The following outlines how version 1 was incrementally enhanced to create an updated metamodel which can help to represent recovered architectural models of microservice systems more accurately. The results of these increments present new requirements that need to be fulfilled as enhancements to the MiSAR PIM metamodel (version 1) as shown in Figure 1. These enhancements led to the final version of the MiSAR metamodel.

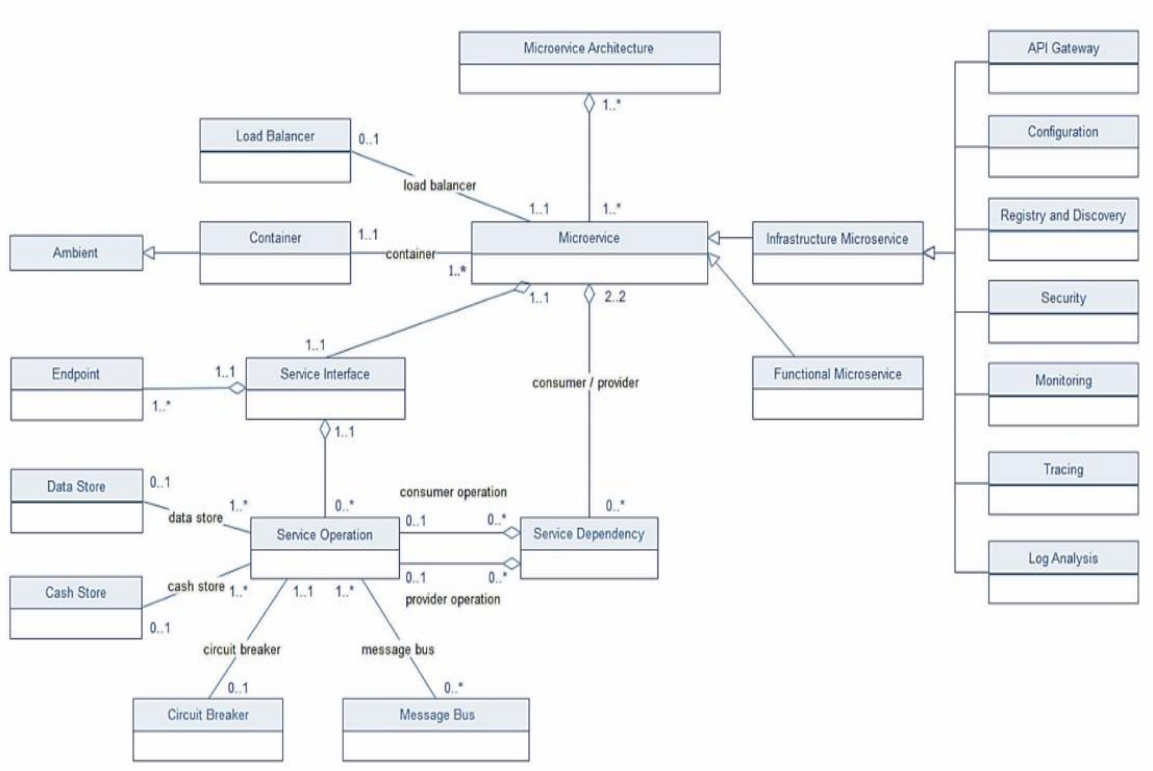

Figure 1: PIM metamodel (version 1)

Increment 1: Supporting Components of Microservice Patterns

Context-1: During the process of manual recovery, the association from the Service Operation concept towards the Data Store, Cash Store, Circuit Breaker and Asynchronous Message Bus concepts in the PIM metamodel (version 1) could not be recovered for many Infrastructure Microservices and these destination concepts were recovered disconnected (such as the Circuit Breaker instance in Figure 2). This is because the Service Operations of Infrastructure Microservices are not explicitly

implemented in the source artefacts.

For illustration purposes, I will show how I manually recovered the instance diagram of the edge-service microservice in case study 7 (mentioned in Table 1). By applying the mapping rules (see Table 2), I could manually recover that edge-service (m5) is an instance of an API Gateway concept (applying rules 39-41 in Table 2) as shown in Figure 2, which is a subtype of Infrastructure Microservice according to metamodel (version 1). Also, a Circuit Breaker element is recovered (applying rule 52 in Table 2). However, it is noticeable that the Service Operation concept was not recovered at all from the edge-service Java source artefacts, and that Circuit Breaker object (cb1) in Figure 2 is recovered disconnected with no associations at all. The reason is that infrastructure providers, such as edge-service microservice, when implemented with Spring Boot/Cloud framework, do not need to explicitly implement any Service Operations in their source artefacts. This discussion leads to the following requirement:

Requirement-1.1➔ Infrastructure components such as Circuit Breaker, Data Store, Cash Store and asynchronous Message Bus concepts need to be directly associated with Microservices in order not to result in disconnected instances.

Enhancement-1.1➔ Reposition the association of the Data Store, Cash Store, Circuit Breaker and asynchronous Message Bus concepts from Service Operation to Microservice instead.

Table 1: Selected systems

ID	Project Name	Project Repository URL	Microservice Count	No. of Developers	Project Timeline	Documentation	Architecture Diagram
1	spring-netflix-oss-microservices	https://github.com/fernandoabcampos/spring-netflix-oss-microservices	9	1	Mar 13, 2016 – Mar 20, 2020	Not Available	Not Available
2	spring-rabbitmq-messaging-microservices	https://github.com/jonashackt/spring-rabbitmq-messaging-microservices	7	2	Nov 4, 2018 – Mar 2 , 2020	Available	Available
3	cloud-enabled-microservice	https://github.com/sergeikh/cloud-enabled-microservice	7	1	Mar 5, 2017 – Mar 2, 2020	Available	Not available
4	event-sourcing-microservices-example	https://github.com/kbastani/event-sourcing-microservices-example	10	5	Oct 22, 2017 – Mar 2, 2020	Available	Available
5	spring-cloud-sidecar-polygot	https://github.com/BarathArivazhagan/spring-cloud-sidecar-polygot	7	3	Aug 20, 2017 – Mar 2, 2020	Available	Available
6	microservices-basics-spring-boot	https://github.com/anilallewar/microservices-basics-spring-boot	10	3	Apr 23, 2017 – Mar 2, 2020	Available	Available
7	spring-cloud-event-sourcing-example	https://github.com/kbastani/spring-cloud-event-sourcing-example	15	4	Mar 20, 2016 – Mar 2, 2020	Available	Available
8	spring-boot-graph-processing-example	https://github.com/kbastani/spring-boot-graph-processing-example	9	3	Dec 13, 2015 – Mar 2, 2020	Available	Available
9	BookStoreApp-Distributed-Application	https://github.com/devdcores/BookStoreApp-Distributed-Application	14	1	May 12, 2019 – Mar 2, 2020	Available	Available

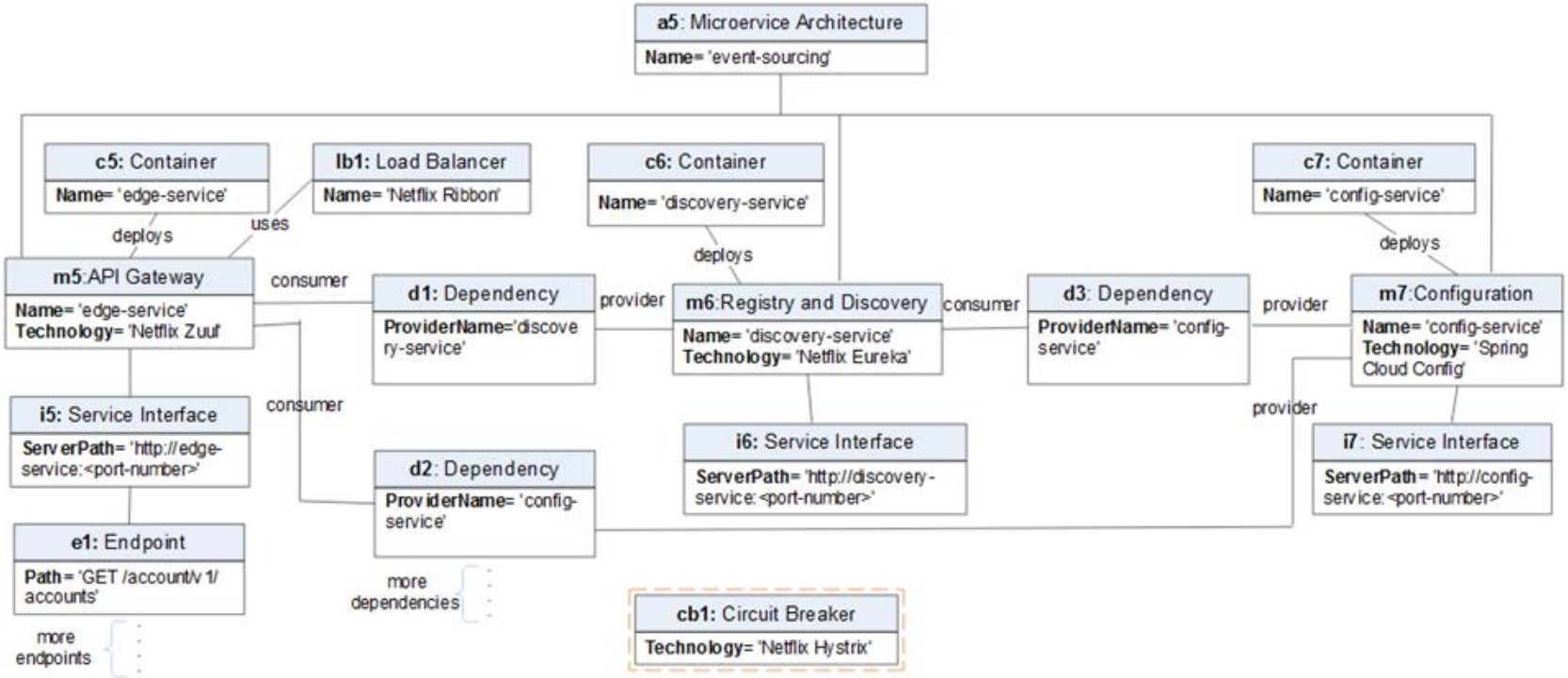

Figure 2: PIM instance recovered for edge-service from case study 7 using PIM metamodel (Version 1).

