Model-based Testing of Adaptive Microservice Systems

Michalis Papapetros
michalis.papapetros@student.uva.nl

July 27, 2025
74 pages

Academic supervisor: dr. Ana Oprescu, a.m.oprescu@uva.nl
Daily supervisor: Aleandro Mifsud, aleandro.mifsud@axini.com
Host organization: Axini, https://www.axini.com/

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering

Abstract

Testing in Microservices Architectures (MSA) is challenging due to their distributed nature, independent deployments, and complex interactions. The problem becomes even harder when systems are adaptive, dynamically reconfiguring their components at runtime. Current strategies either focus on evaluating specific quality attributes or have a limited scope, resulting in untested system-wide behavior. Model-based Testing (MBT) has been proposed to address this issue, however, most studies either tested services in isolation or lacked thoroughness, leaving open the question of whether MBT is applicable and effective in this context. This research introduces a gray-box, End-to-End (E2E) MBT approach for adaptive microservice systems, using DYNAMOS, a research data exchange middleware, as a case study. The proposed method passively observes inter-component communication by leveraging the existing architecture and introducing a dedicated AMQP testing queue. A custom adapter is developed to translate system-specific messages into the symbolic labels of the Axini Modeling Platform (AMP). Two primary models are implemented in the Axini Modeling Language (AML) to represent the alternative execution paths and serve as the reference against which the System Under Test (SUT) is checked for specification conformance. The solution was evaluated through experiments where both expected and erroneous scenarios were executed against the developed models. Full state and transition coverage were achieved, while the majority of injected faults were successfully detected. Beyond fault detection, the testing process also exposed instabilities in the SUT. The results suggest that this approach provides a practical and structured way to assess the behavior of adaptive microservice systems. Despite this, it appears to identify the presence of defects more often than their root cause. For this reason, it is considered more suitable as an integrated part of the development cycle, complementing other techniques. This study highlights the potential of MBT for distributed environments and sets the ground for its broader application in the future.

Contents

Chapter 1

Introduction

Traditionally, software systems were developed as monolithic applications, where all components were tightly coupled and deployed as a single unit. While this approach simplified early development and deployment, it became increasingly unsustainable as systems expanded in size and complexity. To overcome these hurdles, a new pattern was introduced, termed microservices. Microservices Architecture (MSA) has grown significantly in popularity over the past decade, becoming a dominant choice for contemporary systems [1].

In contrast to the monolithic paradigm, microservices promote the design of applications as collections of independently deployable services, organized around business capabilities [2]. While microservices have numerous benefits, including scalability, flexibility, and resilience, they also pose several notable problems [3]. Software testing stands out as a significant one among them. Unlike monolithic systems, where testing is straightforward, microservices require a more sophisticated approach, mainly due to their distributed nature.

The issue is further intensified when the architecture consists of components that are adaptive, changing their behavior and structure at runtime. This complicates the construction of testing scenarios even more, rendering typical approaches inappropriate.

To address these challenges, a systematic and formalized methodology is required. A proven method to achieve this can be Model-based Testing (MBT), which provides a way to specify expected behavior and verify implementations, even as the system alters its internal state. This black-box technique offers a comprehensive understanding of system behavior, improves test coverage, and reduces manual effort.

1.1 Problem Analysis

On the one hand, the application of software testing methodologies to conventional systems is an established research area, with a plethora of tools and frameworks at our disposal. Model-based testing is one of them, providing a systematic means of evaluating behavior against formalized expectations.

On the other hand, studies reveal that microservices testing remains an insufficiently researched topic [3] and highlight the lack of dedicated frameworks [4]. Currently, the most popular approaches for defect detection in MSA-based systems are integration and unit testing [4], which are also common in traditional system scrutiny. However, we argue that both require a considerable amount of manual work and are not enough to test software in its entirety. Widely adopted solutions such as Pact [1] or Postman [2] narrow their focus only on certain aspects, namely contract and API testing, respectively.

Testing microservices is inherently challenging due to their distributed nature, asynchronous communication patterns, and independent deployment. Each service is autonomous and communicates with its environment over a potentially unreliable network, which introduces non-determinism and complicates the reproduction of system states and thus testing. Moreover, the presence of multiple interacting services makes fault localization difficult. Obstacles increase when dealing with self-adaptive systems, where components organize differently in response to internal conditions or external stimuli. As a result, the introduction of additional layers of intricacy mandates a more robust approach to perform thorough testing.

Recent studies have examined the intersection of microservices and MBT. Van den Brink et al. [5] attempted to test microservices as separate components without requiring full deployment, while Booy

[1] `https://pact.io/`
[2] `https://www.postman.com/`

et al. [6] replaced the underlying services of a system with mocks to test in isolation. Despite the merits of both efforts, the aspect of adaptivity remains largely unexplored. An initial investigation into this direction was conducted by Ganzevoort [7], who questioned the feasibility of applying MBT to an adaptive microservices context, opening the way towards addressing unresolved challenges.

Building upon these efforts, this study aims to further explore the application of MBT in self-adaptive microservice environments. We believe that model-based testing is capable of effectively representing the dynamic behavior and assessing the correctness of such systems. In this thesis, we aim to design models that accurately describe the system and to develop supporting mechanisms, with the ultimate purpose of improving fault detection and ensuring specification conformance of distributed systems.

1.2 Research Questions

Our research goal is to analyze *how model-based testing can be applied to adaptive microservice systems to assess their behavioral correctness.* We formulate two research questions that support this goal:

RQ1 : *How can the behavior of an adaptive microservice system be systematically observed and exposed to enable model-based testing?*

RQ2 : *What modeling strategies can be developed to specify the behavior of an adaptive microservice system?*

1.3 Research Method

The objective of this research is to explore the usefulness of model-based testing for microservice systems exhibiting dynamic behavior, using the Axini Modeling Platform (AMP). As a case study, we use the DYNAMOS middleware to evaluate the applicability of MBT in an adaptive execution setting.

To answer the research questions, we adopt a methodology structured in the following steps:

1. **Understand the system and identify key behaviors**: Analyze the architecture of the system to determine which aspects of its behavior should be captured and made visible to the testing infrastructure.
2. **Develop the integration mechanism**: Implement an adapter component that exposes internal behavior to the testing platform.
3. **Build specification models**: Design MBT models that represent system behavior.
4. **Test and evaluate**: Execute tests against the system to verify conformance and assess the effectiveness of the approach.

1.4 Contributions

Our research makes the following contributions:

1. A method for passively observing and exposing internal behavior in adaptive microservice systems.
2. An integration mechanism that connects adaptive microservice systems to a model-based testing engine.
3. A set of executable models that determine conformant system behavior under different configurations.

1.5 Outline

In Chapter 2, we introduce the foundational concepts of microservices and model-based testing. In Chapter 3, we present the rationale and structure of the proposed testing approach. In Chapter 4, we describe the development of the integration mechanism and the construction of the models. In Chapter 5, we outline the test setup and describe the experiments to be conducted using the developed models. In Chapter 6, we report on the testing outcomes. In Chapter 7, we interpret the results, reflect on the research questions, and address the limitations and threats to validity. In Chapter 8, we review work that is related to the concepts and methods developed. In Chapter 9, we summarize the main contributions and suggest directions for future research.

Chapter 2

Background

To better understand the work presented in this thesis, it is necessary to establish a common foundation. In this chapter, we introduce several key concepts. In particular, we cover the principles behind microservices and adaptive behavior, provide an overview of MBT, and examine their intersection. These topics offer the required context to reason about the approaches and challenges discussed in subsequent chapters.

2.1 Microservices

2.1.1 Overview

During a software architecture workshop in May 2011, the term "microservices" was coined as a means to incorporate the attendees' shared ideas in architecture patterns, evolving the then-emerging paradigm of Service-oriented Computing (SOC). In 2014, Martin Fowler and James Lewis popularized the term through a series of widely cited articles discussing its principles [2]. Microservices Architecture (MSA) aims to remove complexity and enable developers to construct manageable services with a single functionality [8].

According to a 2020 survey by O'Reilly [9], the vast majority of respondents have adopted microservices, with 92% reporting success. An analysis conducted by Statista in 2021 [10] reveals that microservices were embodied by 85% of esteemed organizations with over two thousand employees. Businesses that implemented microservices reported a marked increase in productivity, as stated in a study by DevOps Research and Assessment (DORA). Major technology enterprises such as Netflix, Amazon, Uber, and Zalando have successfully employed this architectural pattern and, in doing so, have pioneered their own implementations, sharing their experiences via technical blogs and industry reports.

Microservices are loosely coupled, independently deployable components, equipped with their own persistent storage [8]. They are relatively small, fine-grained services providing only a single business capability, making them ideal for development and maintenance by a single Agile team per service. They are typically packaged and deployed in the cloud using lightweight container technologies [11]. Their architecture is flexible, modular, and evolvable, offering faster delivery, improved maintainability, scalability, and autonomy. The most used architecture patterns are API Gateway and Backend for Frontend (BFF) [12]. Nevertheless, microservices face significant challenges, including coordinating distributed actions and maintaining reliable inter-service communication [11].

2.1.2 Communication

In traditional service-oriented architectures, communication between components was often based on orchestration, where a central coordinator, such as an Enterprise Service Bus (ESB), managed the interactions between services and controlled the overall workflow. In contrast, microservices architectures adopt a more decentralized approach known as choreography, in which services collaborate directly, without relying on heavy middleware. [8]. Fowler emphasizes this shift as a change towards "smart endpoints and dumb pipes" [2].

Microservice communication can be characterized either as synchronous or asynchronous and, based on the number of recipients, as *one-to-one* or *one-to-many*.

In a **synchronous** model, the sender's execution flow is suspended until the recipient processes the request and responds, typically referred to as *request/response*. If a reply is not returned to the client within a predefined time frame, a failure message is received. This one-to-one communication is also utilized when multiple recipients are involved. The most common protocol implementing this style is HTTP.

Conversely, **asynchronous** communication is a method in which the sender sends a message and continues without waiting for an immediate response. The one-to-one version of this is called *notification* style, often described as a fire-and-forget mechanism. The one-to-many variation is known as *publish/-subscribe*, where multiple recipients can subscribe to a topic to which a sender can publish information. The asynchronous model leverages an intermediate entity, usually a message broker, which ensures messages are forwarded from publishers and can be received by consumers. Protocols like AMQP are widely used to support this pattern. Examples of message brokers include RabbitMQ [1], Apache Kafka [2], and Azure Message Bus [3].

2.1.3 Adaptivity

Microservices Architectures are usually classified by design patterns. However, a less common categorization distinguishes between static and adaptive microservices.

Static microservices represent the conventional case, where services are deployed on a cluster and remain continuously available. They receive input, process data, and produce output in a long-running manner.

In contrast, **adaptive** (or **ephemeral**) microservices display dynamic behavior by adjusting to changing conditions at runtime. This may involve spinning up additional service instances to respond to a load increase [13], or deploying specific components to simulate production system testing [14]. A more specialized use case is the spawning of microservice chains in the form of jobs. These ephemeral services are created on demand according to predefined rules. They process and exchange data sequentially, and terminate once their task is complete. This self-adaptive pattern is the primary focus of this thesis.

2.2 Software Testing

Software testing is the process of verifying that a system behaves as expected and identifying defects. The cost of repairing a bug increases substantially from the moment requirements are gathered until the bug is discovered in production, according to numerous studies [15]. There are several different types of tests, each serving a unique purpose. Mike Cohn presented the Testing Pyramid [16] (see Figure 2.1), primarily to reason about test granularity and the number of tests, and highlight the importance of testing automation and constant review.

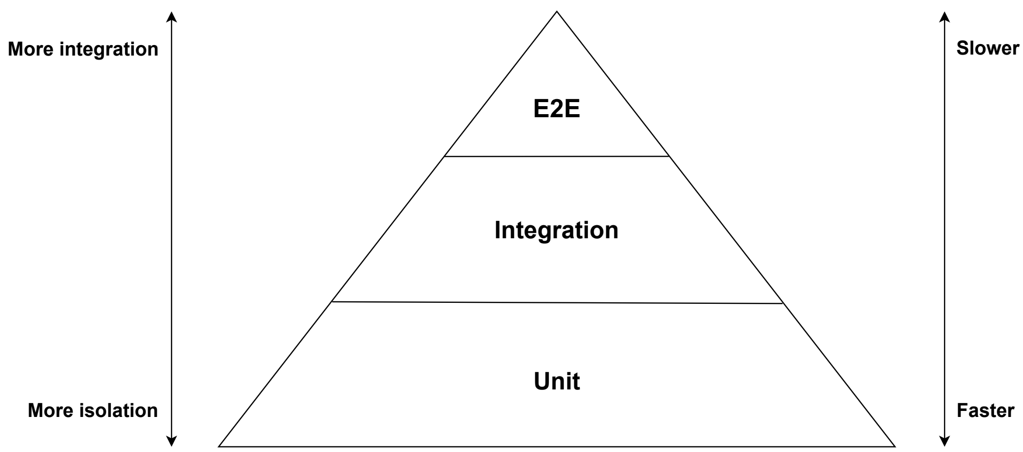


Figure 2.1: Testing Pyramid (Adapted from [16]).

[1] https://www.rabbitmq.com/
[2] https://kafka.apache.org/
[3] https://learn.microsoft.com/en-us/azure/service-bus-messaging/service-bus-messaging-overview

Fernandes [3] proposes a modified model adapted to the context of MSA (see Figure 2.2), suggesting that an effective testing strategy should address all levels, with a particular emphasis on automating tests at the lower layers. Their model incorporates the following layers:

1. **Unit Testing**: Verifying the functionality of individual components within a microservice.
2. **Integration Testing**: Checking the interaction between microservices and their dependencies.
3. **Contract Testing**: Verifying that microservice APIs conform to predefined interface contracts.
4. **Component Testing**: Testing each microservice as an independent component within the larger system context.
5. **End-to-End (E2E) Testing**: Ensuring the complete system's functionality from the perspective of an end user.
6. **Load Testing**: Assessing the system's performance and scalability under load.
7. **Resilience Testing**: Evaluating the system's ability to remain operational despite failures of individual components.

