^{n} eval(\varphi_i)}{n} \end{aligned} \tag{8}$$

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 ζ. Equation (9) shows the calculation of the mean for multiple forks. While higher number of forks (i. e., >> 10) leads to increasing overall execution times, the results become more reproducible.

$$\begin{aligned} T_{mean} &= \frac{\sum_{j=1}^{\zeta} T_{mean/fork}(j)}{\zeta} \\ T_{\sigma} &= \sqrt{\frac{\sum_{i=1}^{\zeta} (T_{eval}^{i} - T_{mean})^2}{\zeta}} \end{aligned} \tag{9}$$

In addition to the mean, EIPBench measures a confidence value for the result with a confidence level α of 99% (i. e., confidence interval ci). 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_{ci}.

$$T_{ci} = \begin{cases} T_{mean} - \alpha \cdot \frac{T_{\sigma}}{\sqrt{\zeta}}, & lower. \\ T_{mean} + \alpha \cdot \frac{T_{\sigma}}{\sqrt{\zeta}}, & upper. \end{cases} \tag{10}$$

[4]JMH, visited 04/2015: http://openjdk.java.net/projects/code-tools/jmh/.

Subsequently, the T_{ci} values will be shown as error bars for macroscale plots.

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 `ONC`-iterators instead of JSON objects during T_{Load}. 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 *Q*1–*Q*6, 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., 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 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 branching factor on the throughput can be considerable. An even stronger impact on the throughput comes from more complex routing conditions in RT-1 (C). Hence, as answer to questions *Q1* and *Q2*, the results show a significant impact of multiple branchings and complex routing conditions and let us assume that selectivity estimations on the conditions and re-orderings similar to DB queries should be further investigated (cf. [7]). Particularly, for the TIP/AC implementations, parallel routing condition evaluation could bring performance improvements, however, that would probably require a change of the pattern semantics (cf. [25]).

Table 6: Message throughput of Content-based Routing (RT) and Message Transformation (MT) pattern benchmarks compared to the baseline (BL).

Benchmarks	Scale	Java/AC (early-out)	Java/AC	TIP/AC
BL	–	n/a	300, 837 +/- 8, 252	n/a
RT-1	A (simple)	174, 795 +/- 8, 100	176, 319 +/- 4, 704	179, 528 +/- 5, 485
	B (branching)	158, 838 +/- 3, 002	100, 070 +/- 2, 635	163, 672 +/- 4, 186
	C (complex)	115, 599 +/- 3, 901	98, 237 +/- 2, 261	115, 859 +/- 3, 417
	D (join)	165, 644 +/- 3, 132	–	176, 926 +/- 6, 513
MT-1	A (medium)	n/a	172, 545 +/- 7, 612	193, 378 +/- 4, 407

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_{MF}^{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 *MDS-1* (A–F) for an increasing number of retries o starting with $1 \leq o \leq 32$. The variant "use-original message" (not shown) does not show a significantly different throughput behaviour. Since *MDS-1* MRoE is a "loop" pattern, this test allows insight in the loop-

Table 7: Throughput Benchmarks for message delivery semantics.

Benchmarks	Scale	Java/AC
MDS-1	A (1 redelivery)	$70,585$ +/-$2,323$
	B (2 redeliveries)	$31,649$ +/-$1,131$
	C (4 redeliveries)	$14,774 + / - 513$
	D (8 redeliveries)	$9,456 + / - 268$
	E (16 redeliveries)	$4,906 + / - 139$
	F (32 redeliveries)	$1,995$ +/-85
MDS-2	A (sequence of 10 messages)	$161,918$ +/-$4,883$
MDS-3	A (duplicate after $100,000$)	$172,544$ +/-$6,156$

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

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.

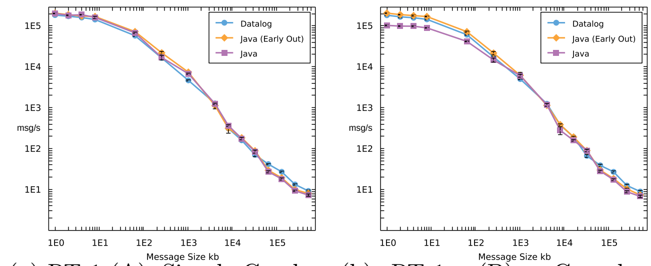
Conceptually, the (transient) idempotent receiver and the message filter patterns are comparable. This is supported by the similar message throughput as shown in Tab. 7 *MDS-3* (A) with a duplication factor of $m = 100,000$ messages.

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.

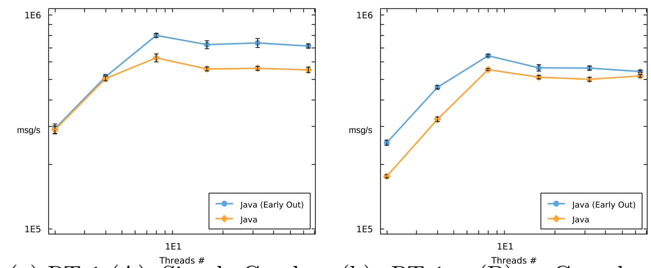
On the impact of increasing message sizes (Q4): The "data-aware" messaging question $Q4$ about increasing message sizes for content-based routing leverages *RT-1* together with messages of type MSG^{OC}_{scale}. Figure 4 shows the immense impact of big messages for *RT-1 (A)* and *RT-1 (B)*. Notably, 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 $scale_{cu}(\omega)$ 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



(a) RT-1 (A): Simple Conds. (b) RT-1 (B): Complex Conds.