Table 2: Mapping rules applied in edge-service in case study 7

SEQ	PIM Concept (Source)	PIM Concept (Destination)	Mapping Rule	edge-service
1	Microservice Architecture	-	The name of Microservice Architecture concept is indicated by the name of the root GitHub Repository which contains all artifacts of the application's project.	1
2	Microservice Architecture	-	The name of Microservice Architecture concept is indicated by the value of <project><artifactId> key in the Build File of the application's project.	1
3	Microservice Architecture	-	The name of Microservice Architecture concept is indicated by the value of <project><parent><artifactId> key in the Build File of the microservice's project.	1
4	Microservice Architecture	Microservice	The name of Microservice concept is indicated by the value of <project><modules><module> key in the Build File of the application's project.	1
5	Microservice Architecture	Microservice	The name of Microservice concept is indicated by the value of <project><artifactId> key in the Build File of the microservice's project.	1
6	Microservice Architecture	Microservice	The name of Microservice concept is indicated by the value of the property 'spring.application.name:' in the Configurations File of the microservice's project.	1
7	Microservice Architecture	Microservice	The name of Microservice concept is indicated by last section of package name of a Java Class with '@SpringBootApplication' annotation in the Source Code File of the microservice's project.	1
8	Microservice Architecture	Microservice	The name of Microservice concept is indicated by last section of package name of a Java Class with Java Method with identifier of 'main' that invockes another Java Method with identifier of 'SpringApplication.run' in the Source Code File of the microservice's project.	1
9	Microservice Architecture	Microservice	The name of Microservice concept is indicated by the JAR application name argument of 'ADD' command in the Container Build File of the microservice's project.	1
10	Microservice Architecture	Microservice	The name of Microservice concept is indicated by the key name of service container definition in the Container Orchestration File of the application's project.	1
11	Microservice	Container	The name of Container concept is indicated by the value of <project><modules><module> key in the Build File of the application's project.	1
12	Microservice	Container	The name of Container concept is indicated by the value of <project><artifactId> key in the Build File of the microservice's project.	1
13	Microservice	Container	The name of Container conceptis indicated by the value of the property 'spring.application.name:' in the Configurations File of the microservice's project.	1
14	Microservice	Container	The name of Container conceptis indicated by last section of package name of a Java Class with '@SpringBootApplication' annotation in the Source Code File of the microservice's project.	1
15	Microservice	Container	The name of Container concept is indicated by last section of package name of a Java Class with Java Method with identifier of 'main' that invockes another Java Method with identifier of 'SpringApplication.run' in the Source Code File of the microservice's project.	1

16	Microservice	Container	The name of Container concept is indicated by the JAR application name argument of 'ADD' command in the Container Build File of the microservice's project.	1
17	Microservice	Container	The name of Container concept is indicated by the key name of service container definition in the Container Orchestration File of the application's project.	1
18	Microservice	Load Balancer	A 'Netflix Ribbon' Load Balancer to a Microservice is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-eureka' in the Build File of the microservice's project.	1
19	Microservice	Load Balancer	A 'Netflix Ribbon' Load Balancer to a Microservice is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-zuul' in the Build File of the microservice's project.	1
20	Microservice	Load Balancer	A 'Netflix Ribbon' Load Balancer to a Microservice is indicated by the property 'eureka.client.serviceUrl.defaultZone:' in the Configurations File of the microservice's project.	1
21	Microservice	Load Balancer	A 'Netflix Ribbon' Load Balancer to a Microservice is indicated by the property 'ribbon.ReadTimeout:' or 'ribbon.ConnectTimeout:' with nonzero values in the Configurations File of the microservice's project.	1
22	Microservice	Load Balancer	A 'Netflix Ribbon' Load Balancer to a Microservice is indicated by a Java Class with '@EnableZuulProxy' annotation in the Source Code File of the microservice's project.	1
23	Microservice	Load Balancer	A 'Netflix Ribbon' Load Balancer to a Microservice is indicated by a Java Class with '@EnableEurekaClient' annotation in the Source Code File of the microservice's project.	1
24	Microservice	Service Interface	The server path of Service Interface concept is indicated by the value of <project><modules><module> key in the Build File of the application's project.	1
25	Microservice	Service Interface	The server path of Service Interface concept is indicated by the value of <project><artifactId> key in the Build File of the microservice's project.	1
26	Microservice	Service Interface	The server path of Service Interface concept is indicated by the value of the property 'spring.application.name:' in the Configurations File of the microservice's project.	1
27	Microservice	Service Interface	The server path of Service Interface concept is indicated by last section of package name of a Java Class with '@SpringBootApplication' annotation in the Source Code File of the microservice's project.	1
28	Microservice	Service Interface	The server path of Service Interface concept is indicated by last section of package name of a Java Class with Java Method with identifier of 'main' that invockes another Java Method with identifier of 'SpringApplication.run' in the Source Code File of the microservice's project.	1
29	Microservice	Service Interface	The server path of Service Interface concept is indicated by the JAR application name argument of 'ADD' command in the Container Build File of the microservice's project.	1

30	Microservice	Service Interface	The server path of Service Interface concept is indicated by the key name of service container definition in the Container Orchestration File of the application's project.	1
31	Microservice	Service Dependency	A 'Spring Cloud Config' Configuration provider to a Microservice is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-config' in the Build File of the microservice's project.	1
32	Microservice	Service Dependency	A 'Netflix Eureka' Registry and Discovery provider to a Microservice is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-eureka' in the Build File of the microservice's project.	1
33	Microservice	Service Dependency	A 'Spring Cloud Config' Configuration provider to a Microservice is indicated by the hostname section of the url value of the property 'spring.cloud.config.uri:' or 'spring.cloud.config.failFast: true' in the Configurations File of the microservice's project.	1
34	Microservice	Service Dependency	A 'Netflix Eureka' Registry and Discovery provider to a Microservice is indicated by the hostname section of the url value of the property 'eureka.client.serviceUrl.defaultZone:' in the Configurations File of the microservice's project.	1
35	Microservice	Service Dependency	A 'Spring Cloud OAuth2' Security provider to a Microservice is indicated by the hostname section of the url value of the property 'security.oauth2.resource.userInfoUri:' or 'security.oauth2.client.accessTokenUri:' in the Configurations File of the microservice's project.	1
36	Microservice	Service Dependency	A 'Spring Cloud OAuth2' Security provider to a Microservice is indicated by a Java Class with '@EnableResourceServer' or 'EnableOAuth2Client' annotation in the Source Code File of the microservice's project.	1
37	Microservice	Service Dependency	A Registry and Discovery provider to a Microservice is indicated by a Java Class with '@EnableEurekaClient' annotation in the Source Code File of the microservice's project.	1
38	Microservice	Service Dependency	A Microservice provider to a Microservice is indicated by the service container name of 'depends_on' or 'links' key in the Container Orchestration File of the application's project.	3
39	API Gateway	-	An API Gateway concept with technology of 'Netflix Zuul' is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-zuul' in the Build File of the microservice's project.	1
40	API Gateway	-	An API Gateway concept with technology of 'Netflix Zuul' is indicated by the property name that starts with 'zuul.routes.' in the Configurations File of the microservice's project.	1
41	API Gateway	-	A 'Netflix Zuul' API Gateway is indicated by a Java Class with '@EnableZuulProxy' annotation in the Source Code File of the microservice's project.	1
42	API Gateway	Service Dependency	A Microservice provider to a 'Netflix Zuul' API Gateway is indicated by the property that starts with 'zuul.routes.' and ends with the microservice name in the Configurations File of the microservice's project.	7

43	Registry and Discovery	Service Dependency	A Microservice provider to a 'Netflix Eureka' Registry and Discovery is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-eureka' in the Build File of the microservice's project.	1
44	Registry and Discovery	Service Dependency	A Microservice provider to a 'Netflix Eureka' Registry and Discovery is indicated by the property 'eureka.client.serviceUrl.defaultZone:' in the Configurations File of the microservice's project.	1
45	Registry and Discovery	Service Dependency	A Microservice provider to a 'Netflix Eureka' Registry and Discovery is indicated by a Java Class with '@EnableEurekaClient' annotation in the Source Code File of the microservice's project.	1
46	Monitoring	Service Dependency	A Microservice provider to a 'Netflix Hystrix Dashboard' or 'Netflix Turbine' Monitoring is is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-hystrix' in the Build File of the microservice's project.	1
47	Monitoring	Service Dependency	A Microservice provider to a 'Netflix Hystrix Dashboard' or 'Netflix Turbine' Monitoring is indicated by a 'compile' Gradle command with an argument 'org.springframework.cloud:spring-cloud-starter-hystrix' in the Build File of the microservice's project.	1
48	Monitoring	Service Dependency	A Microservice provider to a 'Netflix Hystrix Dashboard' or 'Netflix Turbine' Monitoring is indicated by the non-zero property 'hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds:' or the property 'feign.hystrix.enabled: true' in the Configurations File of the microservice's project.	1
49	Monitoring	Service Dependency	A Microservice provider to a 'Netflix Hystrix Dashboard' or 'Netflix Turbine' Monitoring is indicated by a Java Class with '@EnableHystrix' annotation in the Source Code File of the microservice's project.	1
50	Log Analysis	Service Dependency	A Microservice provider to a Log Analysis is indicated by the property name that starts with 'logging.level.' in the Configurations File of the microservice's project.	1
51	Service Interface	Endpoint	An Endpoint to Service Interface is indicated by the value of the property that starts with 'zuul.routes.' and ends with the microservice name in the Configurations File of the microservice's project.	$7 * N$[1]
52	Service Operation	Circuit Breaker	A 'Netflix Hystrix' Circuit Breaker to Service Operation is indicated by the non-zero property 'hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds:' or the property 'feign.hystrix.enabled: true' in the Configurations File of the microservice's project.	1

[1] Number of endpoints in each provider microservice.

Context-2: In the initial design of the PIM metamodel (version 1), each infrastructure provider, e.g. API Gateway, Configuration, Discovery and Registry, Security, Log Analysis, Monitoring, and Tracing, is represented as an independent Infrastructure Microservice. For example, it was assumed that an instance of the Discovery and Registry microservice provides only the service registry and discovery functionality. However, this representation encountered certain limitations. The current PIM metamodel (version 1) allows for only one subtype of Infrastructure Microservice at a time. With regard to this issue, the bookstore-consul-discovery microservice from case study 9 is manually recovered as either an instance of Discovery and Registry concept or Configuration concept, subtypes of the Infrastructure Microservice (applying rules 6 and 7 in Table 3). However, being an image of Consul Agent provided by HashiCorp[2] the bookstore-consul-discovery microservice provides multiple infrastructure patterns all at once and out-of-the-box, including Configuration and Registry and Discovery.

Requirement-1.2➔ One Infrastructure Microservice can support multiple-infrastructure patterns.

Enhancement-1.2➔ A new Infrastructure Pattern Component concept is introduced. A microservice can aggregate zero to many Infrastructure Pattern Components. A pattern component refers to an architectural element that supports the functionality of a pattern. Infrastructure pattern components have more specific categories. All subtypes of Infrastructure Microservice types in metamodel version 1 become instances of a new enumeration type named Infrastructure Pattern Category, defining the category of one Infrastructure Pattern Component instance.

Table 3:Mapping rules applied in bookstore-consul-discovery in case study 9

SEQ	PIM Concept (Source)	PIM Concept (Destination)	Mapping Rule	bookstore-consul-discovery
1	Microservice Architecture	-	The name of Microservice Architecture concept is indicated by the name of the root GitHub Repository which contains all artifacts of the application's project.	1
2	Microservice Architecture	Microservice	The name of Microservice concept is indicated by the key name of service container definition in the Container Orchestration File of the application's project.	1

[2] https://cloud.spring.io/spring-cloud-static/spring-cloud-consul/2.2.0.M1/.