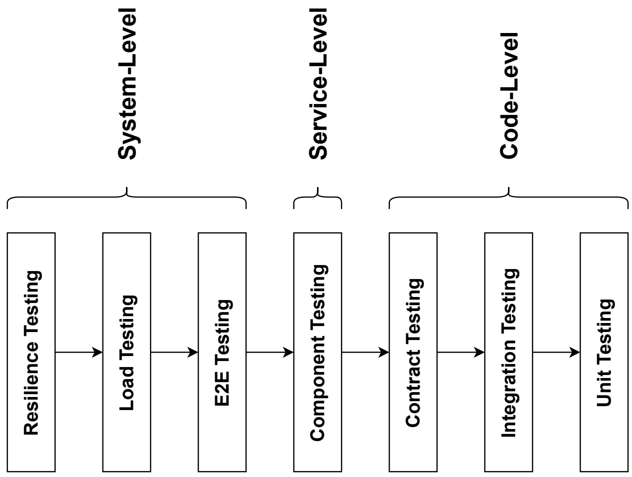


Figure 2.2: MSA Testing Pyramid (Adapted from [3]).

This thesis primarily targets the E2E and Component layers.

2.3 Model-based Testing

A well-established software testing method is Model-based Testing (MBT), which is the proposed approach of this research. According to the European Telecommunications Standards Institute (ETSI), MBT is "an umbrella of approaches that generate tests from models" [17]. More precisely, an implementation or System Under Test (SUT) is tested against a model that captures its expected behavior, in order to decide whether it conforms to its specification. In this formal technique, test cases are derived and executed from the model automatically. Test scenario generation is guided by coverage criteria defined over the model. The system is considered a black box exhibiting behavior and interacting with its environment. The tester interacts with the SUT solely through its externally visible interfaces, without access to its internal structure. For formal and complete definitions of the theoretical underpinnings of MBT, we refer the reader to [18]. In this section, we are going to summarize the core concepts used as a foundation for this thesis.

2.3.1 Models

The system may be modeled using diverse formalisms, namely Finite State Machines (FSM) or Labeled Transition Systems (LTS). To leverage existing research initially conducted by Tretmans [18], we choose to adopt an extension of the latter, called Symbolic Transition System (STS).

2.3.1.1 Labeled Transition Systems

A Labeled Transition System (LTS) is a formal model used to describe the behavior of a system in terms of its possible sequences of actions called *traces*. It consists of a set of *states* and *transitions*, where each transition connects two states and is associated with a *label* that represents an action that the system can perform.

Some actions are externally visible, while others are internal and unobservable, commonly represented by the τ (*tau*) label. In addition, the observable absence of output from the SUT is known as *quiescence* and is treated explicitly, as it can be meaningful in determining whether a system is behaving correctly.

LTSs are widely used in model-based testing as a means of reasoning about the behavior of reactive systems. By comparing the observed behavior of a system to the expected one described in the semantic model, it becomes possible to determine whether the system conforms to its specification.

2.3.1.2 Symbolic Transition Systems

A Symbolic Transition System (STS) extends the LTS formalism by introducing the notion of data, variables, and conditions dependent on data. Instead of listing every possible action explicitly, an STS uses symbolic parameters and conditions to describe many related behaviors in a generalized form.

In an STS, transitions can include parameters. Conditions (or guards) specify when a transition is valid, and updates may modify variables that represent part of the system's state. This allows the model to capture data-dependent behavior without the need to enumerate all possible values.

STSs are particularly well-suited for modeling intricate modern systems in a compact and expressive manner. A convenient aspect of STSs is that they can be interpreted as LTSs, allowing them to benefit from existing theory, including conformance relations.

2.3.2 Conformance

An implementation is said to conform if it behaves according to the specification described by the model. A formal definition of this relationship is the *Input-Output Conformance (IOCO)* relation, introduced by Tretmans [18]. Under IOCO, for every sequence of actions of the model, including quiescence (*Strace*), every output of the implementation after that sequence must be specified in the model. In other words, any output of the implementation must have been foreseen by the model. It is crucial to note that the model may be *underspecified* to allow modeling only part of the SUT, or *overspecified*, permitting multiple valid outputs. An assumption of the IOCO relation is that the system must be *input-enabled*, meaning that every state should have an outgoing transition for every input. Input-enabledness can be achieved by employing *angelic* or *demonic completion*.

2.3.3 The MBT Process

The model-based testing process typically follows four main steps [19]:

1. **Design a test model**
   The process begins with creating a model that captures the expected behavior of the system under test, written in a formalism such as a transition system. It serves as both a specification and a basis for test generation.
2. **Select test generation criteria**
   Once the model is in place, the next step is to determine how tests will be selected or prioritized. This can involve structural coverage criteria, such as ensuring all transitions or states are traversed, or other strategies such as boundary value exploration.
3. **Generate the tests**
   Based on the chosen criteria, test cases are automatically generated from the model using a respective algorithm. Depending on the strategy, this can be conducted *offline* (before execution) or online (during execution). Each test consists of a sequence of inputs and outputs that will be compared against the system's real-world behavior.
4. **Execute the tests**
   The generated tests are executed against the implementation. A critical component of MBT is the **adapter**, which acts as a bridge between the abstract model and the system. During test execution, the SUT's responses are captured and compared to the model to determine whether each test passes, fails, or results in an error.

The above process enables systematic and automated testing, allowing both broad coverage and repeatable evaluation of system correctness. In agile development environments, this process is repeated as part of the iterations [20].

The online approach is depicted in Figure 2.3.

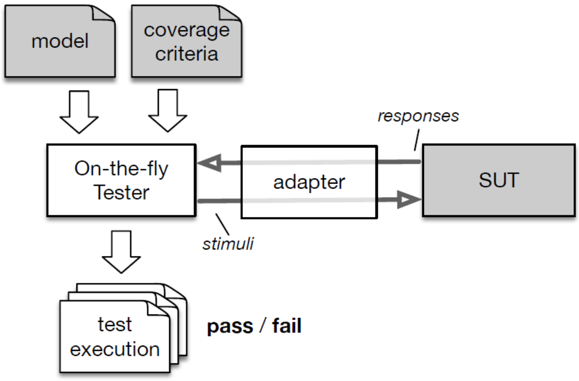


Figure 2.3: Online Testing: Test cases are generated step-by-step during interaction with the SUT [21].

2.3.4 Axini Modeling Platform

In this thesis, MBT is applied using a commercial tool called the *Axini Modeling Platform (AMP)*. AMP supports the construction of symbolic models in the form of an STS, the automatic on-the-fly generation of test cases, and the execution of tests against a running system, whose verdict is determined using IOCO. The overall architecture of AMP is shown in Figure 2.4.

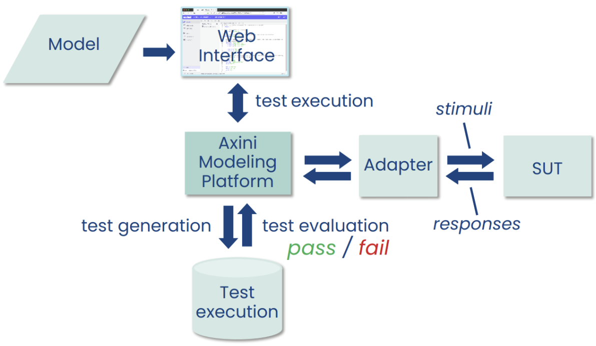


Figure 2.4: AMP Architecture [21]

Models in AMP are defined using a Ruby-based Domain-Specific Language (DSL), namely the *Axini Modeling Language (AML)*, and include a visual representation. Figure 2.5 illustrates a simple AML model.

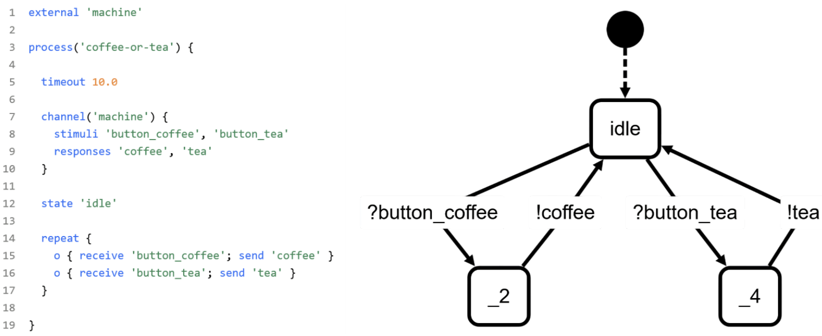


Figure 2.5: AML model of a machine that dispenses beverages.

Communication between the model and the SUT is handled through the adapter mentioned previously. This component is responsible for translating logical to physical messages and vice versa, where inputs are referred to as *stimuli* and outputs as *responses*. Adapters are typically designed to be as uncomplicated and stateless as possible. A minimal adapter aids in avoiding delays, unintended behavior, or incorrect assumptions about the SUT. Despite this, unconventional systems such as adaptive microservices display complex behavior, which may justify the development of a more sophisticated adapter, as it plays a pivotal role in the modeling process. This study complies with the plugin adapter policy developed by Axini [4]. AMP combined with the adapter simulates the environment of the SUT.

2.3.5 Benefits & Considerations

Model-based testing offers a number of benefits. A considerable change in modern practice is MBT's movement towards earlier phases of development, enabling the validation of requirements and early detection of defects [22]. This "shift left" approach helps reduce the cost of errors by identifying them before implementation is complete. Furthermore, the effort traditionally spent on test scripting is transferred to model construction. The model becomes the central artifact, from which tests can be automatically derived and evaluated. Moreover, when expected states or outcomes are included in the model, it also serves as a test oracle, addressing the *oracle problem* by automating both test generation and verdict assignment [23]. Finally, models are generally easier to maintain than large test suites, and changes in system behavior can often be reflected with model updates, enabling rapid reproduction of test cases.

Nevertheless, MBT also presents several challenges. The upfront cost of modeling is significant, and ensuring that the model is both accurate and maintainable requires domain expertise [24]. Additionally, developing adapters to connect models to real systems can be complex and time-consuming. In some cases, symbolic models can lead to an explosion of test cases, making careful modeling and test selection strategies essential [20].

2.3.6 Applying MBT to Microservices

The application of model-based testing to MSA presents both challenges and opportunities. The main attributes of microservices complicate conventional testing methodologies that depend on static structures and centralized control. As such, microservices are challenging to test extensively, especially when functionality relies on interactions between ephemeral components.

MBT offers potential advantages in this context. Since microservices typically expose clear input-output interfaces and communicate over well-defined protocols, their observable behavior aligns naturally with the orientation of MBT. The finding that microservices are intrinsically input-enabled [6] also emphasizes their compatibility. Moreover, since communication happens over the network, it is possible to intercept the messages through an adapter without modifying the service logic itself. However, applying MBT requires tool support for the communication protocols involved, appropriate handling of data dependencies, and strategies for dealing with distributed state.

Recent work has examined various approaches to bridge this gap, such as mocking [6] or compositional testing [5], and, in the context of adaptive systems, several challenges have been identified [7]. These include the risk of state explosion, the impact of symbolic data on coverage, and the integration of persistent components into the testing scope. These observations suggest that while MBT aligns conceptually with the nature of microservices, its practical application involves a number of thoughtful decisions.

[4] `https://github.com/Axini`

Chapter 3

Design for MBT Application

In order to apply MBT to an adaptive microservice system, certain design decisions must be made to enable the systematic observation and interaction with the system. This chapter introduces the DYNAMOS middleware as the System Under Test, explains the necessity and design of the intermediate adapter component, and discusses the main considerations related to the system and modeling.

3.1 DYNAMOS

In this thesis, we use DYNAMOS as a case study for the application of model-based testing to adaptive microservice systems. Dynamically Adaptive Microservice-based OS is an open-source research platform designed to facilitate data exchange between distributed parties based on formal policy agreements. It builds on earlier work on data-sharing infrastructures, such as the Digital Data Marketplace (DDM) architecture [25]. The scenario chosen to test DYNAMOS with is the *AMdEX UNL* [1] *use case* [2], which is analyzed below.

A *data analyst* initiates a request to obtain data from distributed datasets, which are subject to policy agreements between the involved institutions. Upon request approval, the client queries the *data stewards*. This interaction reflects a realistic scenario where organizations collaborate while retaining control over their data. DYNAMOS coordinates this process through the appropriate components.

3.1.1 Core System Components

The execution of a request involves the following central entities:

- **API Gateway**: Serves as an interface to the outside world, forwarding requests to the system and vice versa.
- **Orchestrator**: Acts as the central coordinator. It handles request validation through the Policy Enforcer, selects the appropriate data-sharing pattern, generates a job specification, and communicates with the Distributed Agents to initiate execution.
- **Policy Enforcer**: Checks whether a requested data-sharing operation complies with the established formal agreements between institutions.
- **Distributed Agents**: Run on systems of Data Stewards. They receive instructions from the Orchestrator and spawn the appropriate microservice chain.

Figure 3.1 illustrates the interaction between the above components. The API Gateway routes client requests to the Orchestrator. After validating with the Policy Enforcer, the Orchestrator communicates with the Distributed Agent(s) to initiate execution, while the API Gateway forwards them the initial request. Upon completion, the result is passed back to the client via the API Gateway.

[1] `https://www.universiteitenvannederland.nl/en`
[2] `https://amdex.eu/news/sharing-research-data-under-ones-own-conditions/`

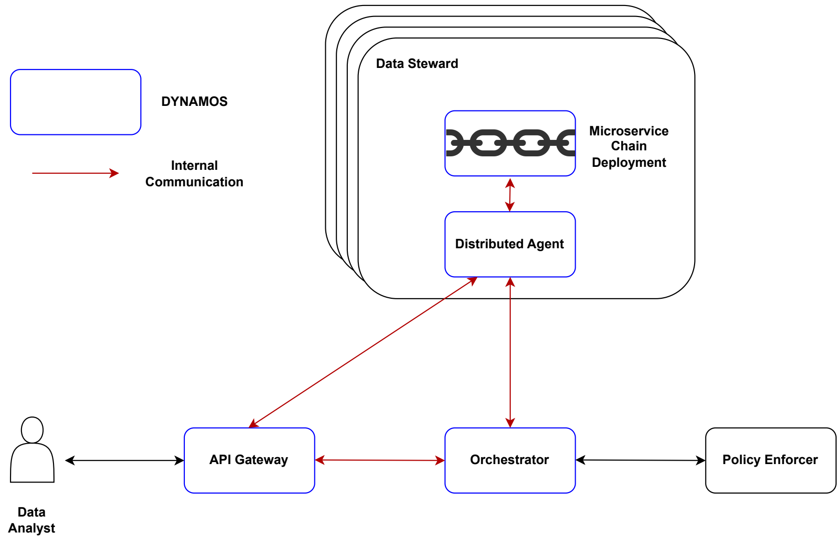


Figure 3.1: High-level representation of DYNAMOS components (Adapted from [26]).

