A Conceptual Framework for API Refactoring in Service-Oriented Architectures

Fabrizio Montesi[1], Marco Peressotti[1], Valentino Picotti[1], and Olaf Zimmermann[2]

[1] University of Southern Denmark {fmontesi,peressotti,picotti}@imada.sdu.dk
[2] University of Applied Sciences of Eastern Switzerland olaf.zimmermann@ost.ch

Abstract. In the lifetime of a service-oriented architecture, the Application Programming Interfaces (APIs) offered by services may need to be refactored in order to adapt to changing business and technical requirements. Previous studies focused on the effects that such API refactorings have on API definitions, with general considerations on related forces and smells. By contrast, the development strategies for realising these refactorings have received little attention. This paper addresses this aspect.
We introduce EMI, a conceptual framework for the implementation of API refactorings. Our framework is designed to elicit the trade-offs and choices that significantly affect the efficiency, maintainability, and isolation of the resulting architecture. We evaluate our framework by evolving a use case through the implementation of several refactorings, illustrating the different implementation choices that can be made. Based on our experience, we illustrate how to derive mechanical recipes for API refactoring that can follow different strategies in our framework.

1 Introduction

In service-oriented architectures, individual (micro)services perform specific functions and interact through well-defined Application Programming Interfaces (APIs) [4]. Over time, service APIs evolve because of changing requirements [5–8]. This prompted the study of *API refactoring*: the modification of interfaces to improve quality attributes, such as efficiency [18, 19].

Recently, a catalogue of API patterns provided a basis for API refactorings [19, 25]. Previous studies focused on the high-level forces (e.g., modifiability) and smells (e.g., high latency) that motivate and guide these refactorings. Conversely, it is yet unexplored how developers are supposed to implement an API refactoring and assess its quality; a research gap that we aim to address.

In this paper we introduce EMI (efficiency, maintainability and isolation), a conceptual framework for assessing the implementation of API refactoring in service architectures (Section 3). EMI centres around two dimensions: 1) *generality*, which assesses the degree of abstraction of the refactored API source code; 2) *distribution*, which elicits where the refactored API source code resides. Realising the combination of both dimensions results in six development strategies,

each representing design choices for the implementation with respective trade-offs. The trade-offs pertain the quality aspects of *efficiency* (E), *maintainability* (M), and *isolation* (I) of the resulting architecture. We score each of these three aspects from 1 to 3 for our strategies, yielding the EMI score for API refactoring. There is no silver bullet: no strategy scores perfectly (9), emphasising the importance of making conscious implementation decisions.

We validate the applicability of our framework by carrying out several API refactorings on an illustrative publication catalogue service (Section 4). Specifically, we apply our six development strategies to the same refactoring: the introduction of the API Key pattern – which rejects requests without a valid key – to a service that offers a catalogue of scientific publications (Section 4.1). We then broaden our study to patterns that do not require behavioural changes, Merge Endpoints and Version Identifier, reaching modular solutions (Section 4.2). Our development is available as a benchmark for future research [12]. A main finding is that our framework can be used to distill systematic recipes for API refactoring, which developers can mechanically apply step by step to achieve an implementation with a declared EMI score (Section 5). As examples, we provide recipes for the introduction of API Key and Pagination patterns.

In summary, we contribute the EMI framework (Section 3), a scheme to guide and assess the implementation of API refactorings, we validate EMI by applying API refactorings to (micro)service architectures (Section 4), and provide canonical recipes to obtain certain EMI scores for architecture evolution (Section 5). We also discuss related work in Section 2, elicit threats to validity in Section 6, and conclude with an outlook on future work in Section 7.

2 Related Work

Our study builds on the reference catalogue of patterns for API design [25], which addresses the challenge of remote API design [23] through peer-reviewed patterns published in the period 2017–2020 [10, 20–22, 24].

API refactoring was previously investigated based on the same catalogue [18, 19]. Those studies focus on architectural considerations and especially *why* and *when* an API pattern should be introduced, considering forces and smells. Differently, our work is the first to investigate *how* an API pattern is implemented. Specifically, we are interested in the different choices regarding the code of a refactoring, and the quality trade-offs that they yield. Another difference is that we study how to refactor both the definition and implementation of an API, whereas previous work focuses only on the definition. Our frameworks can be seen as a refinement of Attribute-Driven Design and Architecture Trade-off Analysis Method, which are concerned with informing and assessing architectural decisions in light of quality attribute requirements [1].