3	Microservice	Container	The name of Container concept is indicated by the key name of service container definition in the Container Orchestration File of the application's project.	1
4	Microservice	Service Interface	The server path of Service Interface concept is indicated by the key name of service container definition in the Container Orchestration File of the application's project.	1
5	Infrastructure Microservice	-	An Infrastructure Microservice concept is indicated by a service container definition that does not have 'build' key in the Container Orchestration File of the application's project.	1
6	Configuration	-	A 'Consul' Configuration concept is indicated by an 'image:' key with value that starts wth 'consul:' in the Container Orchestration File of the application's project.	1
7	Registry and Discovery	-	A 'Consul' Registry and Discovery concept is indicated by an 'image:' key with value that starts wth 'consul:' in the Container Orchestration File of the application's project.	1

Context-3: In the PIM metamodel version 1, it is not straightforward to determine the type of infrastructure pattern requested by a consumer microservice involved in a Service Dependency with an infrastructure provider. Hence, this design requires some improvements. To illustrate, let us look at the edge-service microservice from case study 7, as depicted in Figure 2. By applying the mapping rules 32, 34, 37 and 38 in Table 2, edge-service (m5) is an instance of the APIGateway microservice, which has a Service Dependency (d1) associating it to a Registry and Discovery instance named discovery-service (m6). The (m6) acts as a Registry and Discovery provider to edge-service (m5), since the edge-service uses this pattern to register its address. The edge-service, by applying rules 31, 33 and 38 in Table 2, has another Service Dependency instance (d2) associated to config-server (see Figure 2). The (m7) is considered a Configuration provider since edge-service (m5) needs this pattern to pull its centralized configuration properties. The PIM (version 1) does not express literally that edge-service microservice is using both infrastructure patterns: Registry and Discovery and Configuration. Enhanced PIM instances are provided in Figure 3.

Requirement-1.3➔ The metamodel design needs to clarify information about infrastructure pattern components that microservices use. Infrastructure pattern components should be divided into two types: ones that provide services to microservices and others that call/request services from microservices.

Enhancement-1.3➔ Infrastructure pattern component is extended by two subtypes: Infrastructure Pattern Server Component and Infrastructure Pattern Client Component. The first represents infrastructure patterns provided by a microservice, i.e. subtypes of infrastructure microservice, while the second represents infrastructure patterns that are used/requested by a microservice, i.e. consumers of remote infrastructure microservices.

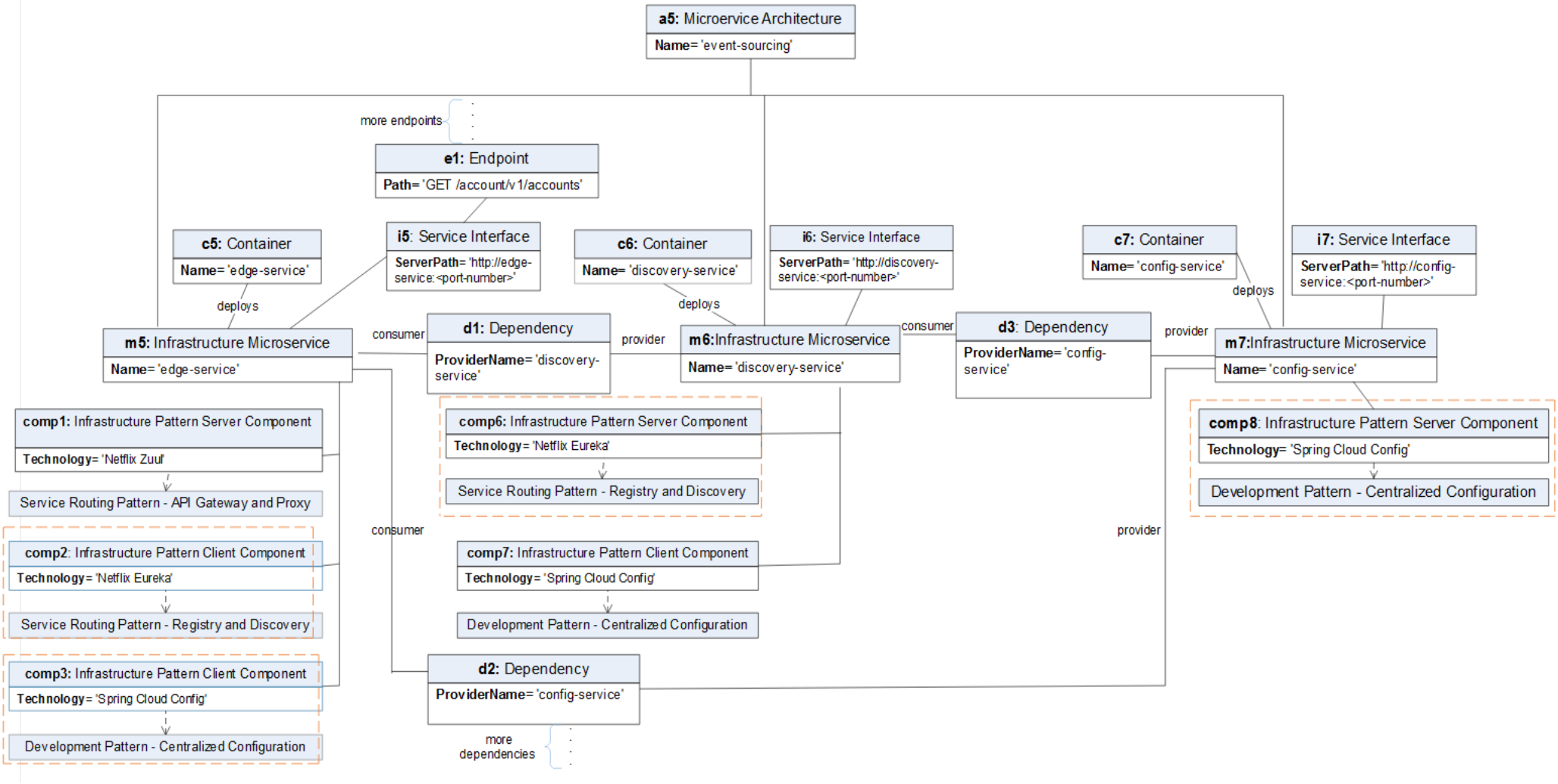

Figure 3: Enhanced PIM instance recovered for edge-service, discovery-service and config-service microservice from case study 7.

Context-4: In PIM (version 1), we previously considered representing the Data Store, Data Cache, Asynchronous Message Bus and Load Balancer as backend infrastructure patterns within the consumer microservice itself that can only be used/requested by client microservices, as explained in context-1. Therefore, the providers of these backend patterns were represented as mere infrastructure microservices, because the subtypes of Infrastructure Microservice did not yet include the Data Store, Data Cache, Asynchronous Message Bus or Load Balancer. To illustrate, the kafka microservice from case study 4 is a provider of asynchronous Message Bus as illustrated in Figure 4, however, according to the initial mapping rule 6 in Table 4, it is represented as mere Infrastructure Microservice. Similarly, the redis microservice from case study 7 is a Cache Store provider, and the mysql, neo4j and mongo microservices from case study 7 are Data Stores, however, according to the initial mapping rules, they are represented as mere Infrastructure Microservices.

Requirement-1.4➔ Data Store, Cash Store, Load Balancer and Asynchronous Message Bus need to be represented as Infrastructure Pattern Components provided and requested by microservices.

Enhancement-1.4➔ Data Store, Cash Store, Load Balancer and Asynchronous Message Bus are appended to an enumeration type Infrastructure Pattern Category so that they can be represented as Infrastructure Pattern Components.

Table 4: Mapping rules applied in Kafka in case study 4

SEQ	PIM Concept (Source)	PIM Concept (Destination)	Mapping Rule	kafka
1	Microservice Architecture	-	The name of Microservice Architecture concept is indicated by the name of the root GitHub Repository which contains all artifacts of the application's project.	1
2	Microservice Architecture	Microservice	The name of Microservice concept is indicated by the key name of service container definition in the Container Orchestration File of the application's project.	1
3	Microservice	Container	The name of Container concept is indicated by the key name of service container definition in the Container Orchestration File of the application's project.	1
4	Microservice	Service Interface	The server path of Service Interface concept is indicated by the key name of service container definition in the Container Orchestration File of the application's project.	1
5	Infrastructure Microservice	-	An Infrastructure Microservice concept is indicated by a service container definition that does not have 'build' key in the Container Orchestration File of the application's project.	1
6	Infrastructure Microservice	-	A 'Kafka' Infrastructure Microservice concept is indicated by an 'image:' key with value that contains 'spotify/kafka' in the Container Orchestration File of the application's project.	1

The application of the first enhancement increment to the PIM metamodel (version 2), in response to all four requirements aforementioned, is shown in Figure 5.

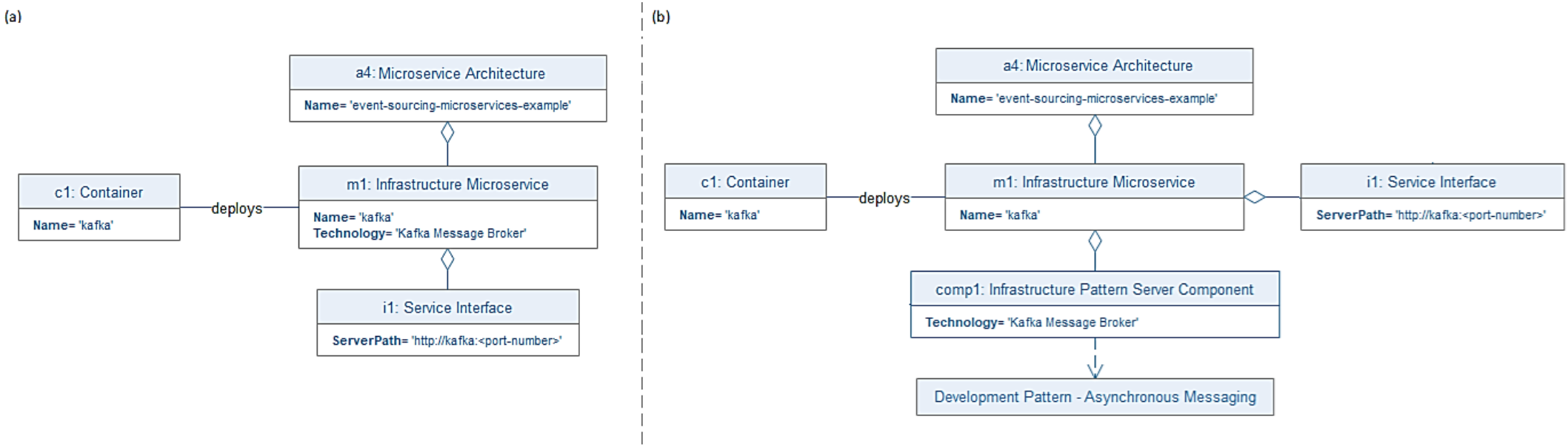

Figure 4: (a) PIM instances recovered for Kafka microservice from case study 4 based on PIM metamodel (Version 1). (b) Enhanced PIM instances recovered for *kafka* microservice based on enhanced PIM metamodel (Version 2).