3.1.2 System Architecture

From a technical point of view, DYNAMOS is deployed as a set of containerized services managed by a Kubernetes [3] cluster, each being in a separate pod. Within each data steward, a dedicated *data store* hosts the datasets that the client queries on. The system also relies on a shared `etcd` [4] key-value store accessible to the Orchestrator and Agents for maintaining system state and objects. Microservices are implemented in Go, although the system is language-agnostic. Since communication is standardized, the implementation language is irrelevant from both an integration and testing perspective.

The architecture contains both static and dynamic components. Core services are deployed ahead of time and remain active throughout the system's operation. In contrast, microservice chains are instantiated dynamically in response to incoming requests. Each chain is a Directed Acyclic Graph (DAG) of lightweight services that perform specific processing steps based on the selected plan. These chains are spawned per request as Kubernetes jobs and are automatically terminated after execution completes. Since these components are ephemeral, this behavior represents the adaptive aspect of DYNAMOS, which is central to this study.

Each unit is deployed together with a secondary container known as a *sidecar*. The sidecar acts as a proxy for all communication, handling message passing on behalf of the main microservice. This design decouples business logic from the exchange layer. In the case of microservice chains, it also serves as the entry and exit point for each service, routing messages and ensuring correct sequencing as data progresses through the chain. For the purposes of this thesis, **the sidecar plays a central role**, as it captures the observable behavior targeted during model-based testing.

3.1.3 Communication

In general, communication within the system occurs across three layers, each using a different protocol. First, external communication between the user and the platform utilizes HTTP. The user interacts with the system by sending requests to the API Gateway, which serves as the entry point and forwards these requests internally. The results are also returned using the same protocol.

Second, communication between major components is handled through a RabbitMQ message broker using the AMQP protocol. However, these components do not interact with it directly. Instead, each sends and receives messages via its respective sidecar. The component issues gRPC [5] calls to it, which serializes the message and communicates with RabbitMQ.

Third, within each microservice chain, data is streamed between services in a unidirectional manner using gRPC. Here, the sidecar also acts as the messaging layer, providing endpoints to certain services

[3] `https://kubernetes.io/`
[4] `https://etcd.io/`
[5] `https://grpc.io/`

and forwarding messages through the configured job queue.

DYNAMOS defines two gRPC services using Protocol Buffers (Protobuf) [6]:

- `Microservice`, which exposes an interface for sending internal communication events. These may include the triggering of microservice chains or other interactions within the system. All payloads use the `MicroserviceCommunication` message format, which contains structured data, metadata, and routing details.
- `RabbitMQ`, which provides a broader set of coordination and messaging Remote Procedure Calls (RPCs). Among these is `SendMicroserviceComm`, which also accepts the same type of message and delivers it to the appropriate queue.

All gRPC calls are unary and delegated to the sidecar, which asynchronously forwards the message content to RabbitMQ. This design keeps the components decoupled from the details of message delivery.

Figure 3.2 depicts a more detailed view of DYNAMOS' architecture, including the communication type, messages, sidecars, data stores, and microservice chains.

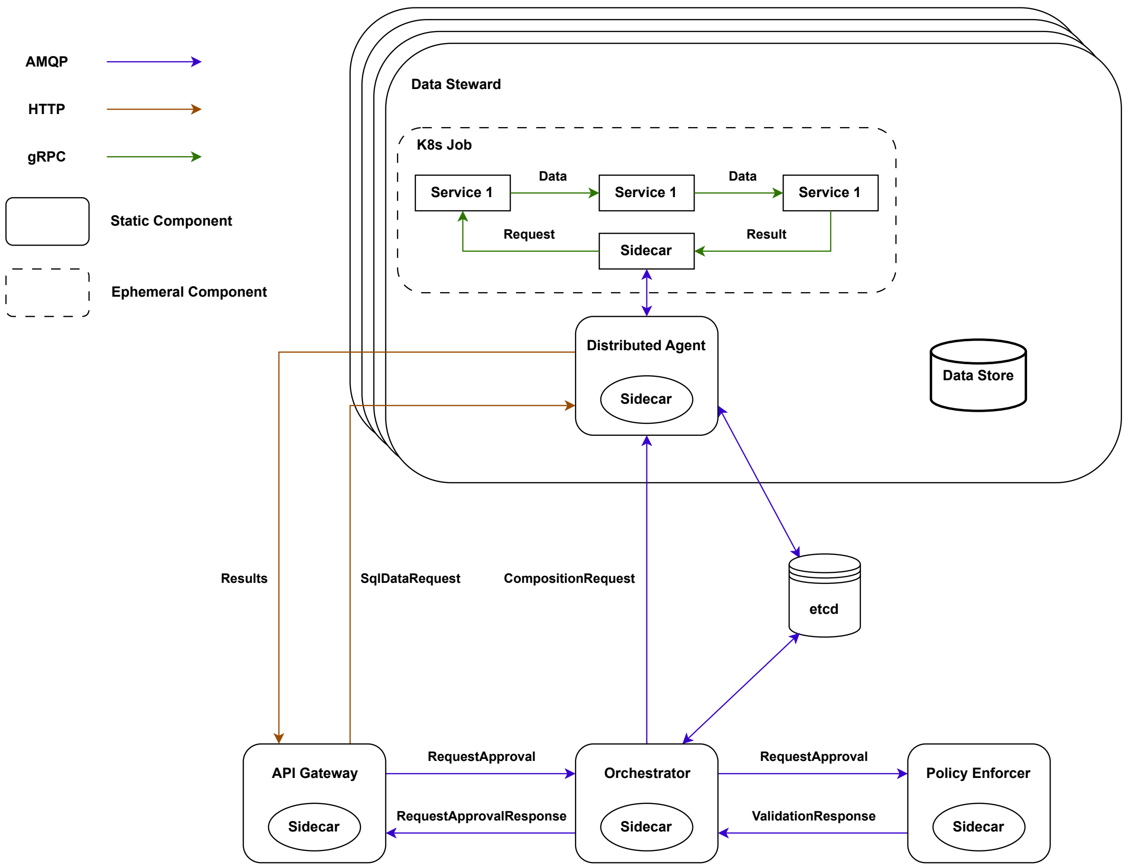


Figure 3.2: The architecture of DYNAMOS (Adapted from [26]).

3.1.4 Archetypes

The UNL use case specifies different data-sharing patterns between organizations based on their level of trust, which are called *archetypes*. An archetype determines the way data and computation are distributed, how intermediate results are combined, and what responsibilities are assigned to each participant.

In practice, once a request is submitted and validated, the Orchestrator selects an archetype based on request type, policy agreements, and available components. This choice influences how the system behaves at runtime, making it self-adaptive. The Orchestrator then constructs a job specification and instructs the relevant Distributed Agents to deploy a corresponding microservice chain. The resulting topology is thus not predetermined, but derived dynamically.

[6] `https://protobuf.dev/`

The responsibilities of each Data Steward and therefore each Agent are formalized as *roles*:

- **computeProvider**: Responsible for executing a computation task.
- **dataProvider**: Responsible for the provision of access to local datasets.
- **all**: Combination of the above roles.

DYNAMOS implements two archetypes, as defined by Shakeri et al. [25]:

- **Compute To Data (computeToData)**
  In this archetype, a single Agent assumes the `all` role. The Orchestrator sends a composition request to the Agent, followed by a data request from the API Gateway. After processing, the result is returned to the API Gateway. DYNAMOS contains a use case related to this scenario with dummy `UVA` and `VU` agents.
- **Data Through Trusted Third Party (dataThroughTtp)**
  This archetype introduces a trusted third-party Agent as the `computeProvider`, while one or more Agents act as `dataProvider`. The Orchestrator sends composition requests to all participating Agents, and the API Gateway sends a data request to the Trusted Third Party (TTP). The TTP then issues data requests to each data provider, receives their responses, performs the computation, and finally returns the result to the API Gateway. In DYNAMOS, this archetype is implemented using a `SURF` Agent as the TTP, with `UVA` and `VU` Agents providing data.

As we discussed previously, once an archetype is selected and roles are assigned, each Agent deploys the appropriate microservice chain. Each chain contains a set of required microservices and may include optional services. For instance, all chains require at minimum a *query service*, which executes a Structured Query Language (SQL) query on the local dataset, and an *algorithm service*, which applies a computation such as a wage gap analysis.

Additional microservices are:

- The **anonymize service**, which is added when specific columns must be masked before data leaves a steward's domain.
- The **graph service**, which is included when the data analyst requests graphical output in the request specification.
- The **aggregate service**, which is used only in the `dataThroughTtp` archetype, where partial results from multiple data providers must be combined before final processing.

Figure 3.3 shows a valid microservice chain, distinguishing between required and optional services and specifying the role responsible for its deployment.

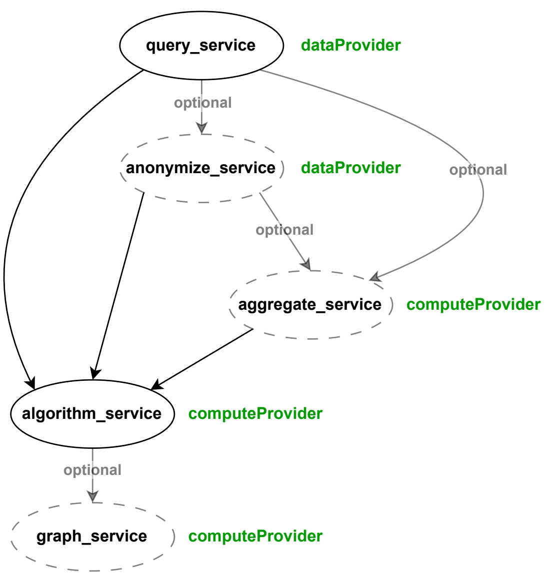


Figure 3.3: Microservice chain for a SQL data request [26].

From a testing perspective, these chains are particularly important, since they define the precise sequence of operations and hence the observable behavior exposed at the sidecar interface.

3.2 MBT Integration Design

Having established our target system, we may proceed to employ a suitable design strategy for model-based testing. In this section, we aim to define the aspects that need to be considered with respect to the system under test, the adapter, and the model. In particular, we examine the considerations that are pertinent to DYNAMOS, as its distinctive characteristics require thoughtful design decisions.

3.2.1 System-level Considerations & Design

3.2.1.1 Defining the SUT

A crucial part of the early model-based testing phase is delimiting the system we are analyzing. This refers to both choosing the appropriate level of granularity and scope. If this is not clearly defined from the beginning, then we may encounter obstacles related to the layer at which we are examining and the range of services that are included. This decision is not always trivial, as it is influenced by various factors. One of them is the determination of the testing objective. If we aim to examine at the code level, for example, we should choose unit testing instead of MBT. Another factor is the degree of autonomy. If we have the freedom to investigate code, or even change it, then, despite the obvious advantages, it can also be a hindrance, because the appropriate layer of abstraction must be manually constructed. An additional parameter affecting this choice is ignorance. When starting to experiment with a system, a tester is unaware of internal components in their entirety and how they communicate with each other. This implies that there may be constraints that are not initially apparent, such as a potential inability to observe or receive communication from a particular service. All of the above concludes that it is vital to have a clear understanding of the system under evaluation from the outset in order to conduct effective testing and implement any necessary modifications that may arise.

In our approach, the objective is to perform E2E testing with elements of component testing on DYNAMOS. As a result, despite MBT being considered black-box, it can seamlessly be converted to *gray-box*, which is desired in this case. There are numerous benefits to following this methodology. The primary one is system-wide validation, since testing the system as a whole confirms that all services function properly under real workloads. The gray-box element signifies that we are able to observe behavior one layer deeper and thus more thoroughly, while still obtaining the complete picture. Another advantage is production-like testing, which, unlike mocking, reveals real interactions that are hard to replicate in isolation. Furthermore, this method allows us to possess observability in adaptive systems. For microservices that are dynamically instantiated, we gain insight into how effectively the system handles scaling, failure recovery, and resource allocation in real time.

It is apparent that applying MBT at the system level comes with trade-offs. Tests run against real deployments, making them slower and harder to manage than mocked setups. The adaptive behavior of microservices adds complexity to orchestration and debugging, while failures can be harder to isolate. However, we believe that the benefits outweigh the costs, since these obstacles can be overcome through appropriate actions.

In view of the above, we choose to define the SUT as the set of DYNAMOS components and, more specifically, the communication layer at which they communicate. This means that our approach is not concerned about the entities' internal implementation, only about what is observable through the interface they select to expose. The components include the API Gateway, the Orchestrator, the Policy Enforcer, the Agents, and the respective Microservice Chains. By adopting such a tactic, not only is the end-to-end goal achieved, but also the objective of verifying compliance at an appropriate internal level. This is mainly possible due to the ability to add code within DYNAMOS in order to intercept inter-component communication.

3.2.1.2 Observing the SUT

Model-based testing is traditionally applied to reactive systems, a class of systems that respond to external inputs with observable outputs. A coffee machine, for instance, produces a drink in response to a button press. By contrast, microservice systems are harder to reason about, since they are primarily data-driven, among other things. DYNAMOS is not an exception, let alone considering that it is a large system with many interacting components, while also exhibiting adaptive behavior. As a result, understanding the SUT's behavior as a first step towards modeling it becomes nearly impossible when solely relying on code examination. Most real-world systems are accompanied by an official specification document which can provide a source of information. However, since DYNAMOS is a research tool, there is no such artifact, but a considerable amount of information can be collected from various parts that have been explained, serving as an extra point of reference.

To address this, various tools may be employed to explore the system's interactions in practice. These can help reveal the structure and sequencing of messages, and the timing between components. By combining such techniques, a comprehensive picture of the system can be drawn, facilitating the modeling process at a later stage.