The two axes of generality and distribution in EMI can be seen as systematic organisations of various considerations made in the literature of programming languages and microservices. Methods for high generality include aspect-oriented programming [9] and linguistic abstractions for implementing reusable decorators

(e.g., couriers in Jolie and delegation in Kotlin). The importance of distribution was already partially observed in [13] for the development of circuit breaker – a pattern for increasing resilience [14] – and later acknowledged in security recommendations [3]. The developments in [13] fall under Parametric/Adjacent and Parametric/External in our framework (Section 3). Our interest in the present work is much more general: rather than focusing on a specific use case, we formulate a framework that can be used to reason about any API refactoring. Furthermore, the quality aspects considered here are not considered in [13].

The code in our validation of EMI is written in the service-oriented programming language Jolie [11], for reasons of exposition: Jolie provides native abstractions for the key concerns of API definition and refactoring, like API endpoints and the definition of APIs polymorphic on other APIs (which we use to achieve reusable refactorings) [11, 17]. However, our findings are not tied to Jolie. We refer to alternative technologies in Section 6.

3 The EMI Framework for API Refactoring

We now present our conceptual framework for API refactoring – the EMI framework. It is depicted in Table 1. We explain it in the rest of this section.

		Distribution		
		Internal	Adjacent	External
Generality	Parametric	E ★★☆ M ★★☆ I ★☆☆	E ★★☆ M ★★★ I ★★☆	E ★☆☆ M ★★★ I ★★★
	Ad-hoc	E ★★★ M ★☆☆ I ★☆☆	E ★★☆ M ★★☆ I ★★☆	E ★☆☆ M ★★☆ I ★★★

Table 1. The EMI framework.

API refactoring changes both an interface and its implementation, while improving at least one quality attribute [18, 19]. This may affect the external behaviour of an API observed by clients, without altering its capabilities.

We introduce some terminology. In the remainder, we refer to the changes introduced by an API refactoring as the *new functionality*, bearing in mind that such functionality does not alter the feature set of the API [19]. In line with the API domain model of [25], we consider an API to be a collection of operations that can be invoked by clients. Services can offer APIs through *endpoints*, which expose operations at a designated location according to a given transport protocol. We call such services *API providers*. We distinguish the API and implementation that we start from and then end up with after a refactoring with the prefixes *original* and *refactored*.

3.1 Generality and distribution

The EMI framework focuses on two dimensions to assess the quality attributes of the implementation of an API refactoring: *generality* and *distribution*.

The *generality* dimension concerns whether the implementation of the new functionality depends on or abstracts from the definition of the original API. We identify two possibilities.

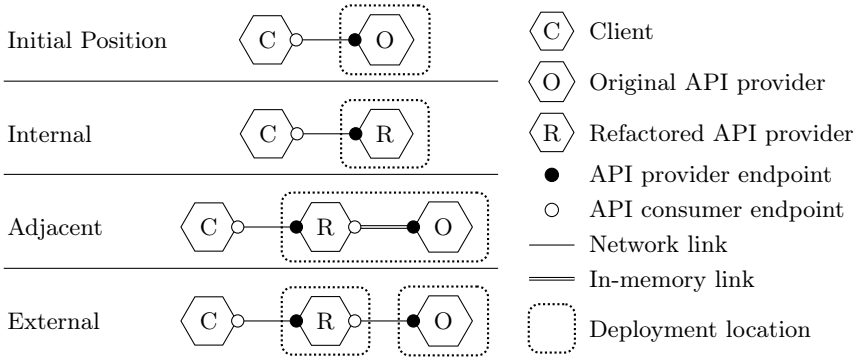

Fig. 1. Possible choices for distribution.

Ad-hoc The code of the new functionality depends on hardcoded information on the names, types, or behaviours of the operations in the original API.

Parametric The code of the new functionality abstracts from the names, types, or behaviours of the operations in the original API.

Generality serves as an indicator of the logical coupling between the new code and the old. It is significant because API patterns provide, at least conceptually, reusable solutions to recurring problems. Thus, in a way, generality indicates how much this reusability is achieved in real code.

The *distribution* dimension concerns where the code for the new functionality is located in relation to the original API provider and its clients. There are three possibilities, depicted in Fig. 1.

Internal The code of the new functionality is mixed with the code of the original API provider. Thus, they share state. After the refactoring, the original API provider becomes the refactored API provider.

Adjacent The refactored API provider is a separate service. They have separate state and are executed independently, but they are deployed such that they can communicate efficiently through local resources (local memory channels, inter-process communication, loopback network interfaces, etc.).

External The refactored API provider is a separate service. It is deployed remotely from the original, and thus can communicate with it only through network communication.