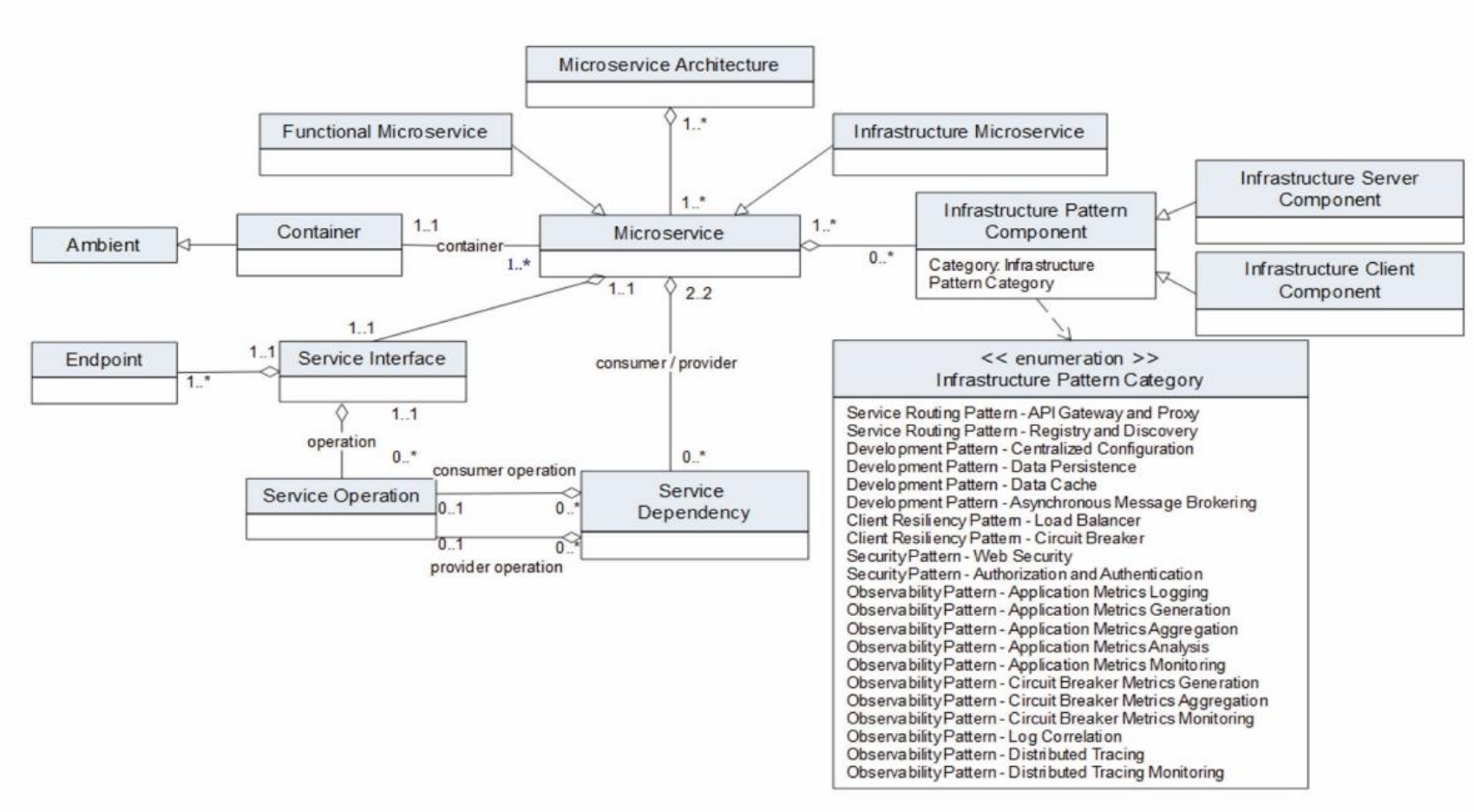

Figure 5: PIM metamodel (version 2).

Increment 2: Supporting Synchronous Communication through Endpoints

Context-1: Request-response synchronous inter-service communication is represented in the PIM metamodel (version 2, see Figure 5) using the Service Dependency concept, which specifies the Service Operation of the consumer that invokes a remote Service Operation of the provider. However, the request-response inter-service communication in the PIM metamodel (version 2) is based on Service Operation, which is low-level and probably unrecoverable in some cases, and Service Operations are not linked to their exposed Endpoints.

To illustrate, it can be observed from case study 1 that the card-statement-composite consumer microservice sends two consecutive requests: the first is a request to card provider microservice through an endpoint GET api/card/{cardId}, and the second is a request to statement provider microservice through an endpoint GET api/statement?cardId={cardId}, as illustrated in the sequence diagram in Figure 6. A PIM instance of the card-statement-composite microservice based on version 2 is shown in Figure 7. It states that Service Dependency instance (d3) defines the consumer's Service Operation, named getStatementByCardId, which invokes a remote provider's Service Operation named getCard. Similarly, Dependency instance (d4) indicates that the same Service Operation of the consumer invokes a remote Service Operation named getStatements of another provider statement. However, the invocation to the provider is made first to the Endpoint, which, in turn, maps the request to the Service Operation. One observation on the PIM instance is that it does not demonstrate the mapping of Service Operations by equivalent Endpoints.

Requirement-2.1➔ Service Operations should be linked to their exposed Endpoints.

Enhancement-2.1➔The association of Service Operation is repositioned from Service Interface to Endpoint. This association is an optional association that goes from Endpoint to Service Operation. It is optional support for modelling Infrastructure Microservices that tend to hide, i.e. abstract, implementation of their Service Operations.

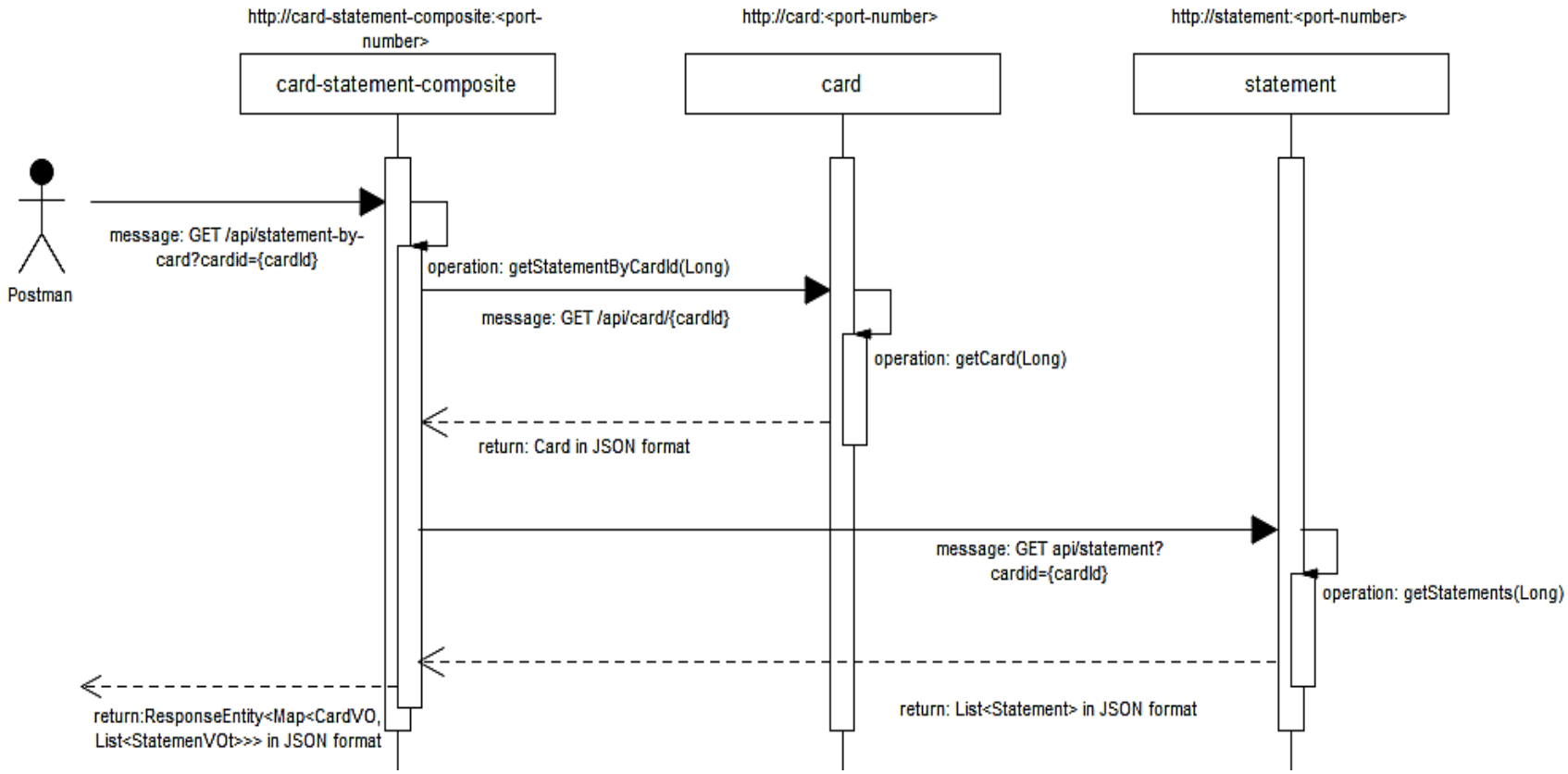

Figure 6: Synchronous request-response between card-statement- composite, card and statement microservices from case study 1.