One tool that we use is Postman, primarily due to its ability to test an API with minimal configuration. In addition, its versatile capabilities allow the tester to create an environment where they can experiment with different types of requests in an automated way with the support of scripts. It is a suitable tool for our case, because the initial client request on DYNAMOS is performed using the User Interface (UI), which can be adequately simulated by a simple HTTP request. After trying a broad range of requests, we see that well-formed requests yield responses in a concise text format, which is a positive indicator that this HTTP communication is already sufficient for modeling at the highest level since MBT allows it.

The choice of the second tool arises from the fact that DYNAMOS already utilizes observability, with Jaeger [7] being responsible for distributed tracing. This feature is especially relevant, because it provides context with respect to the communication flow between components. Therefore, upon performing the request through Postman, we are able to track the message sequence in a timely manner. The core verdict from this process is that only a limited yet useful subset of exchanged messages is present, potentially due to minimum points in the code being recorded. This only reinforces the belief that a holistic approach such as MBT is necessary to perform thorough testing in such a setting.

3.2.1.3 Leveraging the Sidecar

Since DYNAMOS is a distributed system comprised of microservices, the exchanged data that our approach aims to capture originates from scattered and self-contained components in the network. The fact that it does not come from a single source is a major challenge. One potential strategy for collecting messages from multiple senders is attempting to inject code in every component of interest, so that our intermediate mechanism can capture data from each one. The most obvious drawbacks of this are being laborious, because each communication channel needs to be handled separately, and also inconsistent, given that components may utilize different formats internally. In section 3.1, we emphasized the significance of the sidecar within DYNAMOS, which effectively splits application logic from communication handling, promoting the single-responsibility principle. Therefore, one core decision is to take advantage of this sidecar in order to have a unified way of intercepting communication. By adopting this approach, the only location in the current system that needs to be modified is within the sidecar, streamlining the process by allowing us to primarily focus on the testing aspect. Whether this is sufficient to monitor communication from both static and ephemeral entities remains to be determined during the implementation phase. This is because transient parts may require special handling due to their adaptive behavior, in which they might not be able to communicate properly.

3.2.1.4 Proposed Testing Architecture

The selection of the sidecar's abstraction implies that a mechanism needs to be identified in order to transmit test-related responses to the adapter from it. One way of achieving this can be by logging the exchanged communication in dedicated files to which the adapter has access. This becomes troublesome, because separate logic for file management will need to be implemented, where complex state must be retained during a test case and cleared between different test runs, leading to unnecessary overhead. Another potential option could be receiving data from the recipients' side, which is possible due to the majority of system communication being exchanged using asynchronous messaging. In practice, this would entail subscribing to every single AMQP queue as a normal receiver. It is evident that this approach is also problematic, since the adapter would need to handle every queue separately, hence obstructing scaling in case the system incorporates new components in the future and creating multiple points of failure. Our solution to this problem lies in altering the perspective from which communication

[7] `https://www.jaegertracing.io/`

can be observed, specifically the sender's viewpoint. To accomplish precisely that, we introduce a **testing queue**. The mode of operation of this mechanism is configuring the sidecar to transmit messages to both the designated original destination and the testing queue. Employing this methodology signifies that messages are duplicated with negligible cost, while also guaranteeing that standard system operation is not disturbed.

Figure 3.4 represents the testing architecture we are going to apply based on the options discussed.

The system functions in the following manner: Upon receiving a stimulus, SUT components are triggered as usual, namely handling incoming input using their sidecar, processing data, and forwarding output using the latter once again. The only difference is that outgoing messages are also sent to the MBT testing queue, whose sole responsibility is to have its contents (responses) consumed by the adapter. This passive approach successfully captures end-to-end communication and preserves the flow, since every critical component utilizes the sidecar regardless of testing.

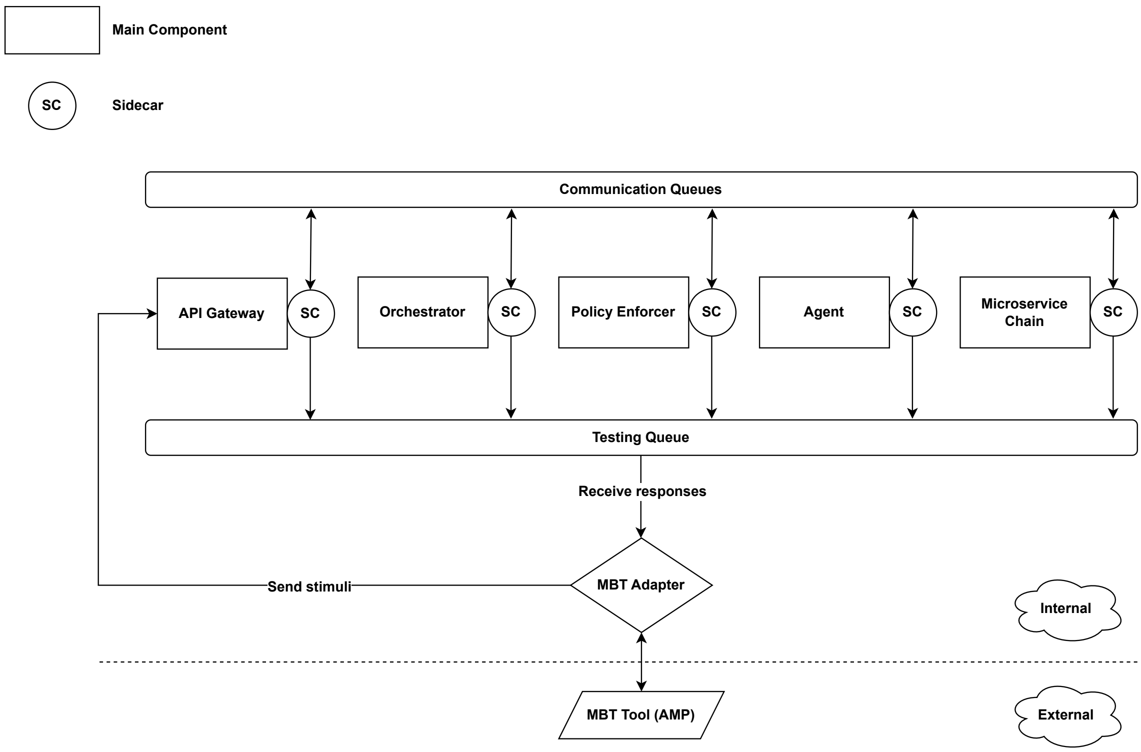


Figure 3.4: Abstract architecture of the E2E MBT framework.

3.2.2 Adapter Design

Previously, we established that E2E model-based testing mandates the non-intrusive observation of communication between internal gears of the system. The natural subsequent step is to define the intermediate entity that is going to be located between the SUT and the modeling platform, namely the adapter. Without it, effective communication between them would not be possible. The adapter's core purpose is to serve as a translator while simultaneously maintaining its underlying logic at minimal levels to simulate authentic testing conditions.

3.2.2.1 Adapter Position

Figure 3.5 illustrates the horseshoe paradigm, which defines the position of the adapter and modeling platform in relation to the system.

In our setup, the adapter constitutes an internal system element. This is to conform to Axini's Plugin adapter protocol [27], which states that adapters shall operate as close to the SUT as possible. On the one hand, the image captures the proximity between the system's interface and the adapter, which operates right outside the border of the SUT. On the other hand, its strong correlation with the testing platform suggests that this path is of equal importance and communication through it must be carefully maintained.

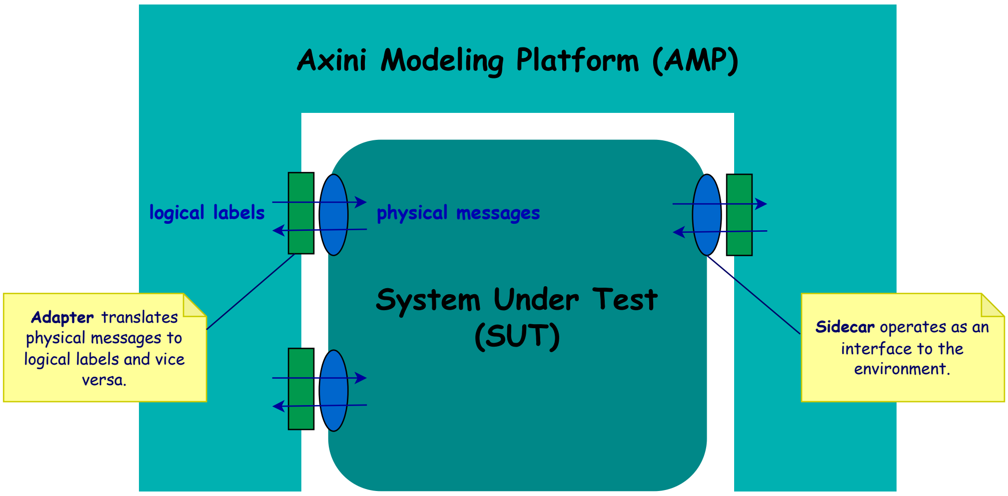


Figure 3.5: The horseshoe paradigm.

3.2.2.2 Determining the Requirements

Designing an adapter that links model-based testing with a microservice system such as DYNAMOS requires careful consideration. The adapter must not only translate between abstract test inputs and concrete system interactions, but also operate within the behaviors and restrictions of a distributed architecture. To clarify the scope and expectations of this component, a set of high-level requirements is identified. Each requirement reflects a specific need or constraint that emerges from the characteristics of DYNAMOS, the MBT process, or the interaction between them. These are presented below, each with a short title and a description of its role in shaping the adapter's functionality.

Req-01-Ad: Bridging Protocols and Data Formats
As we highlighted earlier, the adapter's primary role is to enable communication between the SUT and the testing platform, despite using different protocols and data formats. This bridging is essential for ensuring that both systems can understand each other and collaborate effectively. All DYNAMOS communication mechanisms must be considered by the adapter, handling the challenge of supporting multiple protocols and message formats as required by the system. By translating messages and preserving their meaning, the adapter makes it possible to automate testing, reducing errors that could arise from incompatibility.

Req-02-Ad: Managing Connections
For the adapter to function as a dependable intermediary, it must be able to manage connections to both the SUT and AMP. This involves not only establishing these connections at the right moments, but also monitoring their status and terminating them when needed. Effective connection management helps ensure that communication remains stable throughout the process. It also allows the adapter to gracefully recover from failures, such as AMQP connection drops, minimizing downtime as much as possible and reducing the risk of losing data.

Req-03-Ad: Controlling Lifecycle and State
The adapter must manage its own lifecycle and internal state carefully to achieve uninterrupted and predictable operation during testing. This includes handling transitions such as initialization, configuration, running, resetting, and shutting down. If it maintains control over these phases, it can coordinate its activities with both the system under test and testing platform properly, ensuring that each step happens at the appropriate time. Figure 3.6 depicts the protocol's simplified STS with respect to the adapter, which is useful to comprehend its lifecycle.

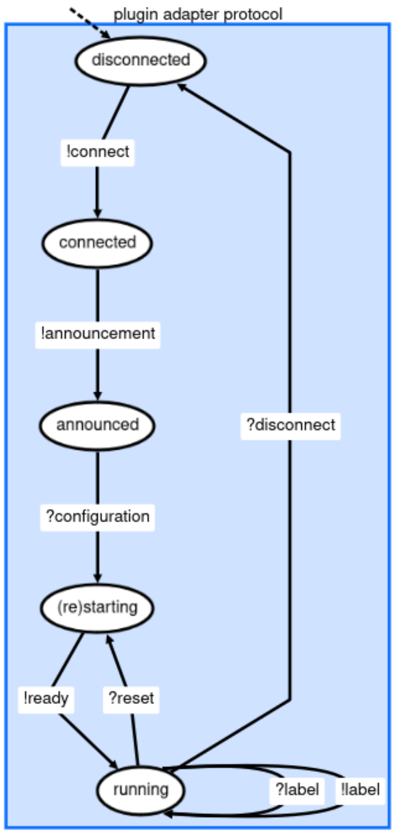


Figure 3.6: Visualization showing the state machine of the plugin adapter protocol. Error handling behavior is omitted for clarity [27].

Req-04-Ad: Message Handling

The adapter should guarantee that all messages between DYNAMOS and AMP are handled with reliability and without unnecessary delays. They must be processed asynchronously, allowing multiple messages to be managed at once and preventing bottlenecks. This approach maintains an efficient and orderly flow. In the context of the given system, this means processing messages received from queues, which may arrive out of order or in bursts, and ensuring that all relevant responses are forwarded to AMP without loss.

Req-05-Ad: Error Handling

The adapter should detect and log errors during communication with either side. It must send error notifications to AMP when issues occur and close the connection if necessary. When a connection drops or an error is detected, the adapter should attempt to recover by reconnecting and restoring its previous state, achieving little to no disruption. For DYNAMOS, this includes handling malformed SUT messages and request failures with appropriate error reporting and recovery actions.

3.2.3 Design for Modeling

3.2.3.1 Platform Requirements

Before identifying the modeling approach, it is important to understand what is expected from the testing platform in order to support the modeling effort. The following requirements describe the necessary capabilities to connect with the SUT, configure and execute test cases, and express the intended behavior at the right level of abstraction. These aspects are essential for making the modeling process both feasible and effective.

Req-01-Md: Adapter and Label Registration

Before testing can commence, the platform must be able to establish a connection with the adapter and correctly register its capabilities. This includes recognizing the adapter instance, as well as the symbolic labels it supports and exposes. Once the labels are obtained, the platform should be able to manipulate them according to the logic defined in the test model. This enables the platform to map between model actions and system-level stimuli and observations.

Req-02-Md: Configurable Test Execution
The tester must allow for configuration of key testing options. This enables customization of the testing process to suit the objectives of the test campaign. Examples include the selection of the test strategy, the maximum number of steps per test case, and solver.

Req-03-Md: Label Parameter and Path Variability
The platform must support variation in both label parameters and paths of the Symbolic Transition System. This includes the ability to select parameter values randomly, from predefined sets, or via constraint solving. This level of control allows the exploration of different behaviors across test cases.

Req-04-Md: Support for Structured Logic and Abstractions
The language must offer modeling constructs that enable the grouping of reusable behavior and the expression of complex logic. This involves support for variables, behavioral abstractions, and constraint-based conditions. Such capabilities are critical for maintaining model clarity, avoiding duplication, and expressing with accuracy the conditions under which specific behaviors should be enabled or not.