3.2 EMI scores

The combination of the axes of generality and distribution gives rise to six possible development strategies, each presenting different trade-offs. To help in navigating these trade-offs, we score each strategy on three quality attributes using a three-level scale (★☆☆, ★★☆, or ★★★): efficiency (E), maintainability (M), and isolation (I). We explain each score next.

Efficiency (E)

E ★★★ The new functionality is implemented optimally, with no extra overhead caused by design choices.
E ★★☆ Design choices cause extra overhead in terms of local resources (memory, local communication, etc.).
E ★☆☆ Design choices cause extra overhead in terms of remote resources (e.g., network communication).

Maintainability (M)
M★★★ The original and refactored API providers can be maintained independently.
M★★☆ The implementations of the new functionality and the original API provider are separate but tightly coupled.
M★☆☆ The implementations of the new functionality and the original API provider are completely mixed.

Isolation (I)
I ★★★ The original and refactored API providers do not share any local resources for their execution.
I ★★☆ The original and refactored API providers share execution resources (e.g., CPUs, memory), but do not share state and interact purely by means of the original API.
I ★☆☆ The new functionality and the original API implementation share internal program state (e.g., stack, variables, heap).

The levels of these scales are intentionally broad, in order to avoid being tied up by very specific technological details. This is in line with the technology-agnosticism of microservices [4].

3.3 Scoring development strategies

We end the presentation of our framework with an analysis that justifies the EMI scores of each development strategy, referring also to examples of API patterns and technologies where relevant.

Ad-hoc/Internal (E ★★★ M★☆☆ I ★☆☆) This is the most efficient strategy, because the new functionality is implemented directly by changing the behaviour of the original API provider. Thus, the code of the new functionality encounters no unnecessary overhead in integrating with the original implementation. For example, introducing the PAGINATION pattern to an implementation that queries a database gives the possibility to modify the query in order to retrieve fewer results – those for the page being requested. For the very same reasons, however, this is also the least maintainable and isolated choice, since the new code is mixed and shares all resources with the old code. Examples of this strategy are shown in Sections 4.1 and 5.

Parametric/Internal (E ★★☆ M★★☆ I ★☆☆) This strategy trades some efficiency for maintainability by abstracting from the operation names and behaviours of the original API. The code of the new functionality can be reused across different APIs, but has limited access to changing their behaviour: the new functionality can only intercept, modify, and conditionally forward request

and response message to and from the original implementation. Examples of this strategy are implementations adopting Java Servelet Filters or Express middleware functions.

Ad-hoc/Adjacent (E ★★☆ M★★☆ I ★★☆) Compared to Ad-hoc/Internal, implementing the new functionality in a separate component trades some efficiency for partially improved maintainability and isolation. However, the new functionality remains coupled with the original API (ad-hoc), so changes to the original API require updating the refactored API provider, too. Thus, maintainability is still not ideal. Improved isolation comes at the cost of some overhead in the interaction between the refactored and original API providers. Efficiency is further affected by the new functionality not having access to changing the internal behaviour of the original API provider. This strategy can be implemented with, for example, the sidecar pattern, the ambassador pattern, or Jolie's embedded services (see Section 4.1).

Parametric/Adjacent (E ★★☆ M★★★ I ★★☆) This strategy has the same efficiency and isolation characteristics as the previous one, but greatly improved maintainability by decoupling the implementation of the new functionality from the operation names and message types of the original API. The sidecar and ambassador patterns are again useful to implement this strategy. Jolie's embedded services combined with couriers and interface extenders (see Section 4.1) offer an interesting solution, because the refactored API can be automatically and statically computed.

Ad-hoc/External (E ★☆☆ M★★☆ I ★★★) The strategy with the highest level of isolation, since the new functionality interacts with the original API provider only via remote access and for this reason, this is the least efficient one. This strategy does not achieve the highest maintainability score due to the coupling between the new functionality and the original API. This strategy can be implemented by developing a proxy service offering the refactored API and forwarding each invocation to the original API provider when appropriate.

Parametric/External (E ★☆☆ M★★★ I ★★★) This strategy has the same efficiency and isolation scores as the previous one, but also the highest maintainability score for the same reason given for Parametric/Adjacent.

No strategy scores a perfect nine. The reason lies in the unavoidable tension between efficiency and isolation: optimal efficiency requires sharing resources, which prevents achieving optimal isolation.

4 EMI at Work

In this section, we validate our framework by applying it in depth – exploring all our strategies for a single pattern – and in breadth – applying selected strategies to other patterns. The code of our examples is available online [12].

4.1 API Key in Jolie

We illustrate the use of our framework by applying each strategy to a concrete use case: the introduction of the API Key pattern to a service managing a catalogue

of scientific publications. API KEY identifies clients through respective unique keys, which must be included in requests.