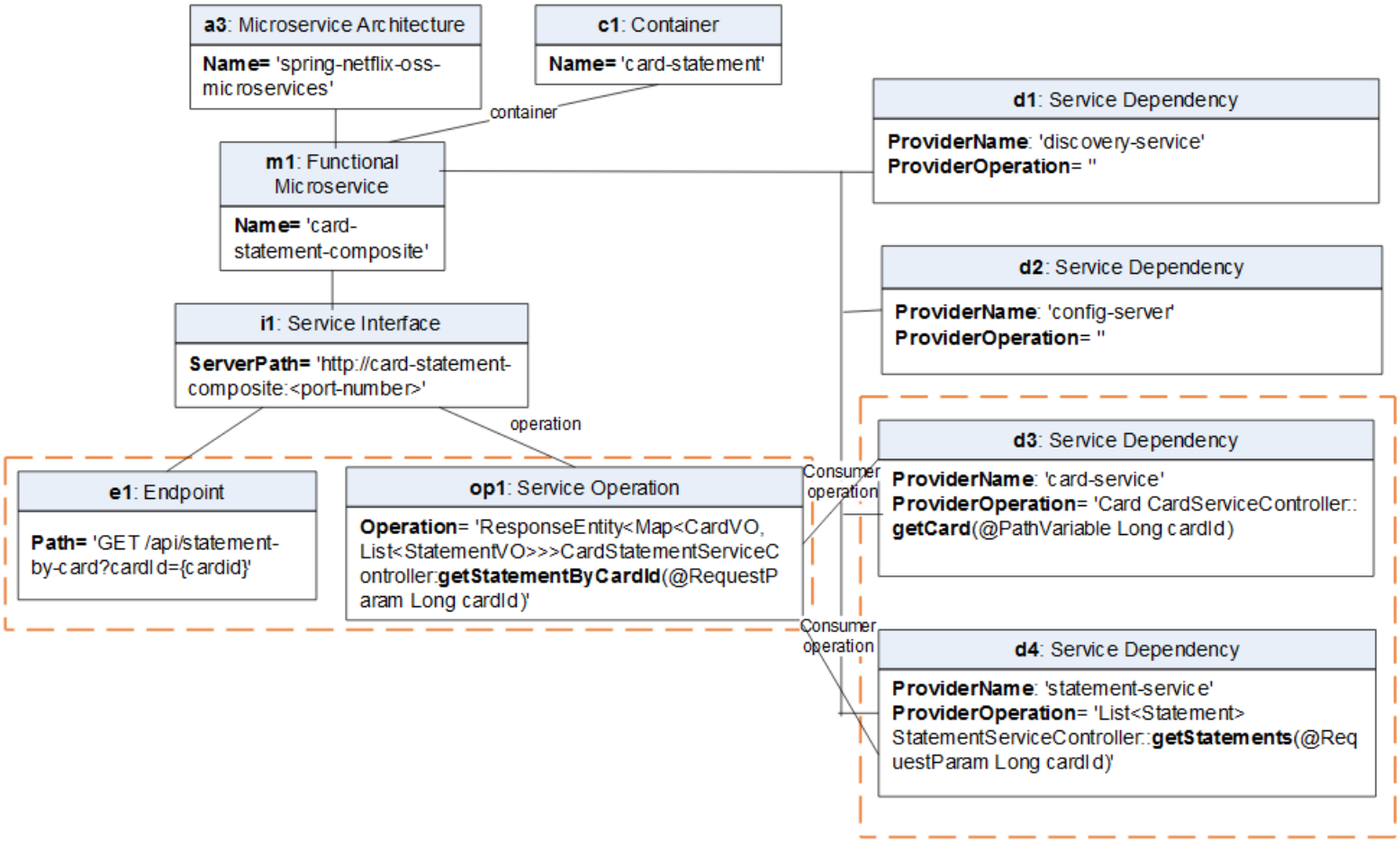

Figure 7: PIM instance recovered for card-statement-composite microservice from case study 1 based on PIM metamodel (version 2).

The second observation is that the representation of the request-response inter-service communication as an invocation to the remote provider's Service Operation is considered low-level, and Service Operations can be unrecoverable, especially in the case of modelling Infrastructure Microservices, as explained earlier in Increment-1.

Requirement-2.2 ➔ Service Dependency between two microservices should mainly involve the consumer microservice and provider microservice. Additional information to consider is the consumer's operation and provider's endpoint if they are available.

Enhancement-2.2 ➔ The two associations between Dependency and Service Operations are replaced with two optional attributes: ConsumerOperation and ProviderEndpoint.

Having shifted the association of the Service Dependency concept to the provider's Endpoint instead of the provider's Service Operation (in Enhancement-2.2), information about the format of the request and response data messages at the provider's endpoint is missing. Service Operation already specifies the format of its request/input message via the data type of the parameter(s), if any, while the response/output message is specified by the data type of the object returned, if any. To illustrate via the Provider Operation value in Service Dependency instance (d3) as in Figure 7, we can see that the data type of the return object is Card, which corresponds to the payload of the response message. The data type is eventually converted to some standard data representation format, e.g. XML or JSON. Thus, it is necessary to provide the schema of the message in a standard format in order for it to be followed by the consumer.

Requirement-2.3➔ An Endpoint of a microservice should define the format and type of its data messages, if any.

Enhancement-2.3 ➔ A Service Message concept is introduced. Service Message is associated with Endpoint and it is defined by Type, i.e. request/response/error, Schema and Schema Format, i.e. XML/JSON.

Figure 8 shows the enhanced PIM (version 3). Figure 9 shows a PIM instance of card-statement-composite microservice based on the enhanced PIM (version 3).

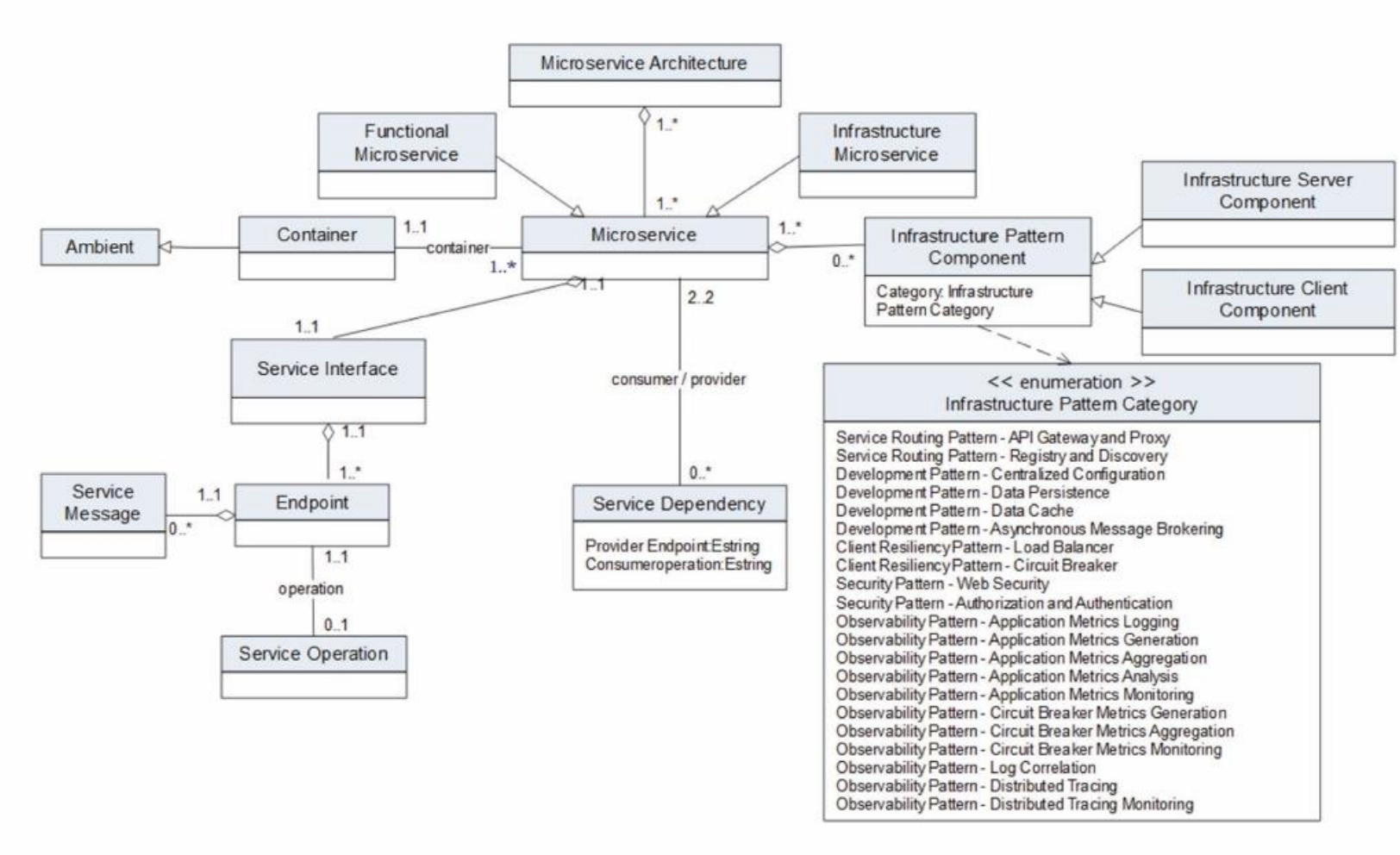

Figure 8: PIM metamodel (version 3).

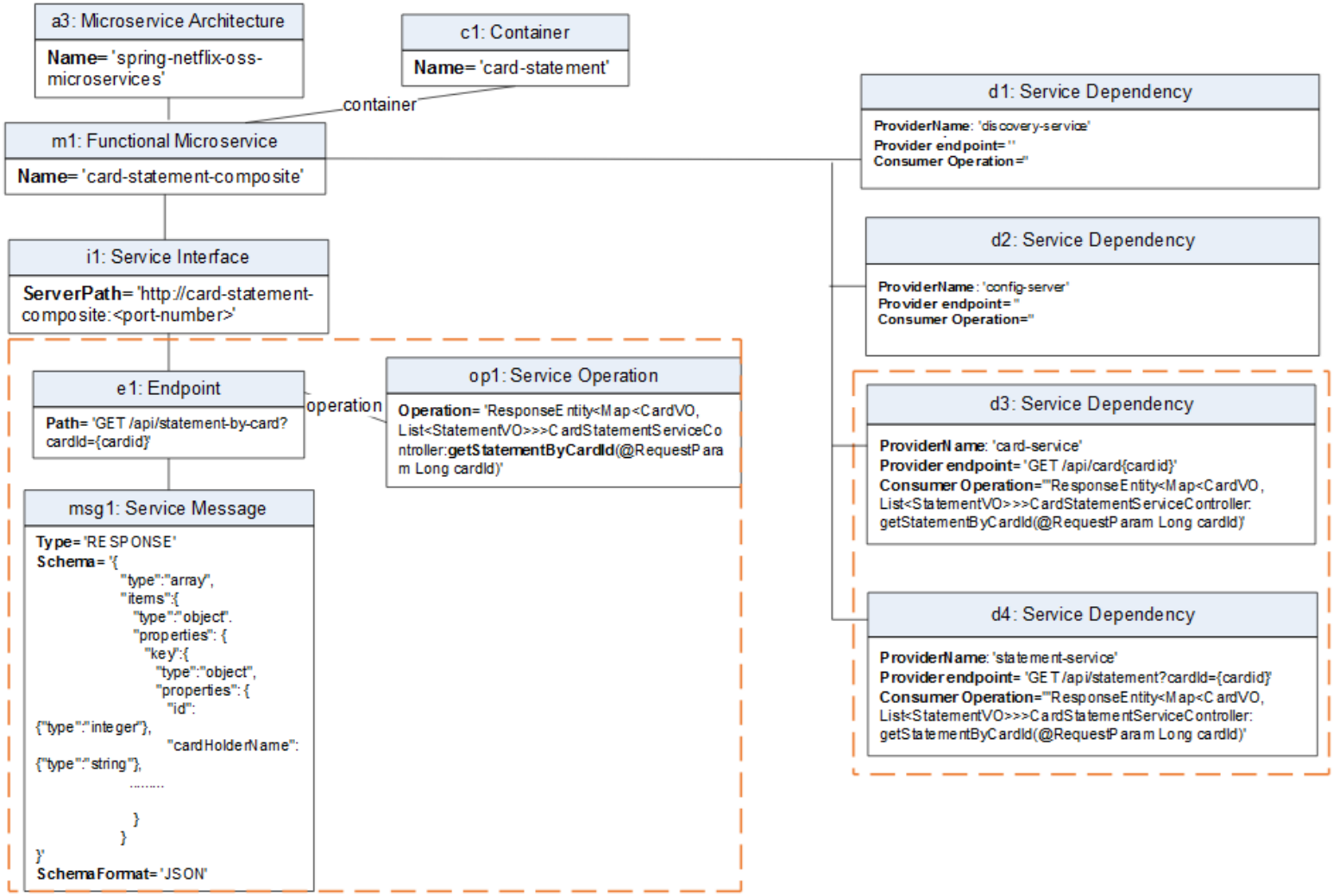

Figure 9: Enhanced PIM instance recovered for card-statement-composite microservice from case study 1 based on enhanced PIM metamodel (version 3).