Req-05-Md: Abstraction of Distributed Systems
Despite low-level communication being handled by the adapter, the platform should still allow the model to express the behavior of a distributed system in an abstract way. In the context of microservices, this means being able to describe how different components interact and coordinate, without referring to the internal mechanics of message passing. This way, the model focuses on behavior, rather than implementation details.

3.2.3.2 Modeling Approach

The core characteristic of the model is capturing the behavior of the system as accurately as possible, including both its interaction with the external environment and its coordination with internal components. This is crucial because MBT approaches testing from the perspective of the system itself. As such, the model must closely simulate how the system operates, as if it were acting in the first person. Achieving this involves receiving external stimuli and managing message sending as if the model itself were the system. This, however, introduces the challenge that the adapter conceptually functions in the opposite direction. Its role is to inject stimuli and observe responses. This asymmetry must be handled with care, but it is fundamental to preserving the correctness and precision of the overall testing activity.

Another critical challenge that ought to be addressed is the simultaneous handling of multiple communication protocols. This obstacle needs to be overcome so that modeling is not affected by the specific characteristics of each protocol or the format of exchanged data. The model should instead present a unified, protocol-agnostic interface. This allows different protocols to be treated uniformly, which improves readability, simplifies the modeler's task, and ensures that the model remains resilient in case the system's communication mechanisms change in the future. To achieve this, beyond the necessary handling on the adapter side, the model itself must maintain awareness of the mapping between labels and the protocols to which they belong in each context and act accordingly.

It should not be overlooked that the system usually responds in an asynchronous and unpredictable manner. The model must be equipped to account for this and to navigate the often subtle distinction between acceptable outcomes, questionable behavior, and clearly invalid responses. This task demands careful judgment, especially when responses do not arrive in a consistent fashion.

As discussed earlier, DYNAMOS does not follow the behavior of a typical reactive system. A more accurate analogy for its operation is that of a fire-and-forget mechanism. Conceptually, it behaves like a streaming server that, upon receiving a request, internally initiates the sending of a data flow before returning the result to the client. This requires us to model a process that unfolds in parallel with the actual response, a pattern that is not commonly encountered in model-based testing.

One of the most decisive factors in modeling this system is the need to account for the archetypes discussed in section 3.1. These are the source of the system's adaptive behavior and determine the roles that Agents assume depending on the request type. As previously explained, they govern which microservice chains are spawned and what components these chains consist of. It is therefore necessary to define a modeling strategy that can encompass this dynamic behavior in a coherent and controlled way.

In MBT, we can also make use of a concept known as *weather type*. This distinction allows us to focus on different aspects of the SUT in a structured way. It is especially useful in our context, as it provides a principled means of separating normal behavior from edge cases and robustness scenarios.

- **Good weather**: Intended system behavior consisting of valid inputs and interactions as described in the specification. Also referred to as the *happy flow*.
- **Bad weather**: Behavior that is specified but undesirable, such as invalid inputs or actions at the wrong time. These cases should not occur under normal conditions but must still be handled.
- **Ugly weather**: Unspecified and deliberately malformed behavior used to probe the system's robustness.

Finally, in addition to the main model(s), it is also possible to create a few small, targeted ones that are not meant to test the system in its entirety but rather to assist in the detection of specific bugs. These auxiliary models are optional and serve as focused tools that can isolate particular behaviors or conditions that may lead to faults.

Figure 3.7 showcases a state diagram that captures the desired high-level behavior of the model.

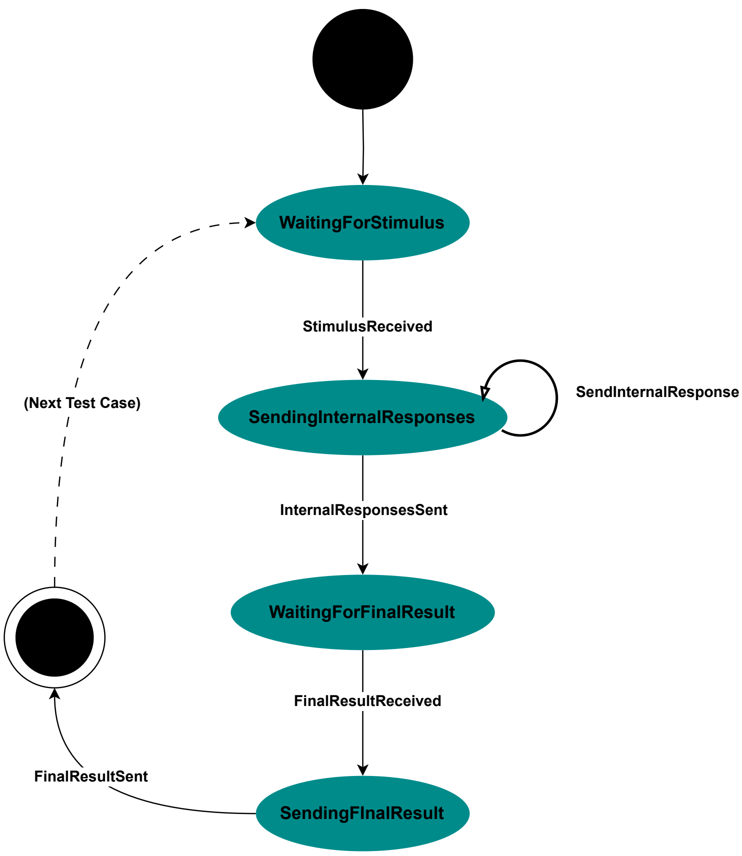


Figure 3.7: Abstract state diagram of the flow during the handling of a single request from the perspective of the model.

Chapter 4

MBT Integration & Implementation

This chapter analyzes the implementation work that brings the previously outlined design into concrete form. We choose to present the components in a top-down manner that reflects the layered architecture of the setup. This structure not only mirrors the flow of execution throughout the testing process, but also reflects the actual way in which the mechanism is developed.

4.1 System-Level Integration

At the foundation of the testing infrastructure lies the ability to observe and interact with the System Under Test in a reliable way. In order for the integration to function as intended, the system must be instrumented to expose its internal behavior and communication patterns. This section describes how this is achieved in the DYNAMOS environment, forming the first layer of the stack and enabling downstream components to access the necessary runtime information.

It has already been discussed that, upon systematic observation, the major components of DYNAMOS communicate with each other primarily through the AMQP protocol, and more specifically via RabbitMQ. Although this messaging layer is abstracted away from each component's core application logic, it remains the fundamental means of inter-service communication. The focus here is on outgoing messages, which are handled by each component's sidecar.

The sidecar implementation is isolated and adequately generic, which makes it an ideal insertion point for testing logic. If the required code is added at this level, we gain an effective means of observing and testing the system under fully realistic conditions, with no mocking or simulation involved. In addition, this modification is non-intrusive and imposes no measurable overhead, as it consists of minimal code placed alongside the existing functionality, without interfering with the normal operation of the component.

This code should act as an observation and forwarding point, capturing the relevant data at runtime and transmitting it to the testing infrastructure. In essence, what we require is a function that, when called with appropriate context, takes care of delivering the message content in a suitable form.

Before proceeding with its implementation, we must determine where this logic should reside. Identifying this point ensures that our addition becomes effective and maintainable. Since we want this logic to be accessible from anywhere within the codebase, the natural choice is to place it inside a library. Looking at the existing structure of DYNAMOS, we observe that there is already a dedicated directory for general-purpose utilities, which contains various reusable functions. By placing it within the corresponding `lib` package, we ensure that it can be invoked from any relevant point simply by supplying the appropriate arguments.

4.1.1 Injected Code

The function responsible for this behavior is defined in the file `mbt_communication.go`, placed inside the existing `lib` package. It is designed to remain lightweight and self-contained. The code in Listing 1 presents the injected logic that is added to DYNAMOS in order to support the testing objectives described above. To keep it concise, logging, error handling, comments, and external utility logic have been omitted.

```go
func SendToTestQueue(ctx context.Context, msgType string, msgBody interface{})
↪   *emptypb.Empty {
    conn, _ := amqp.Dial(getAMQConnectionString())
    defer conn.Close()

    channel, _ := conn.Channel()
    defer channel.Close()

    pbMsg := msgBody.(proto.Message)
    binaryBody, _ := proto.Marshal(pbMsg)

    payload := MbtMessage{
        Type: msgType,
        Body: base64.StdEncoding.EncodeToString(binaryBody),
    }

    bodyBytes, _ := json.Marshal(payload)

    publishing := amqp.Publishing{
        ContentType: "application/json",
        Body:        bodyBytes,
    }

    ctxWithTimeout, cancel := context.WithTimeout(ctx, 8*time.Second)
    defer cancel()

    channel.PublishWithContext(ctxWithTimeout, "", getTestingQueueName(), true,
    ↪   false, publishing)

    return &emptypb.Empty{}
}
```

Listing 1: Core logic for forwarding messages to the testing queue (simplified).

4.1.1.1 Independence from Implementation Language

The `SendToTestQueue` function is implemented in Go, as part of the DYNAMOS codebase. However, the testing integration does not rely on this particular implementation language. What ultimately matters is not the internal logic used to publish messages, but rather the structure and semantics of the transmitted data itself. The testing infrastructure interacts only with the outgoing message as it appears on the queue, and remains entirely unaffected by how this message was generated.

This property makes the integration resilient to future changes. Should the system be rewritten in another language, the injected logic could be reimplemented accordingly without breaking the testing pipeline, as long as the produced messages retain the expected format. This is possible because the interface is defined by Protobufs, a mechanism explicitly designed to be language-agnostic.

4.1.1.2 Establishing the Connection

The first step is to open a connection to the RabbitMQ broker using the existing credentials and create a channel through which the message will be sent. This follows the standard pattern and integrates well with the messaging infrastructure already used in DYNAMOS.

The connection and channel are both closed using `defer` statements to make sure that no resources are held longer than necessary. Although this part is technically straightforward, it is essential to note that without a live channel, no message can reach the testing queue, and thus no observation can take place.

4.1.1.3 Serializing the Payload

The message passed to the function is a populated `MicroserviceCommunication` structure. As discussed earlier in section 3.1, this is the main data container used within DYNAMOS for internal messaging and coordination. It includes structured data, metadata fields, routing information, and various trace elements.

Before the message can be forwarded to the testing infrastructure, it first needs to be converted into a form that can be correctly transmitted and later reconstructed. To achieve this, we serialize the message to binary. This encoding is efficient and preserves the semantics and fields of the original message, as defined in the `.proto` schema. However, binary data cannot be safely included in a JSON payload. It contains non-printable characters and is not always guaranteed to survive transport through the queue.

To address this, we encode the binary output in base64. This is a reversible transformation that converts binary data into plain-text ASCII. Once encoded, the resulting string can be inserted into a JSON object and transmitted across the queue without risk of corruption or loss. This approach ensures that the original message is exactly the same.

To forward the encoded content, we wrap it inside a lightweight structure called `MbtMessage`, which consists of two fields: a `type` string that identifies the nature of the event, and a `body` string that carries the payload. This format provides just enough information for the recipient to interpret the message correctly.

One might ask why the message is not converted directly to JSON, especially since the final wrapper is itself a JSON object. Although libraries offer Protobuf-to-JSON conversions, this route is not suitable in heterogeneous environments, since the JSON representation of a Protobuf message can vary between languages. In contrast, the binary form is always unambiguous, and when combined with a shared schema, it can be decoded by any language.

Encoding the message in this way guarantees that the receiving side can recover the original structure without requiring any knowledge of Go-specific constructs or internal layouts. It only needs the base64 string and the corresponding Protobuf definition, which makes the entire process language-neutral.

4.1.1.4 Sending to the Testing Queue

The final step of the function is to publish the constructed message to the testing queue, allowing the adapter to receive events in a consistent and centralized way. Since the testing mechanism is strictly observational, there is no expectation of a response or acknowledgment. The system continues its normal execution regardless of whether the message is received or not.

The goal of this behavior is to keep the adapter decoupled, so that its presence does not affect how the system operates. For this reason, the message is sent asynchronously in most cases and without blocking the caller. Even if the queue is temporarily unavailable, the rest of the system remains unaffected, reflecting realistic conditions without interference.

4.1.2 Invocation Points

We now shift our focus back to the question of where this function should be invoked within the system. While the sidecar remains the main point of interest due to its architectural position, it turns out that we must also consider a few additional locations where this proves necessary. We describe how these call sites are identified and structured, based on the communication patterns and operational behavior of DYNAMOS.

4.1.2.1 Sidecar Instrumentation

The first step is to determine the central mechanism responsible for publishing messages to RabbitMQ. After analyzing the structure of the component, it becomes clear that the core logic is encapsulated in a helper function named `send`, located in the `rabbit_send.go` file. This function abstracts away the details of AMQP communication and serves as the primary dispatch point for outgoing messages from static components. It is called by various higher-level methods such as `SendRequestApproval` and `SendValidationResponse`, each of which corresponds to a different kind of message.

From a structural standpoint, the `send` function would have been the most direct and centralized location to inject the call to `SendToTestQueue`. However, in practice, we chose to place it directly within the functions that invoke `send`. This allows for better control over which messages are forwarded and enables the insertion of targeted validation logic later in the testing process. As such, the instrumentation logic remains easy to trace, without compromising its coverage.

In addition to these cases, an important exception is identified. The method `SendData`, defined in `grpc_server.go`, is responsible for handling `MicroserviceCommunication` messages that do not pass through the standard AMQP wrappers but are published using a separate path. To ensure that these events are also captured, a second invocation of `SendToTestQueue` is introduced directly into `SendData`. This ensures that no relevant behavior is missed and that all outgoing messages are observable by the testing infrastructure, regardless of their origin.

Each call is wrapped in a lightweight *goroutine* using the below syntax:

```
go lib.SendToTestQueue(ctx, messageType, message)
```

This means that it runs asynchronously in the background using a thread, allowing concurrent execution without blocking the main flow. This makes it possible to preserve the system's responsiveness, even during periods of communication latency.

4.1.2.2 Instrumentation at the API Gateway

We know that DYNAMOS relies on multiple communication protocols internally. This includes not only AMQP for static component messaging but also HTTP for specific flows. One such case, evident in Figure 3.2, involves the API Gateway sending an `SqlDataRequest` to all relevant Agents. This message carries the original SQL query provided by the user and is sent over HTTP rather than RabbitMQ.