We code our examples in Jolie [11]. In Jolie, the operations and message types of an API are defined as an **interface**. The next interface defines the API of our publication catalogue service.

```
type Publications: { publications*: Publication }
type Publication: Proceeding | InProceeding | Article
interface PubCatInterface {
  RequestResponse:  getAuthorPubs( {authorId: string} )( Publications )
                    getConfPubs( {confId: string} )( Publications )     }
```

`PubCatInterface` comprises two operations: `getAuthorPubs`, which expects the unique identifier of an author (as the field `authorId` of the request message) and returns all their publications (message type `Publications`); and `getConfPubs`, which given a conference identifier (`confId`) returns the publications of that conference. The type `Publications` describes a record with a field `publications` containing zero or more (`*`) values of type `Publication`. `Publication` is the union of three types (omitted) corresponding to proceedings (`Proceeding`), papers in proceedings (`InProceeding`), and journal articles (`Article`).

Interfaces are offered to clients by defining an **inputPort**, Jolie for an endpoint that accepts remote invocations. An input port is defined inside of its enclosing **service** and commits to a concrete **location** and transport **protocol** (HTTP, SOAP, binary protocols, etc.). The definition of our publication catalogue service is given next (abstracting some internal implementation details).

Listing 1. Original API Provider.

```
1 /* Service definition */
2 service PubCat {
3   /* API Endpoint */
4   inputPort ip {  location: "socket://localhost:8080"
5     protocol: http { format = "json" } interfaces: PubCatInterface }
6   /* Behaviour */
7   main {
8     [ getAuthorPubs( request )( response ) { /* fetch the data from db */ } ]
9     [ getConfPubs( request )( response ) { /* fetch the data from db */ } ] } }
```

In Lines 4 to 5, service `PubCat` exposes `PubCatInterface` on TCP port 8080 over the HTTP protocol with message payloads in JSON format. Its implementation (Lines 7 to 9) consists of an *input choice* that can react to any invocation of the operations it lists. Each branch in the choice has the form `[ op( req )( resp ){ B } ]` where `op` is the name of the operation, `req` and `resp` are the input and output parameters, and B is the code block computing the response.

Introducing the API KEY pattern requires extending request message types with an additional field `apiKey` (storing the key as a `string`) and declaring a faulty response message `NotAuthorised` for invocations with invalid keys. The refactored API is given next.

Listing 2. Refactored API.

```
1 interface PubCatInterfaceWithAPIKey {
```

```
  RequestResponse:
    getAuthorPubs( {authorId: string, apiKey: string} )( Publications )
      throws NotAuthorised
    getConfPubs( {confId: string, apiKey: string} )( Publications )
      throws NotAuthorised }
```

The refactoring of service `PubCat` and its interface `PubCatInterface` to obtain a service exposing the refactored API `PubCatInterfaceWithAPIKey` can be accomplished following any of the strategies outlined in Section 3. We discuss the most interesting aspects of these applications. To see how the code looks in detail for each strategy, the reader can consult our benchmark [12].

Ad-hoc/Internal We directly modify the code of both the original interface `PubCatInterface` and the service `PubCat`. `PubCatInterface` becomes the refactored API `PubCatInterfaceWithAPIKey` above. In `PubCat`, instead, the implementation of each operation is edited to validate the API key in the request message.

```
service PubCat {
  inputPort ip { /* ... */ interfaces: PubCatInterfaceWithAPIKey }
  main {
    [ getAuthorPubs( request )( response ) {
      /* check validity of request.apiKey */
      if( isKeyValid ) { /* fetch the data from db */ }
      else { throw NotAuthorised( /* fault data */ ) } } ]
    [ getConfPubs( request )( response ){ /* as for getAuthorPubs */ } ] } }
```

Ad-hoc/External We introduce a new service, `PubCatWithAPIKey`, with an endpoint exposing the interface `PubCatInterfaceWithAPIKey`. This service acts as an adapter for the original API provider, `PubCat`, which remains unchanged. The implementation of the API Key pattern is entirely confined to the new service, which forwards valid invocations to `PubCat`. This requires the service `PubCatWithAPIKey` to declare an *output port* (Line 2) pointing to the API endpoint of `PubCat`. Its implementation (Lines 4 to 11) consists of an input choice where each operation checks the validity of the key in the request (`request.apiKey`). If the key is valid, then the key is erased from `request` (Line 8) before invoking the original operation `getAuthorPubs@pc` to obtain the intended `response`. Otherwise, the service replies with a faulty `NotAuthorised` message. Although the implementations of refactored and original API providers are separate, they must be kept in sync wrt future changes to the API, resulting in a negative impact to maintainability.