Increment 3: Supporting asynchronous communication

Context-1: Unlike synchronous request-response, in asynchronous message-driven communication, the consumer does not directly invoke a remote Service Operation nor an Endpoint of the provider; instead, they send an event/message to an intermediary Infrastructure Microservice Asynchronous Messaging, which will eventually forward the event/message to the provider. However, the PIM metamodel (version 3) does not consider message-driven inter-service communication. To illustrate, consider the message-driven inter-service communication implemented in case study 2 using the sequence diagram depicted in Figure 10. This inter-service communication is initiated by an external synchronous request received on the endpoint of the weatherservice microservice: 'POST/weather/forecast'. Then, the forecast operation of the weatherservice microservice will publish an EventGetOutlook message accompanied with a routing key, weatherbackend:queue, to the RabbitMQ infrastructure microservice, i.e. a RabbitMQ message broker.

Inside RabbitMQ, as described in Figure 11, according to the AMQP protocol implemented by RabbitMQ, some kind of exchange will receive the message and then forward it to a particular queue, depending on the routing key provided. In this case, the EventGetOutlook message is received by the default exchange, forwarded to a particular queue named weatherbackend:queue and eventually received by the weatherbackend microservice, the provider, because it is subscribed to that particular queue. After that, the provider's operation handleMessage will process the message received and the response back to the weatherservice microservice by dropping an EventGeneralOutlook message onto RabbitMQ accompanied with a routing key: weatherservice:queue. The message will be forwarded by the exchange to weatherservice:queue and eventually received by the weatherservice microservice. The RabbitMQ infrastructure message bus/broker in case study 2 is configured to create two message exchanges and three message queues (see 11). The three queues are named weathersimple:queue, weatherbackend:queue and weatherservice:queue, and are bound to the message exchanges.

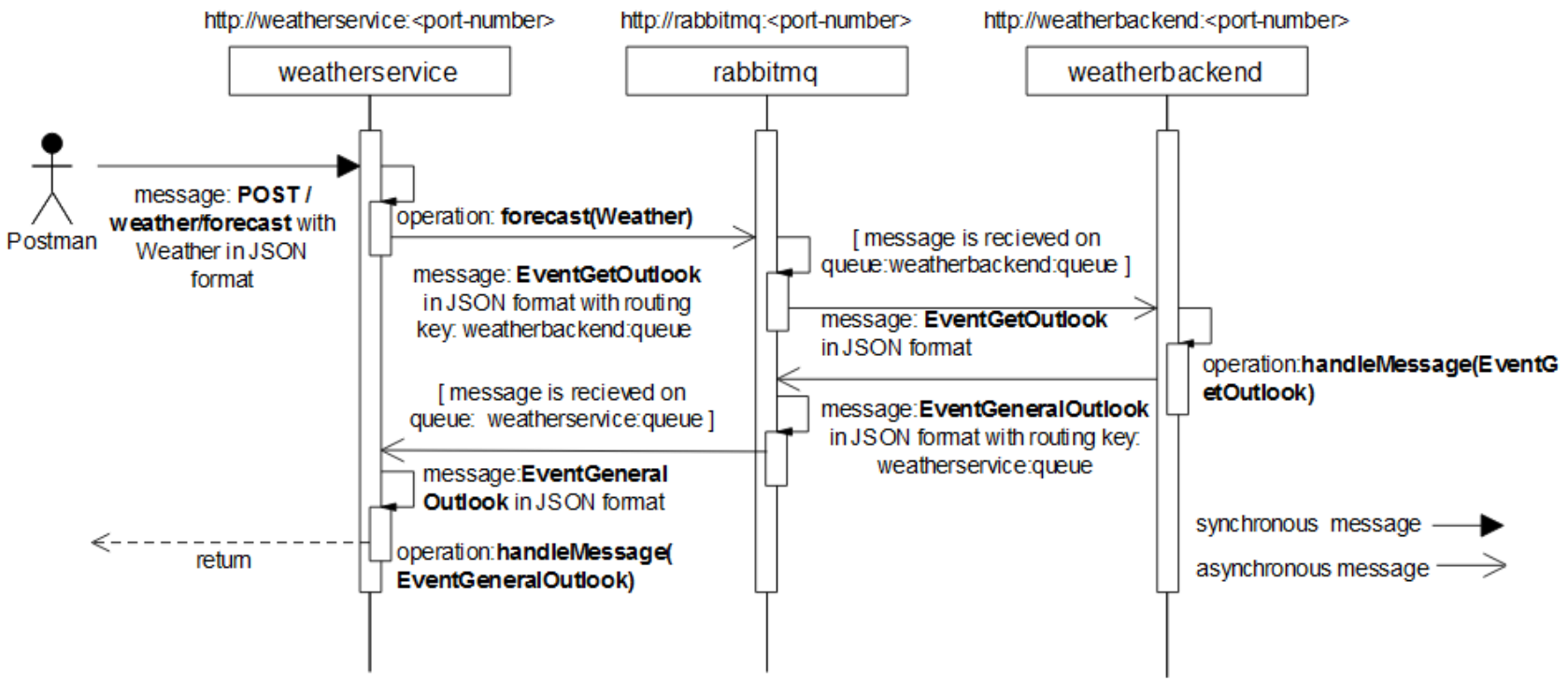

Figure 10:Asynchronous message-driven inter-service communication between weatherservice and weatherbackend microservices (case study 2).

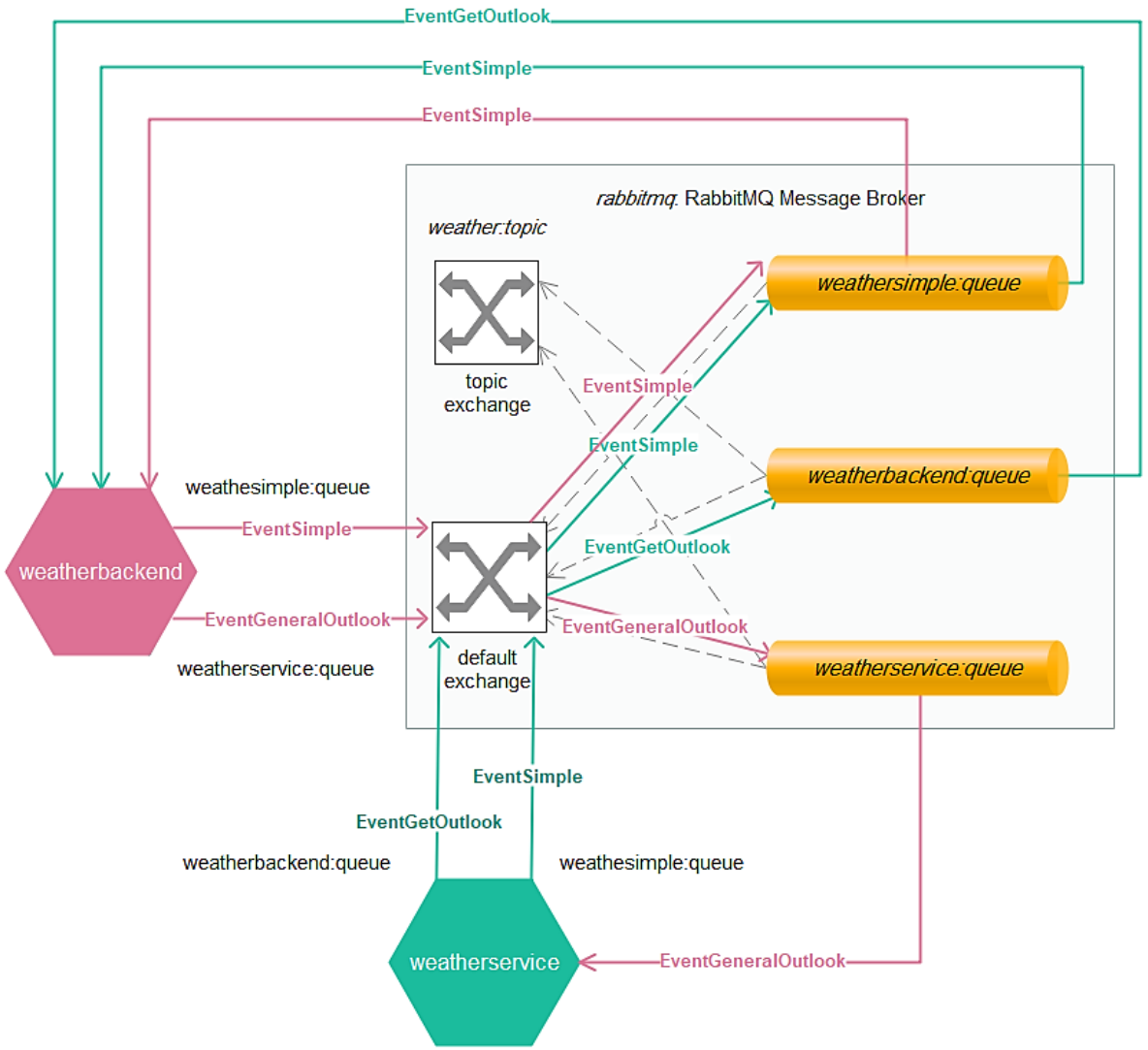

Figure 11: Internal setup of rabbitmq message broker from case study 2.

To illustrate, an outbound queue is where the weatherbackend microservice eventually forwards its events/messages to, however, an inbound queue is where the weatherbackend microservice is subscribed to and receives incoming event/messages. It can be concluded that if the outbound queue name of a consumer microservice matches the inbound queue name of the provider microservice, then the first microservice has a Service Dependency with the second, because it sends an event/message to the second's inbound queue.

This is comparable to request-response inter-service communication, where the microservice that sends a request to the Endpoint of another remote microservice is said to have a Service Dependency with that remote microservice. Therefore, an inbound queue of a microservice can be considered an asynchronous alternative to an Endpoint, and hence it needs to be exposed in a microservice's Service Interface in order to be reachable. According to the previous statement, as seen in Figure 10, the weatherservice microservice is said to have a Service Dependency with the weatherbackend microservice, because it sends an asynchronous *EventGetOutlook* message to *weatherbackend:queue*, i.e. an inbound queue to the weatherbackend microservice. Similarly, the weatherbackend microservice is said to have a Service Dependency with the weatherservice microservice because it sends an asynchronous EventGeneralOutlook message to *weatherservice:queue*, i.e. an inbound queue to the weatherservice microservice. The Enhanced PIM (version 4) is provided in Figure 12. A PIM instance of weatherbackend microservices based on the enhanced PIM (version 4) is provided in Figure 13.