To ensure that testing is truly End-to-End, this message must also be captured. Although it falls outside the AMQP flow handled by the sidecar, this does not present a problem. Since the forwarding logic is implemented as a reusable function within the shared `lib` package, it can simply be called from the point within the API Gateway where the request is made. This allows us to extend observability irrespective of the underlying transport mechanism.

4.1.2.3 Instrumentation within the Microservice Chain

A critical part of this thesis lies in capturing the system's adaptive behavior, particularly as it emerges through the dynamic instantiation of microservices within each Agent-specific microservice chain. Initially, it was assumed that these ephemeral services would interact in the same way as their static counterparts, namely by communicating with each other through their sidecar.

However, further investigation reveals that this assumption does not fully hold. In practice, the connection between the Agent and the RabbitMQ queue tied to the corresponding Kubernetes job takes place only at the start and the end of the job. The Agent initiates communication with the sidecar at the beginning of execution and is contacted again only after the particular job's result is produced. The intermediate microservices within the DAG exchange information exclusively via gRPC. This means that messages passed between these components never reach the sidecar directly and thus bypass the AMQP layer.

To address this, we manually invoke the forwarding function from within each service implementation. This is possible for the same reasons discussed earlier regarding the API Gateway. However, there is a crucial difference that makes this case particularly important. When the function is launched asynchronously using a goroutine, it often fails to complete before the microservice terminates. These components are ephemeral by design and are terminated almost immediately after completing their processing task, leaving insufficient time for the forwarding operation to succeed.

To prevent this, we choose to call the function synchronously in this context, removing the `go` keyword and allowing it to block until it completes. This guarantees, under normal conditions, that the testing message is successfully transmitted before the service shuts down. While this change introduces a potential delay in the main execution flow, in practice this overhead has proven negligible across multiple runs and does not disrupt the operation of the job.

The microservices instrumented in this way are those shown in Figure 3.3, namely `query`, `algorithm`, `anonymize`, and `aggregate`. The `graph` step is absent, not because it is unimportant, but because it turns out that it does not correspond to an actual standalone service. It is implemented internally and does not require separate instrumentation.

One exception to the uniformity arises. The `query` service resides in code written in Python, unlike the rest of the system which is in Go. While this poses no significant technical challenge, it does require us to implement a Python version of the forwarding function. This is accomplished using the `pika` library, which offers AMQP support in Python. The new function, named `send_to_test_queue`, mirrors the behavior of its Go counterpart and is invoked synchronously in the same way, thereby maintaining consistency in the way test data is captured across services.

Finally, to avoid confusion between messages sent from within the microservice chain and those exchanged between the Agent and the chain's sidecar, we assign distinct values to the `type` field of each message. Although they all share the same message format (`MicroserviceCommunication`), the semantic context differs. For clarity, we use the naming pattern `serviceNameFinished` to indicate that a particular service has completed its processing and is ready to hand off data to the next step in the chain. For example, the `query` service sends a message of type `queryFinished` to indicate it has produced its final output.

4.2 Adapter Implementation

Having instrumented the system to emit communication events to a dedicated testing queue, we now turn our attention to the core component that receives and processes them. This section describes the deployment and internal implementation of the adapter, which acts as the essential intermediary between the SUT and AMP. Its purpose is to connect the two worlds, translating between abstract test actions and observable system behavior.

4.2.1 Deployment Architecture

As established in chapter 3, the adapter is designed to operate as close as possible to the system under test. Since the DYNAMOS environment is deployed as a Kubernetes cluster, the most natural and consistent way to integrate the adapter is to treat it as a first-class component within the same infrastructure.

To achieve this, the adapter is deployed as a standalone Kubernetes pod that contains a Docker [1] container, built from source using a custom Dockerfile. The container encapsulates the entire runtime and logic of the adapter, which is implemented in Ruby. Much like the existing components of the system, the adapter is equipped with a sidecar that handles internal message forwarding. This uniformity in deployment structure allows the adapter to blend seamlessly into the architecture, both visually and operationally.

The pod is deployed within its own dedicated namespace, reflecting the principles of distributed execution that govern the DYNAMOS design. This decision isolates the adapter from the rest of the system, while still allowing communication through defined interfaces.

For deployment orchestration, DYNAMOS relies on Helm [2], which is the package manager for Kubernetes. The adapter's configuration is defined through Helm charts, using templated manifests. The core logic is located in a template named `mbt-adapter.yaml`.

From a development perspective, treating the adapter as an independent containerized service introduces the challenge that changes to its code require the image to be rebuilt and redeployed in the cluster. This process involves compilation, followed by uninstallation and deployment. While this flow is necessary, it would be cumbersome if done manually after every change. To streamline this, we introduce a helper script named `redeploy_adapter.sh`. This shell script includes the entire sequence, allowing a full adapter refresh with a single command. This minor but meaningful automation greatly improves the efficiency of local development and testing.

The above setup is illustrated in Figure 4.1, where the adapter is shown running inside a pod alongside its corresponding sidecar.

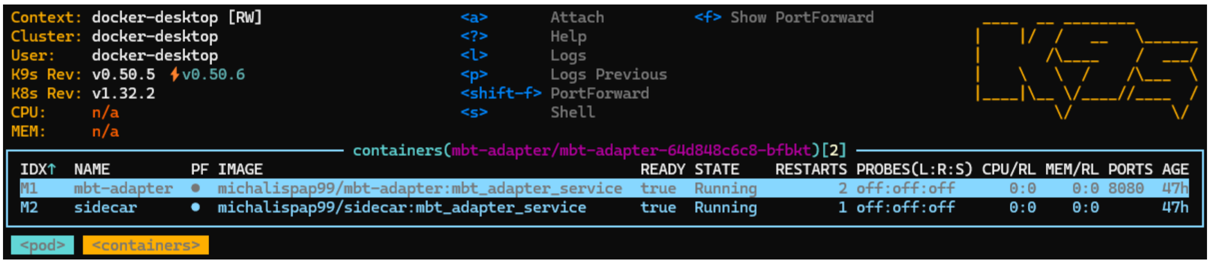


Figure 4.1: Adapter pod view using the `k9s` terminal UI.

[1] https://www.docker.com/

[2] https://helm.sh/

4.2.2 Overview & Structure

The adapter follows the architecture proposed by Axini for plugin adapters, which promotes separation of concerns between generic and domain-specific components. This way, only a small subset of the code is tightly coupled to the SUT, while at the same time the remaining logic is reusable across different implementations.

Figure 4.2 depicts the architecture as described in the official documentation. Communication with AMP takes place over a WebSocket channel, which is defined through Protobufs. The lifecycle of the adapter is driven by the protocol, and each component in the architecture fulfills a distinct role in managing it.

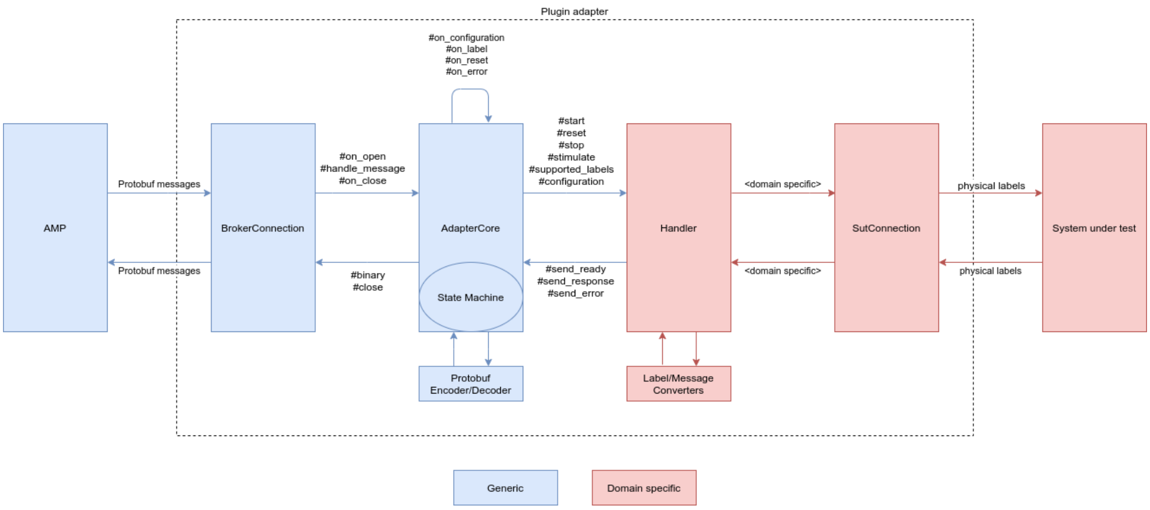


Figure 4.2: Plugin adapter architecture as defined by Axini, showing the separation between generic and domain-specific responsibilities [27].

At a high level, the structure of our Ruby implementation aligns closely with this architecture:

- **BrokerConnection**: Handles the WebSocket communication with AMP. It receives raw protocol messages and calls corresponding callbacks in AdapterCore. It is protocol-agnostic and unaware of message semantics.
- **AdapterCore**: Contains the logic for interpreting protocol messages. It implements the FSM that drives the adapter's lifecycle, from configuration to test execution and shutdown. It is also responsible for encoding and decoding Protobuf messages, and delegates all SUT-specific operations to the Handler.
- **Handler**: Defines the interface through which AdapterCore interacts with the SUT. It translates symbolic labels into concrete actions and collects observations from the system, which are reported back as responses.
- **SutConnection**: Implements the communication mechanism that links the adapter to the system. It deals with concrete technologies, such as RabbitMQ in our case, and reports incoming messages to the Handler.
- **Converters**: Utility components that map between symbolic model-level labels and the physical messages understood by the system. In the current implementation, these are tightly coupled to the Handler logic.

These components interact in the following sequence: BrokerConnection receives messages from AMP, which are passed to AdapterCore for processing. AdapterCore interprets them and assigns actions to Handler, which in turn stimulates the system and listens for responses via SutConnection. Responses are reported back to AMP as soon as they are observed.

In our approach, this separation between components is generally respected, but not enforced strictly. Instead of applying the architecture rigidly, our design is modified to fit the specific characteristics of DYNAMOS. This results in a structure that remains faithful to the original principles but is shaped by practical considerations.

In the following section, we provide a detailed analysis of our implementation, structured around the functional requirements defined in chapter 3.

4.2.3 Protocol Implementation

4.2.3.1 Adapter Lifecycle & Connection Management

Managing the lifecycle of the adapter is essential for achieving repeatable testing. To begin, the adapter must establish a connection to AMP, announce its capabilities, wait for configuration, and then respond with a `ready` signal to indicate it is prepared for test execution. Between test cases, it should reset its internal state and prepare for the next cycle. This process maps to the requirements of **Req-02-Ad: Managing Connections** and **Req-03-Ad: Controlling Lifecycle and State**.

On the one hand, the majority of the above steps is already implemented by the generic components. The system-specific behavior, on the other hand, is implemented in the `DynamosHandler` class. This defines the methods expected by AdapterCore, namely `start`, `reset`, and `stop`. These manage the connection to DYNAMOS via RabbitMQ. When the test session begins, the `start` method is called. It creates a `DynamosConnection` instance and initiates the underlying messaging layer.

Between test cases, the `reset` method is responsible for preparing the system for the next run. If an active connection is present, it sends a special `RESET` message to the system, followed by a `ready` message back to AMP. Otherwise, it reinitializes the connection from scratch.

Finally, the `stop` method cleanly shuts down the RabbitMQ connection and resets the internal state.

4.2.3.2 Label Handling & Message Flow

The mechanisms described in this part address two of our defined requirements. In particular, **Req-01-Ad: Bridging Protocols and Data Formats** and **Req-04-Ad: Message Handling** are central to the adapter's role in converting symbolic labels to concrete system interactions and vice versa.

Stimulus Handling Stimuli are abstract test inputs generated by the model to trigger behavior in the system. When a stimulus is received from AMP, the adapter delivers it to the `stimulate` method of `DynamosHandler`, which determines how it should be processed.

The handler recognizes two stimulus labels for our case: `sql_data_request` and `switch_archetype`. Each corresponds to a different system interaction. The former is the initial client request sent via HTTP POST to the API Gateway, while the latter instructs the Orchestrator to change archetype configuration and is implemented as an HTTP PUT. These cases are shown in Listing 2. The HTTP response handling logic is omitted for clarity and scoping purposes.

```ruby
def stimulate(label)
  api = DynamosApi.new

  sut_message = label_to_sut_message(label)
  @adapter_core.send_stimulus_confirmation(label, sut_message, Time.now)

  case label.label
  when 'sql_data_request'
    api.stimulate_dynamos(sut_message)
  when 'switch_archetype'
    api.switch_archetype(sut_message)
  else
    return
  end
end
```

Listing 2: Stimulus sending logic inside `DynamosHandler` (simplified).

In both cases, the label is first translated into a JSON message using the `label_to_sut_message` method. This function extracts the parameters from the symbolic label and converts them into a structured message expected by the system. Listing 3 shows this logic.

```ruby
def label_to_sut_message(label)
  params_hash = label.parameters.map do |param|
    [param.name, extract_value(param.value)]
  end.to_h
```

```
  case label.label
  when 'sql_data_request'
    sut_top_level_type = params_hash.delete('request_type') || 'sqlDataRequest'
    { type: sut_top_level_type }.merge(params_hash).to_json
  when 'switch_archetype'
    {
      name: 'computeToData',
      computeProvider: 'dataProvider',
      resultRecipient: 'requestor',
      weight: params_hash['weight']
    }.to_json
  else
    { type: label.label }.merge(params_hash).to_json
  end
end
```

Listing 3: Translating a symbolic label to a JSON message.

The function `extract_value` is used to convert individual AMP parameter values into Ruby primitives. It handles all supported types, including strings, booleans, arrays, and nested objects, enabling the adapter to accept a wide range of label formats.

Each request is then passed to `DynamosApi`, which handles the actual HTTP communication.