Listing 3. Ad-hoc/External refactored API provider.

```
service PubCatWithAPIKey {
  outputPort pc { /* PubCat endpoint */ }
  inputPort ip { /* ... */ interfaces: PubCatInterfaceWithAPIKey }
  main {
    [ getAuthorPubs( request )( response ) {
      /* check validity of request.apiKey */
      if( isKeyValid ) {
        undef( request.apiKey ) /* remove API key before forwarding */
        getAuthorPubs@pc( request )( response ) /* forward call */
      } else { throw NotAuthorised( /* fault data */ ) } } ]
    [ getConfPubs( request )( response ) { /* as for getAuthorPubs */ } ] } }
```

Parametric/External To eliminate the coupling between refactored and original API providers, we leverage the Jolie language construct of an *interface extender*, which uniformly extends the types of all operations in an API. The extender `APIKeyExtender` defined in Listing 4 adds the `apiKey` field to all (`*`) request messages and `NotAuthorised` as a new potential faulty response. `APIKeyExtender` precisely describes the changes we have to apply to `PubCatInterface` in order to obtain `PubCatInterfaceWithAPIKey`.

Listing 4. Parametric/External refactored API provider.

```
1  interface extender APIKeyExtender {
2    RequestResponse: *( {apiKey:string} )( void ) throws NotAuthorised
3  }
4  service PubCatWithAPIKey {
5    outputPort pc { /* PubCat endpoint */ }
6    inputPort ip { /* ... */ aggregates: pc with APIKeyExtender }
7    courier ip {
8      [ interface PubCatInterface( request )( response ) {
9        if( isKeyValid ) { forward( request )( response ) }
10       else { throw NotAuthorised( /* fault data */ ) } } ] } }
```

Service `PubCatWithAPIKey` now uses the interface extender to define its API: input port `ip` **`aggregates`** `pc` **`with`** `APIKeyExtender` (Line 6), which instructs Jolie to forward messages for the API of `pc`, extended with `APIKeyExtender`, to `pc`.

Messages forwarded by means of aggregation (applications of **`aggregates`**) can be intercepted by means of a **`courier`** block. A courier is a piece of code attached to an input port, which gets executed whenever one of the input port's operations is invoked. The courier at Lines 7 to 10 implements the API Key pattern for all operations of the interface `PubCatInterface`. Unlike a regular input choice, a courier can be parametric over the operation names of an interface:

```
[ interface PubCatInterface( request )( response ){ B } ]
```

where B is the code that is executed on each invocation of an operation of `PubCatInterface` on input port `ip`, and which can then decide whether to **`forward`** the request to the `PubCat` service, or return the error message `NotAuthorised`. The **`forward`** primitive automatically removes fields added by any interface extenders, so messages to `pc` are well-typed.

Adjacent strategies Jolie supports running separate services in the same application with its native **`embed`** primitive: services `PubCat` and `PubCatWithAPIKey` can be implemented as Adjacent by simply changing their deployment configuration. First, the service `PubCat` is promoted to an in-memory service by changing its location to `"local"` (Listing 1, Line 4). Then, we make the refactored API provider, `PubCatWithAPIKey` (Listings 3 and 4), embed the original `PubCat`: replacing the **`outputPort`** declaration (Lines 2 and 5) by the statement **`embed`** `PubCat` **`as`** `pc` instructs the Jolie runtime to load the service `PubCat` alongside `PubCatWithAPIKey` and make it reachable via an in-memory channel through the output port `pc`. These linguistic features allow for easily switching Jolie codebases between the Adjacent and External columns of the EMI framework, changing the deployment

strategy based on performance considerations (*i.e.*, trade network overhead for CPU and memory consumption).

4.2 Other patterns: Merge Endpoints and Version Identifier

We now illustrate how to introduce two other patterns: Merge Endpoints and Version Identifier. Differently from API Key, these patterns are fully architectural, in the sense that they do not introduce behavioural changes but rather affect only how APIs are accessed. We apply the Parametric/External strategy for both cases.

Merge Endpoints exposes the operations of two endpoints through a single endpoint. Suppose, for example, that we have a `PubCat` service for a publication catalogue and a `CitInd` service for citation indexing. We develop a new service, `PublicationIndex`, that merges their APIs by using aggregation.

```
service PublicationIndex {
  outputPort pc { // publication catalogue
    location: /* ... */ protocol: /* ... */ interfaces: PubCatInterface }
  outputPort ci { // citation index
    location: /* ... */ protocol: /* ... */ interfaces: CitIndInterface }
  inputPort ip { location: /* ... */ protocol: /* ... */  aggregates: pc, ci } }
```

Note that aggregation requires the operations of the aggregated ports to have distinct names, which is in line with the pattern here. If this is not the case, one can use the other Jolie feature of redirection, explained in the next case.