Requirement-3.1 ➔ A message-based asynchronous mechanism of inter-service communication using asynchronous inbound queues and messages should be represented.

Enhancement-3.1 ➔ The concept Queue Listener is introduced, defined by its Name and, as Endpoint, is associated with Service Interface.

Enhancement-3.2 ➔ As Endpoint, Queue Listener is associated with Service Message and maps to Service Operation.

Enhancement-3.3 ➔ Queue Listener and Endpoint are all generalized in a supertype concept called Message Destination, because they all represent the destination at which a remote message is received.

Enhancement-3.4 ➔ An attribute provider's Endpoint is replaced with the provider's destination, which represents both the provider's Endpoint and the provider's Queue.

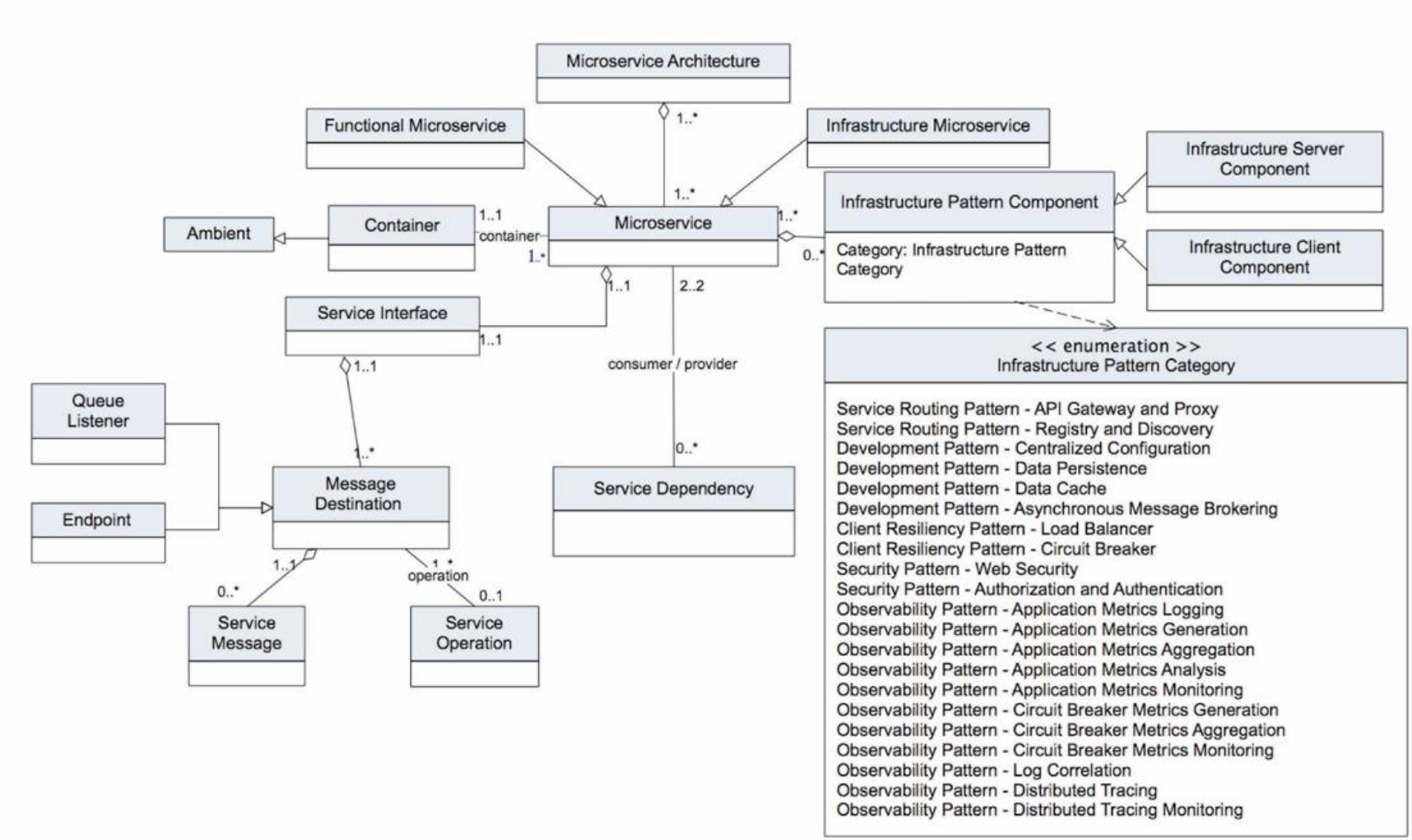

Figure 12: PIM metamodel (version 4).

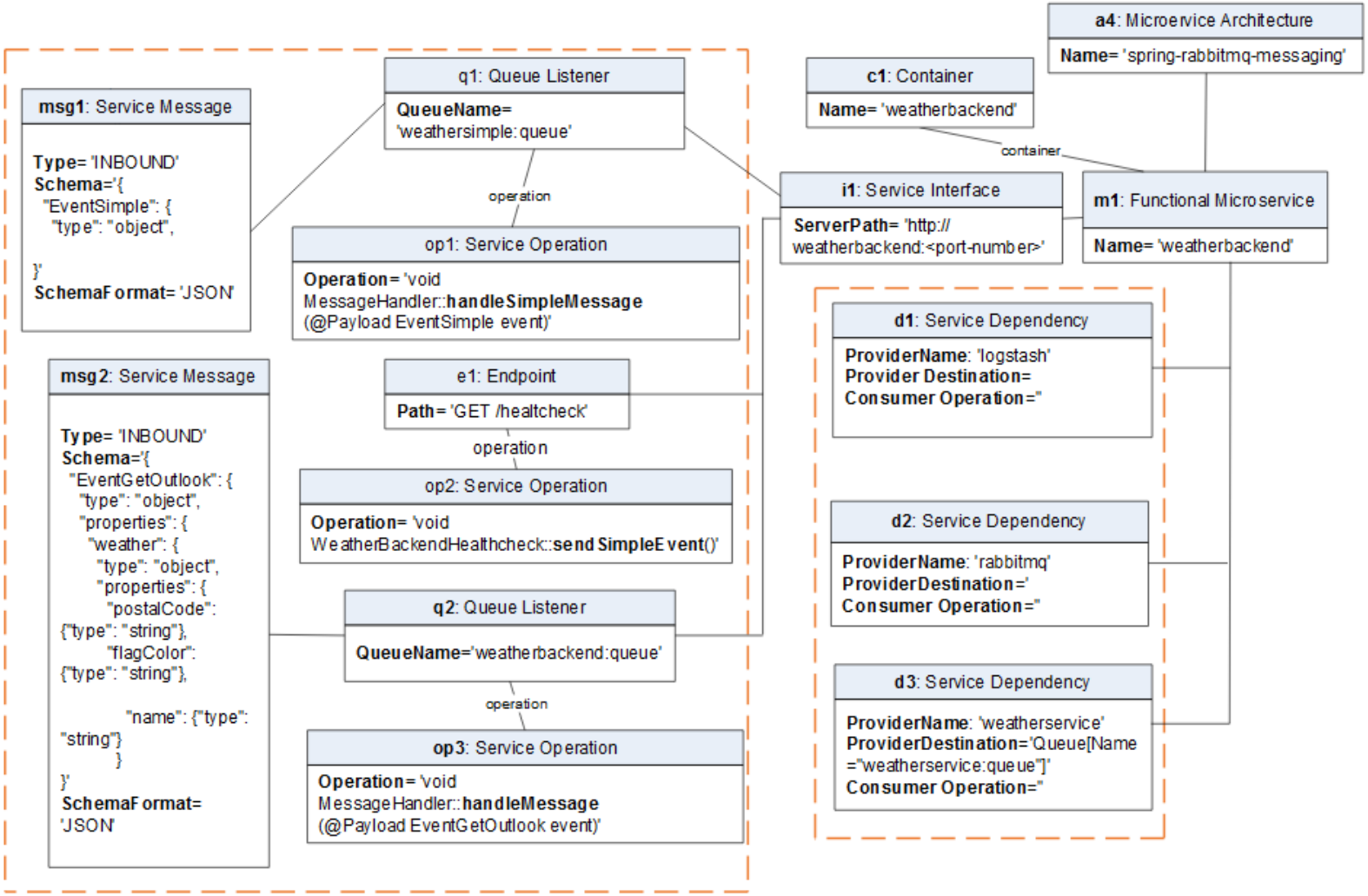

Figure 2: Enhanced PIM instance recovered for weatherbackend microservice from case study 2 based on PIM metamodel (version 4).

Increment 4: Support for Configuration

Context-1: PIM metamodel (version 4) doesn't support multiple configuration profiles implemented for a microservice application, which include default, development, Docker, Kubernetes and test. To illustrate, let us look at the analysis result of the microservice from case study 4, as illustrated in Figure 14. It is noticeable that the model has both embedded and client infrastructure components that are of the same pattern category. For example, components *comp2* and *comp5* both refer to the Kafka message broker; however, the first is a client component for a remote Kafka, while the second is an embedded component. Similarly, *comp3* and *comp4* both refer to the Neo4j graph database; however, the first is a client component for remote Neo4j, while the second is an embedded component. Both *comp6* and *comp7* are client components for service registry and discovery infrastructure patterns, however the first is implemented with Netflix Eureka and the second with Kubernetes. The reason for using this collection of similar components is that the application has multiple configuration profiles; each one is targeted at different running environments. The application uses embedded components in a testing environment only, while the use of service registry and discovery with Kubernetes is intended for an optional situation, where Kubernetes is running and available instead of Netflix Eureka. Apparently, PIM metamodel (version 4) needs enhancement to reflect such multiple configurations for multiple environments. The enhanced and final PIM (version 5) is provided in Figure 15. The PIM instance of *recommendation-service* microservice, based on the enhanced PIM (version 5), is provided in Figure 16.

Requirement-4.1 ➔ Multiple configuration profiles are needed to represent multiple environments.

Enhancement-4.1 ➔ An attribute named Environment is added for the Infrastructure Pattern Component, Message Destination and Dependency concepts.

As a result of this new empirical study, Misar PIM has become able to represent and abstract the technologies and patterns encountered in the systems analyzed as demonstrated in Table 6-5.