HTTP Response Handling Responses are outputs produced by the system and delivered back to AMP to reflect the outcome of a stimulus. In our implementation, HTTP response labels are constructed and sent by `DynamosHandler` using a helper method called `send_response_to_amp` (Listing 4).

```
def send_response_to_amp(message)
    return if message == 'RESET_PERFORMED'
    if message.is_a?(Hash) && message['responses'].is_a?(Array)
        message['responses'] = message['responses'].map { |r| r.is_a?(String) ? r :
        ↪  r.to_json }
    end
    send_amp_label("results", message)
end
```

Listing 4: Sending HTTP response to AMP.

This method is invoked after the handler receives and processes a response body from the system, which is the result of an HTTP request. It accepts a structured object, which typically includes a `jobId` and a list of `responses`. Before any label is created, this method ensures that all elements in the response list are properly serialized, converting any nested hashes into JSON strings where necessary.

Once the message is ready, the handler delegates to `send_amp_label`, a more general-purpose method responsible for creating and sending the AMP label. The name of the label is passed explicitly, along with the physical payload and the extracted parameters. The function constructs an instance of a label (`PluginAdapter::Api::Label`) and populates it with fields such as `type`, `label name`, `channel`, and the full list of `parameters`. These parameters are derived using the recursive helper `value_to_label_param`.

The conversion process handles various cases. Scalar values like strings, integers, and booleans are passed through directly. Arrays are converted into ordered parameter lists. Structured objects are translated into Protobuf `struct` types by walking through each key-value pair and applying the appropriate transformation recursively. This allows the adapter to support arbitrarily complex response messages while maintaining type safety.

The completed label is then timestamped and sent back to AMP using the `send_response` method of AdapterCore. This is to guarantee that the response is associated with the correct step in the test case and can be matched against the expected model behavior.

The above procedure allows the adapter to handle heterogeneous output in a systematic and reusable way. It ensures consistency and minimizes duplication.

AMQP Response Handling Unlike HTTP responses which are received synchronously within the same execution flow as the stimulus, AMQP responses arrive asynchronously from the system and must be handled independently. The adapter listens for these messages via the testing queue, which is connected at the beginning of the test session.

The first point of contact is the `handle_incoming_message` method in `RabbitMQService`. This method is triggered whenever a new message is received on the queue. It proceeds by calling the `parse_message` function, which receives the raw JSON message. It begins by extracting the `type` and `body` fields. Then, it effectively performs the sequence of actions defined in Listing 1, but in the reverse order. This entails decoding the base64 string in the `body` field into binary, and, upon mapping the message type to the corresponding Protobuf class, turning every DYNAMOS Protobuf object into a hash using `dynamos_object_to_hash`. Finally, the method returns a collective hash containing the type, payload, and initial raw message. This is illustrated in Listing 5.

```
def parse_message(json_message)
  parsed_message = JSON.parse(json_message)
  type = parsed_message['type']
  base64_body = parsed_message['body']

  proto_binary = Base64.decode64(base64_body)

  klass = klass_from_type(type)
  decoded_proto_obj = klass.decode(proto_binary)
  ruby_payload = dynamos_object_to_hash(decoded_proto_obj)

  { type: type, payload: ruby_payload, raw_json: json_message }
end
```

Listing 5: Parsing an incoming RabbitMQ message (simplified).

The result of this parsing step is forwarded to the `process_rabbitmq_message` method defined in `DynamosHandler`. This is where the actual transformation into an AMP response label occurs. It inspects the message's type and payload, selects the relevant symbolic parameters, and dispatches the label. Listing 6 shows this logic.

```
def process_rabbitmq_message(parsed_data)
  original_type = parsed_data[:type]
  payload = parsed_data[:payload]
  raw_json_body = parsed_data[:raw_json]

  selected_params = extract_amp_params(original_type, payload)

  send_amp_label(original_type, raw_json_body, selected_params)
end
```

Listing 6: Transforming a parsed message into an AMP label (simplified).

4.2.3.3 Error Handling

The adapter must remain responsive and predictable even in the presence of unexpected failures or disconnections. According to **Req-05-Ad: Error Handling**, the system should be able to detect and report faults, avoid crashes, and recover gracefully where possible.

In the current implementation, error reporting is centralized in the `send_error_to_amp` method of `DynamosHandler`. This method is responsible for notifying AMP of any critical failure encountered during test execution. Internally, it delegates to AdapterCore by calling `send_error`, which formats and sends an error message over the WebSocket channel.

This mechanism is used in situations where:

- a message cannot be parsed correctly from RabbitMQ
- required parameters are missing or malformed

- a stimulus cannot be executed due to internal system issues
- a response from the SUT contains invalid or incomplete data

While the generic core ensures delivery and protocol compliance, it is up to the domain-specific handler to detect these issues and respond accordingly. In most cases, the handler simply returns early from processing when an inconsistency is detected. However, in cases where test execution cannot continue, it explicitly triggers `send_error_to_amp` with a descriptive message.

4.2.3.4 Configuration & Supported Labels

The adapter must be able to inform AMP which labels it supports and what configuration values it expects at runtime, allowing dynamic testing. In our case, supported labels are generated programmatically and reflect the full set of stimuli and responses the adapter can handle. Each label is described along with its expected parameters, including types, arrays, and nested objects where needed. This list is returned during the announcement phase and enables AMP to validate whether the adapter is compatible with the current model.

The adapter also exposes a default configuration, which AMP can override during test execution. At present, only a single field is declared, which is the WebSocket URL used to connect to the SUT. This keeps the interface minimal and avoids overcomplicating the configuration unless necessary. Internally, the configuration system is extensible and could easily support additional runtime fields if required.

At this point, it is worth highlighting a pivotal observation for our study, that, while seemingly a natural choice, is not obvious. There are multiple ways to *map* the individual fields of each message received from the system into the adapter, so that they can be forwarded appropriately to the testing platform. Given this degree of freedom, we make the deliberate decision to map these fields *directly* to the label parameters used by the corresponding responses. This choice guarantees both accuracy and faithful representation of the internal communication between the system's components within the context of our testing methodology. Furthermore, it makes the resulting labels easier to understand in relation to the system specification, which would not be the case if we either omitted these parameters entirely or altered them significantly. In practice, it is sometimes in our interest to preserve and convert only a subset of them, since many are repetitive and introduce unnecessary clutter. This selective transformation does not compromise the correctness or generality of our approach in any way.

The full adapter implementation can be found at: `https://github.com/michalispap/MBT-Adapter`
The corresponding DYNAMOS repository can be found at: `https://github.com/michalispap/DYNAMOS`

4.3 Model Construction

Having completed the testing integration mechanism, we can now focus on the modeling approach. The following section presents the structure, content, and rationale behind the developed models, with particular emphasis on the decisions that enable us to represent distributed actions and adaptivity.

4.3.1 Platform Support for Modeling Requirements

The suitability of the Axini Modeling Platform for our approach can be evaluated by revisiting the requirements defined in chapter 3. These describe the core capabilities for effective modeling.

Regarding **Req-01-Md: Adapter and Label Registration**, which concerns adapter and label registration, AMP automatically detects the adapter and retrieves the complete set of supported labels and parameters. This enables smooth synchronization between the model and the interface exposed by the system.

With respect to **Req-02-Md: Configurable Test Execution**, the platform provides an interface that allows users to configure key test execution parameters.

As for **Req-03-Md: Label Parameter and Path Variability**, AMP supports parameter variation through random number generation, predefined sets, and constraint solving. Execution paths within the model can branch dynamically depending on input values or internal state, making it possible to explore alternative behaviors and achieve full coverage.

AMP also fulfills **Req-04-Md: Support for Structured Logic and Abstractions** by offering structured logic with support for variables, reusable macros, behavior definitions, and more. This allows common sequences to be abstracted and reused, keeping models concise and maintainable.

Finally, **Req-05-Md: Abstraction of Distributed Systems** is addressed through the platform's inherent ability to function at the communication layer, facilitating the modeling of interactions between distributed entities at the suitable level.

4.3.2 Modeling Overview

The models developed in this study are designed to capture the system-level behavior of DYNAMOS as observed through external and internal interactions. In line with the testing approach defined initially, the modeling scope is End-to-End. Each model specifies how the system responds to incoming stimuli through observable outputs, without assuming knowledge of the internal implementation of the services themselves. The primary focus is on capturing functional behavior as it occurs over time, as illustrated in Figure 4.3.

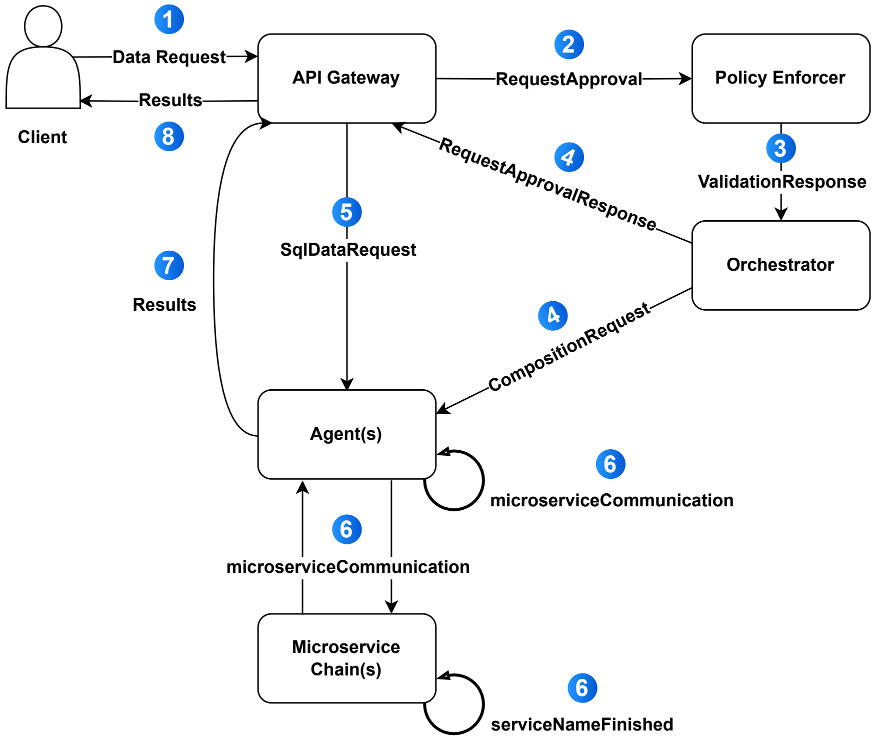


Figure 4.3: Happy flow of a client request through the SUT.

A core challenge in modeling DYNAMOS lies in its adaptive nature. The system modifies its internal behavior depending on contextual information and formal agreements, which are expressed through archetypes. Archetypes determine which Agents' internal chains of services are deployed and how. As such, they constitute the backbone of the system's adaptive logic and provide a natural boundary for modeling. The two principal models developed correspond to the two archetypes implemented in DYNAMOS, namely *computeToData* and *dataThroughTtp*. Each of these models captures the behavior that results when its respective archetype is activated. A third model combines these two under a unified interface in order to enable exploratory testing across multiple flows. Other supporting models are also present and discussed separately.

Internally, the structure of each model makes a clear distinction between the HTTP-level interaction that initiates the flow and the AMQP-based message exchange that follows. This separation mirrors the layered architecture of DYNAMOS and provides a boundary between request initiation and distributed execution. This is not only conceptually appropriate but also practical, allowing the two parts of the model to be evaluated concurrently during testing.

The System Under Test follows a communication pattern where a single stimulus triggers a cascade of responses. These responses may originate from different components and follow varying timing patterns, depending on the selected archetype and conditions. This makes the question of *when* a message is expected to occur just as significant as *what* type of message it is. The model must therefore express behavior along two complementary axes. The first is the logical constraints that define which responses are valid, and the second is the temporal structure that determines when each response should appear. In this context, modeling behavior becomes a matter of specifying what may happen and when.

Lastly, while the majority of the modeling effort focuses on the happy flow, the models also include selected elements of bad weather behavior. These are used to validate the system's handling of incorrect actions. However, since our models are primarily oriented towards capturing responses, it is reasonable that both bad and ugly weather behavior are restricted to stimuli. According to the official documentation, applying these weather types to response messages is not meaningful.

4.3.3 Shared Elements

Before exploring the core models, it is necessary to introduce several elements that define the environment. These include components that are common to all models, as well as configuration mechanisms and control variables that influence behavior globally.

The first such element is the `interface.aml` file. This module declares the external labels of the model. Each label can be associated with parameters, constraints, and behavior. It therefore defines the available vocabulary of the model and allows other files to use these labels without redefining them.

A second important mechanism is configuration. Each model includes the shared `default_config` file and may override selected values by setting entries in the `@config` map. These configuration variables are used throughout the model to control conditional logic, activate specific paths, or tune parameter values. Examples include the selection of the active archetype, the enabling of optional services, and the toggling of specific weather types. By centralizing these definitions, the model remains flexible and can be adjusted without structural modification.

One variable that appears frequently is `timeout`, which is typically set to a value of 20. This setting determines the maximum number of seconds a process will wait for a response before declaring quiescence. The presence of a timeout allows the model to express not only what responses are expected but also how long the system is allowed to remain silent before that absence becomes meaningful. In our case, setting it to this particular value is an empirical decision affected mainly by the specified tolerance.

In addition to the above components, several other shared modules are included, containing auxiliary logic used throughout the modeling effort. Among them are files defining reusable macros, internal communication channels, domain-specific functions, behavior fragments, and state variables.

4.3.4 The HTTP Process

We begin our analysis by examining the HTTP-level interaction that commences each execution flow. This logic is contained in the `dynamos_http_flow` process, which is included at the top level of both archetype-specific models. In AML, a *process* describes the external behavior of the system, and an *external channel* represents an interface of the SUT to its environment. Each process is evaluated independently during testing and may interact with other processes.

The `dynamos_http_flow` process models the initial client request received by the system, the selection of the archetype, and the system's respective replies. It runs in parallel with the AMQP-based process that handles internal message coordination. The separation between the two is intentional and maps directly to the system's architecture, where user-facing actions are handled via HTTP, and backend coordination is performed via RabbitMQ.

The process declares an *internal channel* called `sync_channel`, which is used to synchronize with the second process, `dynamos_rabbitmq_flow`, during execution. Internal channels do not exchange externally visible messages. Rather, they allow one process to notify another using labels that are not exposed to the adapter. They reflect the fact that there is also unobservable communication taking place between separate components within the system (`tau-step`). In our case, the `sync_channel` enables the HTTP flow to trigger downstream logic by sending a signal after a specific stimulus has been received.