Version Identifier exposes two (or more) different versions of the same API under a single endpoint. Here aggregation does not work, because the operation names in two versions of the same API likely overlap. Jolie solves this problem by offering the APIs under different *resource paths* at the same physical endpoint. In the next example, input port `ip` offers `PubCatInterfaceV1` under path `v1` and `PubCatInterfaceV2` under path `v2`. Assuming that a client reaches the refactored API provider at location `pubcat.com`, this means that version 1 will be accessible at location `pubcat.com/v1` and version 2 at location `pubcat.com/v2`.

```
service PubCatWithAPIKey {
  outputPort pcv1 { location: /* ... */ protocol: /* ... */
    interfaces: PubCatInterfaceV1 }
  outputPort pcv2 { location: /* ... */ protocol: /* ... */
    interfaces: PubCatInterfaceV2 }
  inputPort ip { location: /* ... */ protocol: /* ... */
    redirects: v1 => pcv1, v2 => pcv2 } }
```

This approach does not alter the original (versions of) the APIs, by distinguishing between versions based on the accessed location. Therefore, clients just need to be connected to the right location. An alternative to this approach is to extend the request types of all operations with a version identifier. However, this would require updating the clients to include this information. Furthermore, response types would become less precise, since they would need to accommodate the possible responses across all versions.

5 API Refactoring Recipes

In this section, we illustrate how our framework can be used to distill recipes that can be followed mechanically by programmers to apply an API refactoring. We cover a parametric implementation of the API KEY pattern and an ad-hoc implementation of the PAGINATION pattern. The latter is representative of situations where efficiency requires big sacrifices in maintainability and isolation.

We start with our recipe for API KEY.

Refactoring recipe: Introduce API KEY (Parametric)

Intent. Introduce the API KEY pattern by means of a dedicated service that is parametric on the original API.

Participants and Preconditions.

1. Participant: A Jolie *service*, say `Original`, exposing the API subject to refactoring as an interface, say `OriginalAPI`.
2. Precondition: `Original` offers `OriginalAPI` through an *input port* `OriginalInputPort`.

Refactoring steps.

1. Introduce an *interface extender* `APIKeyExtender` that:
   (a) Extends the request message with a field `apiKey` holding an `API Key`.
   (b) Adds a faulty response message `NotAuthorised`.
2. Introduce a new *service* `OriginalWithAPIKey`:
   (a) Introduce a new *output port* `original`.
   (b) Choose between:
   Choice 1 (External): Configure output port `original` (at `OriginalWithAPIKey`) and input port `OriginalInputPort` (at `Original`) so that they communicate via the network.
   Choice 2 (Adjacent): Configure output port `original` (at `OriginalWithAPIKey`) and input port `OriginalInputPort` (at `Original`) so that they communicate via local memory.
   (c) Introduce an *input port* `ip` that aggregates the output port `original` and extends it with `APIKeyExtender`.
   (d) Introduce a *courier* for `ip` that intercepts all operations of `OriginalAPI` and:
      i. Checks the validity of the `API Key`.
      ii. If the key is valid, forwards the request to `original`.
      iii. Otherwise, if the key is invalid, replies with the `NotAuthorised` response.

Postconditions.

1. Invoking any operation `op` at `OriginalWithAPIKey` with a valid `API Key` gives the same `response` message as invoking `op` at `Original` without an `API Key`.
2. Invoking any operation `op` at service `OriginalWithAPIKey` with an invalid `API Key` results into an `NotAuthorised` message.

3. Service `OriginalWithAPIKey` becomes the only client of service `Original`.

Discussion and EMI scoring. This recipe yields a parametric implementation, giving maintainability score M★★★. Choice 1 introduces network overhead, giving E ★☆☆ and I ★★★, while Choice 2 does not, yielding E ★★☆ and I ★★☆. We get the following possible EMI scores:
Choice 1 (External): E ★☆☆ M★★★ I ★★★.
Choice 2 (Adjacent): E ★★☆ M★★★ I ★★☆.

We now present a recipe for the Pagination pattern. Pagination allows clients to retrieve smaller portions ('pages') of large data sets. The aim is to improve network and memory utilisation; this also addresses the stability antipattern of providing responses of unbounded size [14]. There are four variants of this pattern, corresponding to four different ways of identifying the page that the client wants [24, 25]. Here, we implement the offset-based version.

Refactoring recipe: Introduce Pagination (Ad-hoc/Internal)

Intent. Introduce the offset-based Pagination pattern for an operation.