Figure 4: $Q4$: Content-based Router (size scaling).



(a) RT-1 (A): Simple Conds. (b) RT-1 (B): Complex Conds.

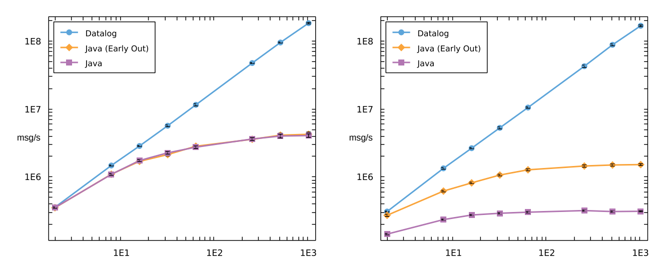
Figure 5: $Q5$: Content-based Router (conc. users).

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-



(a) RT-1 (A): Simple Conds. (b) RT-1 (B): Complex Conds.

Figure 6: $Q6$: Content-based Router (batch scaling).

sponsible 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

[1] AdroitLogic. ESB performance. http://esbperformance.org/, 2013.

[2] A. Arasu, M. Cherniack, E. F. Galvez, D. Maier, A. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear road: A stream data management benchmark. In *Proc. VLDB*, 2004.

[3] M. Böhm, D. Habich, W. Lehner, and U. Wloka. DIPBench: An independent benchmark for data-intensive integration processes. In *Proc. ICDE Workshops*, 2008.

[4] M. Böhm, D. Habich, W. Lehner, and U. Wloka. DIPBench toolsuite: A framework for benchmarking integration systems. In *Proc. ICDE*, 2008.

[5] D. Chappell. *Enterprise Service Bus*. O'Reilly, 2004.

[6] U. Dayal, C. Gupta, R. Vennelakanti, M. Vieira, and S. Wang. An approach to benchmarking industrial big data applications. In *Big Data Benchmarking*. 2015.

[7] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. Fischer. Path sharing and predicate evaluation for high-performance XML filtering. *ACM Trans. Database Syst.*, 28(4):467–516, Dec. 2003.

[8] M. Fowler and J. Lewis. Microservices. 2014.

[9] L. George. *HBase: The Definitive Guide*. O'Reilly, 2011.

[10] O. Gmelch. *User-Centric Application Integration in Enterprise Portal Systems*. EUL-Verlag, 2012.

[11] J. Hammer, M. Stonebraker, and O. Topsakal. THALIA: test harness for the assessment of legacy information integration approaches. In *Proc. ICDE*, 2005.

[12] G. Hohpe and B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2003.

[13] C. Ibsen and J. Anstey. *Camel in Action*. Manning, 2010.

[14] A. Joshi, R. Nambiar, and M. Brey. Benchmarking internet of things solutions. In *Proc. WBDB*, 2014.

[15] D. S. Linthicum. *Enterprise Application Integration*. Addison-Wesley, 2000.

[16] M. R. N. Mendes, P. Bizarro, and P. Marques. A framework for performance evaluation of complex event processing systems. In *Proc. DEBS*, 2008.

[17] M. R. N. Mendes, P. Bizarro, and P. Marques. Towards a standard event processing benchmark. In *Proc. ACM/SPEC WOSP*, 2013.

[18] S. Newman. *Building Microservices*. O'Reilly, 2015.

[19] M. Poess, T. Rabl, and B. Caufield. TPC-DI: the first industry benchmark for data integration. *PVLDB*, 7(13), 2014.

[20] M. Poess, T. Rabl, M. Frank, and M. Danisch. A PDGF implementation for TPC-H. In *Proc. TPCTC*, 2011.

[21] T. Rabl and C. Baru. Big Data Benchmarking. IEEE International Big Data Conference, 2014.

[22] T. Rabl, M. Danisch, M. Frank, S. Schindler, and H.-A. Jacobsen. Just can't get enough - Synthesizing Big Data. In *Proc. SIGMOD*, 2015.

[23] T. Rabl, M. Frank, H. M. Sergieh, and H. Kosch. A data generator for cloud-scale benchmarking. In *Proc. TPCTC*, 2010.

[24] T. Rabl and H.-A. Jacobsen. Big Data Generation. In *Proc. WBDB*, 2013.

[25] D. Ritter. Towards more data-aware application integration. In *Proc. BICOD*, 2015.

[26] D. Ritter and M. Holzleitner. Integration adapter modeling. In *Proc. CAiSE*, 2015.

[27] D. Ritter and J. Sosulski. Modeling exception flows in integration systems. In *Proc. EDOC*, 2014.

[28] K. Sachs, S. Appel, S. Kounev, and A. Buchmann. Benchmarking publish/subscribe-based messaging systems. In *Proc. DASFAA 2010 Workshops: BenchmarX*, 2010.

[29] K. Sachs, S. Kounev, J. Bacon, and A. Buchmann. Performance evaluation of message-oriented middleware using the SPECjms2007 benchmark. *Performance Evaluation*, 66(8):410–434, Aug 2009.

[30] SAP SE. SAP HANA Cloud Integration content. https://cloudintegration.hana.ondemand.com, 2015.

[31] SPEC. SPEC SOA benchmark. https://www.spec.org/soa/, 2010.

[32] J. Zaleski. *Integrating Device Data Into the Electronic Medical Record: A Developer's Guide to Design and a Practitioner's Guide to Application*. John Wiley & Sons, 2009.

[33] O. Zimmermann, C. Pautasso, G. Hohpe, and B. Woolf. A decade of enterprise integration patterns: A conversation with the authors. *IEEE Software*, 33(1):13–19, 2016.