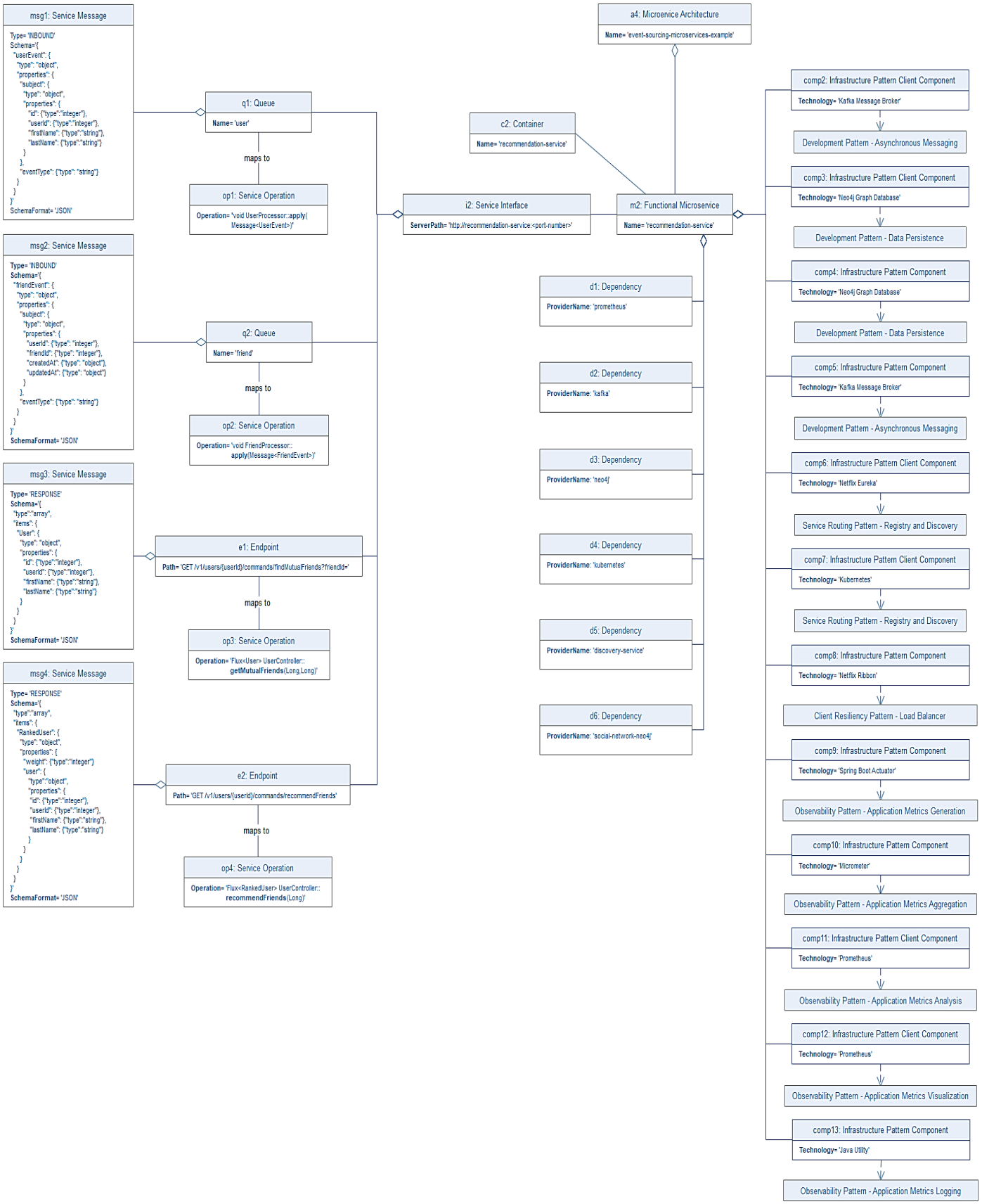

Figure 14: PIM instance recovered for *recommendation-service* microservice from case study 4 based on PIM metamodel (version 4).

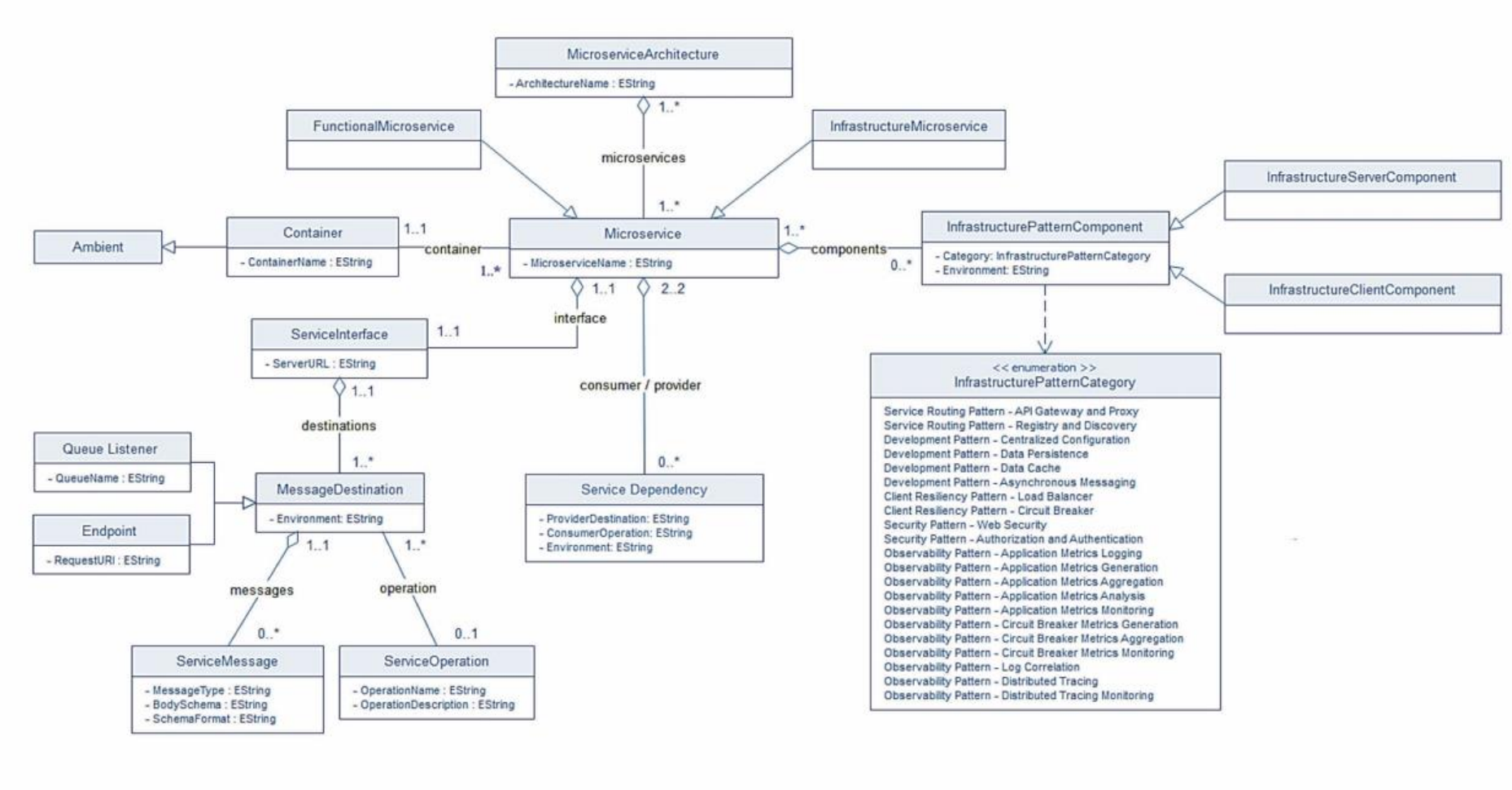

Figure 15: Final PIM metamodel (version 5).

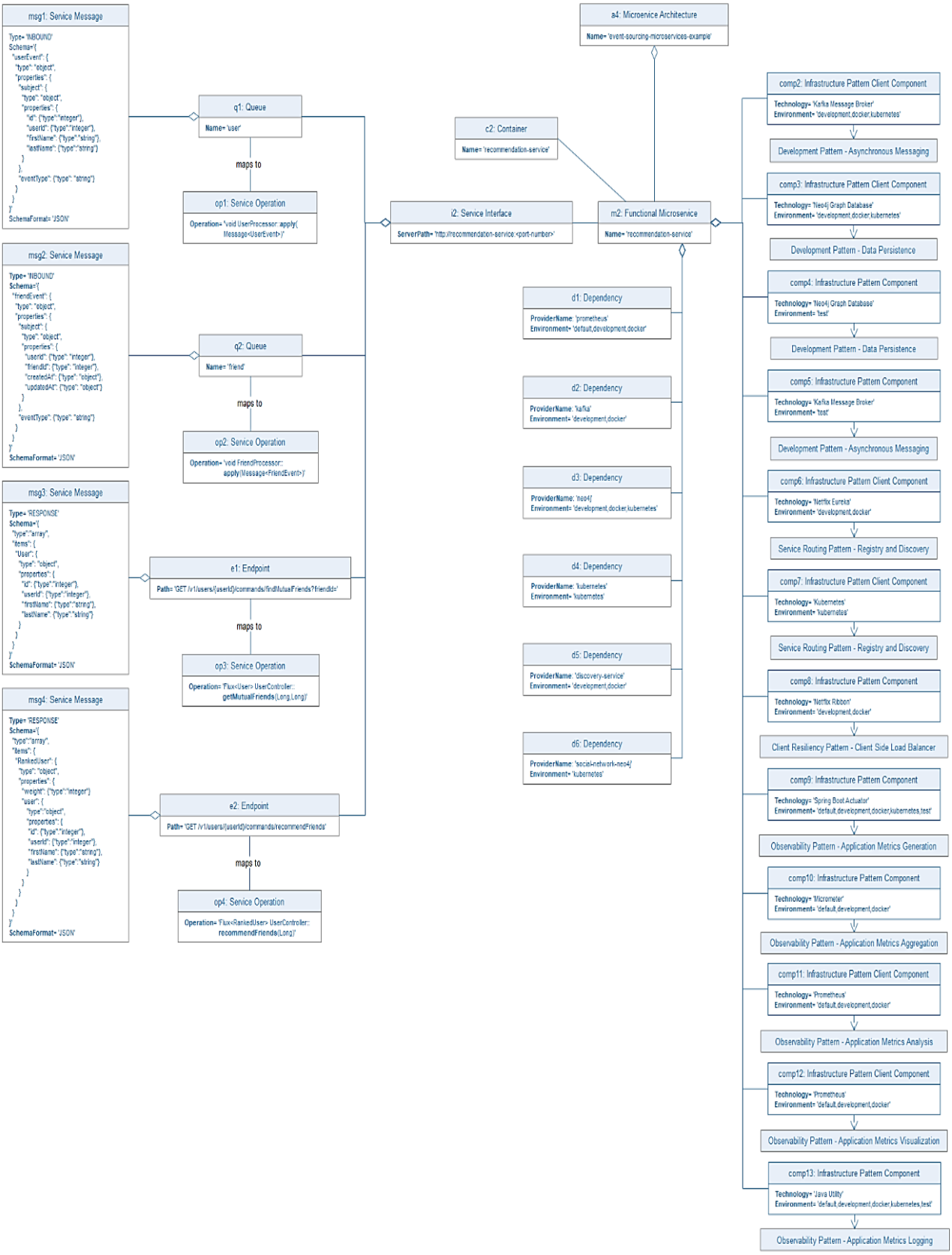

Figure 16: Enhanced PIM instance recovered for *recommendation-service* microservice from case study 4 based on enhanced PIM metamodel (version 5).