Participants and Preconditions.

1. Participant: A Jolie *service*, say `Original`, exposing the operation subject to refactoring, say `op`, as part of an interface, say `OriginalAPI`.
2. Precondition: `op` is a retrieval operation whose response type contains an ordered collection of items to be paginated.

Refactoring steps.

1. Change the definition of `op` in `OriginalAPI` to:
   (a) Extend the *request type* with metadata fields specifying the offset of the requested page, the `limit` of items per page, and the `sort-criterion`, if more than one order exists for items in the collection;
   (b) Extend the *response type* with fields describing the response page such as the page number `offset`, items per page `limit`, `sort-criterion`, and `total` number of pages.
   (c) Add a faulty response message `InvalidPageRequest` in case of invalid page metadata.
2. Change the implementation of `op` to:
   (a) Validate the page metadata fields (and reply immediately with `InvalidPageRequest` in case of failure).
   (b) Paginate the requested data, possibly by leveraging features of the database query language (like `OFFSET` and `LIMIT` for SQL).
   (c) Reply with the requested page and its metadata.

Postconditions.

1. Calling `op` to request page with a given `offset` and size `limit` results into the items of the `collection` returned by the original `op` from position `offset * limit` to position `offset * limit + limit`.

Discussion. Delegating the pagination to the query language of the database in use achieves efficiency score E ★★★. However, since it also modifies the implementation of the specific operation, we obtain maintainability M ★☆☆ and isolation I ★☆☆. The overall EMI score is therefore E ★★★ M ★☆☆ I ★☆☆.

Considerations on alternative implementations. The design smells that motivate the introduction of the PAGINATION pattern are about poor efficiency and thus the Ad-hoc/Internal strategy is a natural choice. If Ad-hoc/Internal is undesirable, other strategies can still be adopted at the cost of high decreases in efficiency. The key problem is distribution. Choosing an Adjacent strategy would still imply that the original API provider fetches all data from its database, but at least this would be 'cut' by the refactored API provider before it is sent back to clients. The same holds for an External strategy, but in this case we would pay also the cost of network communication (of the whole data set) between the refactored and original API providers.

6 Threats to Validity

Our validation of EMI has two main limitations, discussed next.

1. We have explored the application of EMI through a single language (Jolie). There are many other languages and frameworks for programming service-oriented systems, including Spring Boot, Express for Node.js, Ballerina [16], and WS-BPEL [15]. The mechanics and behaviours offered by Jolie's primitives can be achieved in these technologies, sometimes requiring the use of a framework. For example, aggregation and redirection – Jolie for merging and redirecting endpoints – can be implemented via the routing mechanism in Express. We are thus confident that our validation can be reproduced beyond Jolie. However, this requires a dedicated and systematic study, which our benchmark offers a good starting point for.
2. Our validation consists of a sample of four refactoring patterns out of the 22 identified by the Interface Refactoring Catalogue. We have selected this sample because of two main reasons: (i) the API design patterns that they introduce are widely known, and (ii) they are illustrative of the trade-offs between the EMI quality aspects since they do not have a clearly optimal implementation strategy. Nonetheless, the selection remains in part subjective and thus it calls for an exhaustive study of the remaining 18 patterns.

7 Conclusion

We have introduced the EMI framework, the first conceptual framework for navigating the implementation aspects of API refactorings. While broad and technology-agnostic, our scores are informative when it comes to key design decisions on the implementation of API patterns.

Our study opens up at least three interesting lines of future work.

First, in line with previous work [19], we have focused on presenting API refactorings that *add* a pattern. However, our Adjacent and External strategies make it immediate to *remove* a pattern later on. We think that enabling the modular activation and deactivation of patterns is an interesting direction.

The second line of future work deals with exploring additional aspects on top of efficiency, maintainability, and isolation. These aspects are in line with a previous survey on what qualities are important in practice, but there are also others that merit consideration, like scalability and usability [2, 18]. We think that scalability would be a first natural extension of our framework, as it is closely related to efficiency and isolation but not completely captured by them.

Lastly, it would be interesting to extend our evaluation – which focuses on applicability – to a systematic user study, involving practitioners in the field of API design and implementation. This would allow us to better understand how easy it is to use EMI for (i) communicating the trade-offs between different strategies, (ii) choosing appropriate strategies, and (iii) facilitating the implementation of API refactorings (possibly for people with different skill levels).

Acknowledgements We thank Sandra Greiner for the useful discussions. This work is partially supported by Independent Research Fund Denmark (grant no. 0135-00219), Villum Fonden (grant no. 50079) and by the European Union (ERC, CHORDS, 101124225). Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

References

1. L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice.* Addison-Wesley, 3rd edition, 2012.
2. J. Bogner, P. Wójcik, and O. Zimmermann. How do Microservice API Patterns impact understandability? A controlled experiment, 2024.
3. R. Chandramouli. *Security Strategies for Microservices-based Application Systems.* National Institute of Standards and Technology, 2019.
4. N. Dragoni, S. Giallorenzo, A. Lluch-Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina. Microservices: Yesterday, today, and tomorrow. In M. Mazzara and B. Meyer, editors, *Present and Ulterior Software Engineering*, pages 195–216. Springer, 2017.
5. M. Erder and P. Pureur. *Continuous Architecture: Sustainable Architecture in an Agile and Cloud-Centric World.* Elsevier Science, 2015.
6. M. Erder, P. Pureur, E. Woods, and a. O. M. C. Safari. *Continuous Architecture in Practice: Software Architecture in the Age of Agility and DevOps.* Addison-Wesley, 2021.
7. N. Ford, R. Parsons, and P. Kua. *Building Evolutionary Architectures: Support Constant Change.* O'Reilly Media, 2017.
8. M. Fowler. *Patterns of enterprise application architecture.* Addison-Wesley, 2012.

9. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *ECOOP'97 - Object-Oriented Programming, 11th European Conference, Jyväskylä, Finland, June 9-13, 1997, Proceedings*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997.
10. D. Lübke, O. Zimmermann, C. Pautasso, U. Zdun, and M. Stocker. Interface evolution patterns: balancing compatibility and extensibility across service life cycles. In *EuroPLoP*, pages 15:1–15:24. ACM, 2019.
11. F. Montesi, C. Guidi, and G. Zavattaro. Service-oriented programming with Jolie. In *Web Services Foundations*, pages 81–107. Springer, 2014.
12. F. Montesi, M. Peressotti, V. Picotti, and O. Zimmermann. EMI framework benchmark, 2024. https://github.com/xpicox/emi-benchmark/tree/main.
13. F. Montesi and J. Weber. From the decorator pattern to circuit breakers in microservices. In *SAC*, pages 1733–1735. ACM, 2018.
14. M. Nygard. Release it!: design and deploy production-ready software. 2007.
15. OASIS. Web services business process execution language version 2.0. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html, 2007.
16. A. Oram. *Ballerina: A Language for Network-Distributed Applications*. O'Reilly, 2019.
17. M. D. Preda, M. Gabbrielli, C. Guidi, J. Mauro, and F. Montesi. Service integration via target-transparent mediation. In *2012 Fifth IEEE International Conference on Service-Oriented Computing and Applications (SOCA), Taipei, Taiwan, December 17-19, 2012*, pages 1–5. IEEE Computer Society, 2012.
18. M. Stocker and O. Zimmermann. From code refactoring to API refactoring: Agile service design and evolution. In J. Barzen, editor, *Service-Oriented Computing*, pages 174–193. Springer, 2021.
19. M. Stocker and O. Zimmermann. API refactoring to patterns: Catalog, template and tools for remote interface evolution. In *EuroPLoP*, pages 2:1–2:32. ACM, 2023.
20. M. Stocker, O. Zimmermann, U. Zdun, D. Lübke, and C. Pautasso. Interface quality patterns: Communicating and improving the quality of microservices apis. In *EuroPLoP*, pages 10:1–10:16. ACM, 2018.
21. O. Zimmermann, D. Lübke, U. Zdun, C. Pautasso, and M. Stocker. Interface responsibility patterns: Processing resources and operation responsibilities. In *EuroPLoP*, pages 9:1–9:24. ACM, 2020.
22. O. Zimmermann, C. Pautasso, D. Lübke, U. Zdun, and M. Stocker. Data-oriented interface responsibility patterns: Types of information holder resources. In *EuroPLoP*, pages 11:1–11:25. ACM, 2020.
23. O. Zimmermann, M. Stocker, D. Lübke, C. Pautasso, and U. Zdun. Introduction to Microservice API Patterns (MAP). In *Microservices*, volume 78 of *OASIcs*, pages 4:1–4:17, 2019.
24. O. Zimmermann, M. Stocker, D. Lübke, and U. Zdun. Interface representation patterns: Crafting and consuming message-based remote apis. In *EuroPLoP*, pages 27:1–27:36. ACM, 2017.
25. O. Zimmermann, M. Stocker, D. Lübke, U. Zdun, and C. Pautasso. *Patterns for API Design: Simplifying Integration with Loosely Coupled Message Exchanges*. Addison-Wesley, 2022.