Client Request Handling

The flow begins with the reception of an HTTP PUT request labeled `switch_archetype`. This stimulus is issued by the client to indicate the archetype that should be activated for the current execution. A constraint ensures that the value of the `weight` parameter matches the archetype currently selected in the configuration. If the constraint holds, the model replies with a `http_response_status` message, signaling that the archetype has been accepted.

In addition to this main path, the model includes an optional branch that expresses bad weather behavior. When enabled via configuration, this branch activates a conditional block that detects an invalid archetype weight. In such cases, the model may send an immediate `http_response_status` response to indicate that the selection was not accepted. This behavior serves to verify how the system reacts to incorrect input values, while remaining isolated from the standard flow.

Once the archetype has been accepted, the model proceeds to handle the main request. This is implemented through a call to the *macro* `perform_sql_data_request`, which receives the HTTP POST stimulus labeled `sql_data_request`. A macro allows us to leverage Ruby functions to declare transitions before the test, effectively using the language as a preprocessor. Therefore, this request carries the parameters of the client's query and marks the actual start of the processing phase. In practice, it performs the data analyst's HTTP call with a JSON body to DYNAMOS, initially reaching the API Gateway. These parameters are depicted in Table 4.1:

Parameter	**Type**
`request_type`	string
`user[id]`	string
`user[userName]`	string
`dataProviders`	list of strings
`data_request[type]`	string
`data_request[query]`	string
`data_request[algorithm]`	string
`data_request[options][graph]`	boolean
`data_request[options][aggregate]`	boolean
`data_request[options][anonymize]`	boolean

Table 4.1: Fields in the JSON body of the HTTP POST request.

These fields can theoretically be populated with any value allowed by their specified type. In practice, due to limitations in the current version of DYNAMOS that we use, we hardcode a small subset of those values, thus ensuring both comprehensive testing by leveraging AMP's solving capabilities and realistic conditions. Consequently, we only allow AMP to choose between one or more representative examples of data providers (`['VU','UVA','RUG']` or `['UVA']`), and whether to select anonymization by setting the corresponding flag to `true`. The rest of the values are either fixed or overridden by the archetype-specific models. For example, the chosen algorithm is `average` and is used everywhere.

Upon reception, the model sends a response labeled `http_request_received` on the `sync_channel`, which acts as a trigger for the internal logic managed by the `dynamos_rabbitmq_flow` process.

Asynchronous Outcome Handling

At this point, the model performs a branching operation using what is arguably the most important construct that AMP offers for representing asynchronous behavior, namely the `choice` construct. This mechanism allows the model to accept more than one possible behavior, whether in the form of a stimulus or a response. In this case, it is used to include both good and bad weather behavior within the same execution flow, assuming the latter has been enabled via configuration.

The logic that follows is critical to the remainder of the execution. In the successful case, the model expects to receive, within the configured timeout, a stimulus from the `dynamos_rabbitmq_flow` process. This message indicates that processing has completed and includes a parameter specifying the number of responses expected from the active DYNAMOS Agents. This number is stored and later validated to ensure that the system returns the correct number of responses to the client, along with a success HTTP status code, before terminating.

In the alternative case, where bad weather behavior is being tested, the model allows the dispatch of an error-related response. This reflects a scenario in which processing has failed or no valid data could be retrieved. The use of the `choice` construct enables the model to remain agnostic as to which outcome arrives first until runtime. Finally, if bad weather is not enabled, then the first option in the choice block becomes effectively singular and deterministic. Listing 7 illustrates the aforementioned behavior.

```
send 'http_request_received', on: 'sync_channel'

choice {

    o {
      receive 'expectations_ready', on: 'sync_channel', update:
       ↪  'expected_responses_from_sync = _count'
      send 'http_response_status'
      send 'results', constraint: 'length(responses) ==
       ↪  expected_responses_from_sync', update: 'observed_responses =
       ↪  length(responses)'
    }
    if bad_weather?
      o {
        send 'http_response_status'
        send 'results', constraint: 'is_empty(responses) || include(responses, "")'
      }
    end

}
```

Listing 7: Conditional flow after HTTP request based on execution outcome (simplified).

4.3.5 Core Models

The primary models constructed for this study correspond to the two archetypes defined in DYNAMOS. Each of these models represents the complete behavior of the system under the assumption that the respective archetype has been selected and activated. We first provide an analysis of the `computeToData` model, examining its structure and logic. Subsequently, we highlight the points of divergence in the `dataThroughTtp` model. Finally, we introduce a third complementary model, `combinedArchetypes`, which merges both archetypes under a unified abstraction.

It is vital to clarify again that DYNAMOS does not fall under the typical category of a reactive system that continuously interacts with its external environment. As a result, although it may seem counter-intuitive, the majority of messages are not directed towards the outside world, that is, the user, but remain internal to the system. This may initially hinder a clear understanding of the model's behavior. However, once this distinction is made explicit, the rationale behind the modeling approach becomes significantly easier to grasp.

4.3.5.1 `computeToData` Model

The `computeToData` model captures the behavior of the system when a single Agent assumes both the data and computation roles. It is the most straightforward of the two models and serves as the baseline from which the more distributed case is derived. The model consists of the shared HTTP process already described and a second process, `dynamos_rabbitmq_flow`, which orchestrates the internal coordination via AMQP. This is defined locally in the `computeToData.aml` file and expresses the complete logic of both static and ephemeral microservice execution.

Initial Execution Phase The execution begins with a call to a shared *behavior*, an efficient means of grouping related transitions, called `receive_and_approve_request`. This behavior coordinates the initial message exchange between the static components of the system. It first receives the internal label `http_request_received` from the HTTP process via the `sync_channel`, indicating that the client request has been accepted. It then proceeds to send a message `requestApproval` from the API Gateway towards the Policy Enforcer, reflecting the forwarding of the user's request for validation.

Upon internal processing, which is only evident to us through the passage of time, the model sends the label `validationResponse`, which contains information about whether the request has been approved and which data providers from the ones supplied by the user have been deemed valid. This is the message exchange between the Policy Enforcer and the Orchestrator. The parameters are extracted into variables such as `validated_providers` and `approved`, which are subsequently used throughout the model. At this stage, the request has been formally accepted, and the system is ready to initiate the selected execution plan. It is important to note that, in order for `validationResponse` to be accepted by the model, both outcomes of two helper *functions* must be positive at the same time, namely `check_valid_data_providers` and `check_invalid_data_providers`. Functions assist in processing runtime AML data objects. Listing 8 shows an example of one of them.

```
function('check_valid_data_providers', [[:string], [:string]] => :boolean) do
↪  |valid_dataproviders, all_data_providers|
  (valid_dataproviders - all_data_providers).empty?
end
```

Listing 8: Ensuring all valid data providers are present in full list.

Allowing Different Messages Following the successful completion of this phase, our goal is to model the internal message exchanges that take place afterwards. Since information processing occurs concurrently and without predetermined sequencing, we require a modeling mechanism that allows for the repeated dispatch of correct and expected messages, regardless of their order, while still enforcing strict behavioral constraints derived from the DYNAMOS specification. This construct must introduce a form of synchronization and unify the handling of sent messages, whether they originate from static or ephemeral components.

To achieve this, we opt to use the `repeat` construct provided by AML. This is functionally similar to the previously discussed `choice`, with the key distinction that it defines a repetition loop. As such, it must be explicitly instructed when to terminate, through a dedicated condition imposed at the appropriate location in the model.

The first possible message that the system is allowed to send is the `compositionRequest`, which is dispatched by the Orchestrator to all validated Agents. Its parameters specify the selected archetype, the recipient Agent, and the role assigned to that Agent. In order for this response to be emitted, the archetype and role must be valid, and the number of `compositionRequest` messages sent up to that point must not exceed the number of validated providers. The final condition is particularly significant, as it ensures that if the system attempts to exchange additional messages of the same type, this constitutes a clear violation of the specification and, consequently, of the expected behavior. The model is designed to treat such cases as failures. This logic of bounded message emission is enforced consistently for every subsequent response, in order to guarantee that the number of responses remains fully correct.

The second possible message that can be sent asynchronously by the Orchestrator is the forwarding of the Policy Enforcer's reply, namely the `requestApprovalResponse`, which must occur exactly once per client request. As established earlier, this message is directed towards the API Gateway, and it is handled in conjunction with the `compositionRequest` using a shared behavior that groups them under a single `choice` construct.

Next, one expected message is the `sqlDataRequest`, which represents the HTTP request sent by the API Gateway to the participating Agents and carries the SQL query as its payload. Although this is not explicitly a RabbitMQ message, it is nonetheless modeled as such. This choice is motivated by the fact that the request is executed internally within the system and, as previously explained, is forwarded to AMP via the RabbitMQ testing queue.

Another potential response is the `microserviceCommunication`, which reflects the interaction between a given Agent and its corresponding Microservice Chain, or vice versa. Here, an important distinction must be made. If the `result` field in the message parameters is empty, the message represents the initial dispatch from the Agent to trigger the execution of the job. If, on the other hand, the field contains data, and specifically the result of the query, then it constitutes the final response from the chain to the Agent, after which the ephemeral component is terminated.

In both cases, the model enforces a constraint ensuring that only one such message is accepted per Agent. This guarantees that duplicate messages of the same type cannot be received from the same source.

Modeling Adaptivity of Ephemeral Components The next choice is an important one for this thesis, as previously discussed, since adaptivity is not only related to the archetype, but also chain composition, as it is specified at runtime. Therefore, after the microservice chain has been initialized, the model must observe and evaluate what happens within it in terms of processing microservices. To this end, we define a dedicated behavior in which, at least for this archetype, the model must receive a specific sequence of symbolic messages in the prescribed order shown in Figure 3.3. Any deviation from this sequence constitutes a discrepancy.

First, the model expects to receive `queryFinished` from the first service in the DAG. Next, if the user has requested anonymization, the model waits for `anonymizeFinished`. In this case, the `optionally` construct is used, which is particularly useful when waiting for a response that may or may not occur. Third, the model expects `algorithmFinished`, which signifies the end of the processing phase.

A separate instance of this chain must be observed per Agent. It is important to recall that, despite these being gRPC messages, they are modeled within this process for the same reason as the HTTP `sqlDataRequest` discussed earlier. Listing 9 illustrates the corresponding behavior.

```
behavior('send_chain_microservice_finished') {
send 'queryFinished', constraint: 'query_count < length(validated_providers)',
↪  update: 'query_count += 1'
optionally { send 'anonymizeFinished', constraint: 'is_anonymize_enabled &&
↪  anonymize_count < length(validated_providers)', update: 'anonymize_count += 1'
send 'algorithmFinished', constraint: 'algorithm_count <
↪  length(validated_providers)', update: 'algorithm_count += 1'
}
```

Listing 9: Notifying upon chain microservice processing termination (simplified).

Validation Phase The final step of this process is the termination condition. The model checks whether all expected internal actions have been completed. This is achieved through a single symbolic transition guarded by a comprehensive constraint that consolidates the key properties of a successful execution. The model verifies that the number of `compositionRequest` messages sent equals the number of validated providers, that exactly one `requestApprovalResponse` has been issued, and that each provider has responded precisely once. It also confirms that the Agents involved match the set of validated providers and that no error has been recorded during execution.

Moreover, it ensures that the count of completed chain microservices aligns with the expected values. If all these conditions are satisfied, the loop is terminated by invoking `stop_repetition`. At this point, the model updates the variable `expected_number_of_responses` and calls the behavior `send_finished_to_http`, which returns control to the `dynamos_http_flow` process.

By the end of this process, all relevant information has been extracted, processed, and delivered in accordance with the active archetype and the parameters of the original request. The model ensures that each internal action has been validated, each expected response has been emitted, and that the system reaches a well-defined termination point. Despite not enforcing strict rules at every single point of execution, it provides a clear and verifiable structure, allowing for precise observation and testing of the desired behavior.

4.3.5.2 `dataThroughTtp` Model

The second model, `dataThroughTtp`, describes the system's behavior when the roles of data provisioning and computation are distributed between distinct Agents. Structurally, it reuses the same HTTP-level process and macro definitions as the `computeToData` model. As such, the main point of divergence lies in the `dynamos_rabbitmq_flow` process, which is redefined locally in the `dataThroughTtp.aml` file to capture the more distributed and multi-actor execution pattern.

The orchestration logic in this model builds upon the same conceptual foundation, using the `repeat` construct to allow for asynchronous internal responses. However, it introduces several changes that reflect the complexity of coordinating multiple cooperating Agents.

To begin with, it should be noted that this archetype introduces an additional chain service, namely `aggregate`. As a reminder, the `query` and `anonymize` services are handled by the data provider, while `algorithm` and `aggregate` are executed by the compute provider, which in this case is the independent third party. This is precisely why the corresponding behavior differs. Although it remains structurally similar to the one used in `computeToData`, the order of the messages is not enforced. This is because coordination messages are exchanged between Agents in between the microservice completions. As a result, the model uses the `choice` construct and introduces the `aggregateFinished` response, which indicates that the answers have been successfully aggregated by the compute provider, which in our case, is SURF.

Continuing with the differences, the model also performs a check on the number of `sqlDataRequest` messages, which must match the number of Agents, since the query is sent to all of them. What is particularly interesting in this context are the `microserviceCommunication` messages. When the `result` field is empty, it can reflect several different scenarios. If the message is associated with a data provider, it may indicate that the provider is initiating communication with the Microservice Chain as before, or that the third party is requesting the data from them, or even that the data provider is sending back its own result. These intermediate payloads and results are not visible in the label parameters, as they reside in fields that are not explicitly modeled, primarily because they are used only rarely and would clutter the model. If, on the other hand, the message is associated with a compute provider, then it serves as an acknowledgment of the data received per Agent.

Another difference lies in the termination condition. The model now expects, among other minor differences, one additional `compositionRequest` to be sent, as this message is addressed to the compute provider. Finally, at the end of the process, the number of expected responses to be sent internally to the `dynamos_http_flow` process is equal to the number of compute providers, which in this case is one. This is because the third-party Agent is responsible for sending a single, aggregated response.

The diagram in Figure 4.4 provides a graphical representation of the `dataThroughTtp` model's Symbolic Transition System as rendered by AMP. It is nearly identical to that of the `computeToData` model.

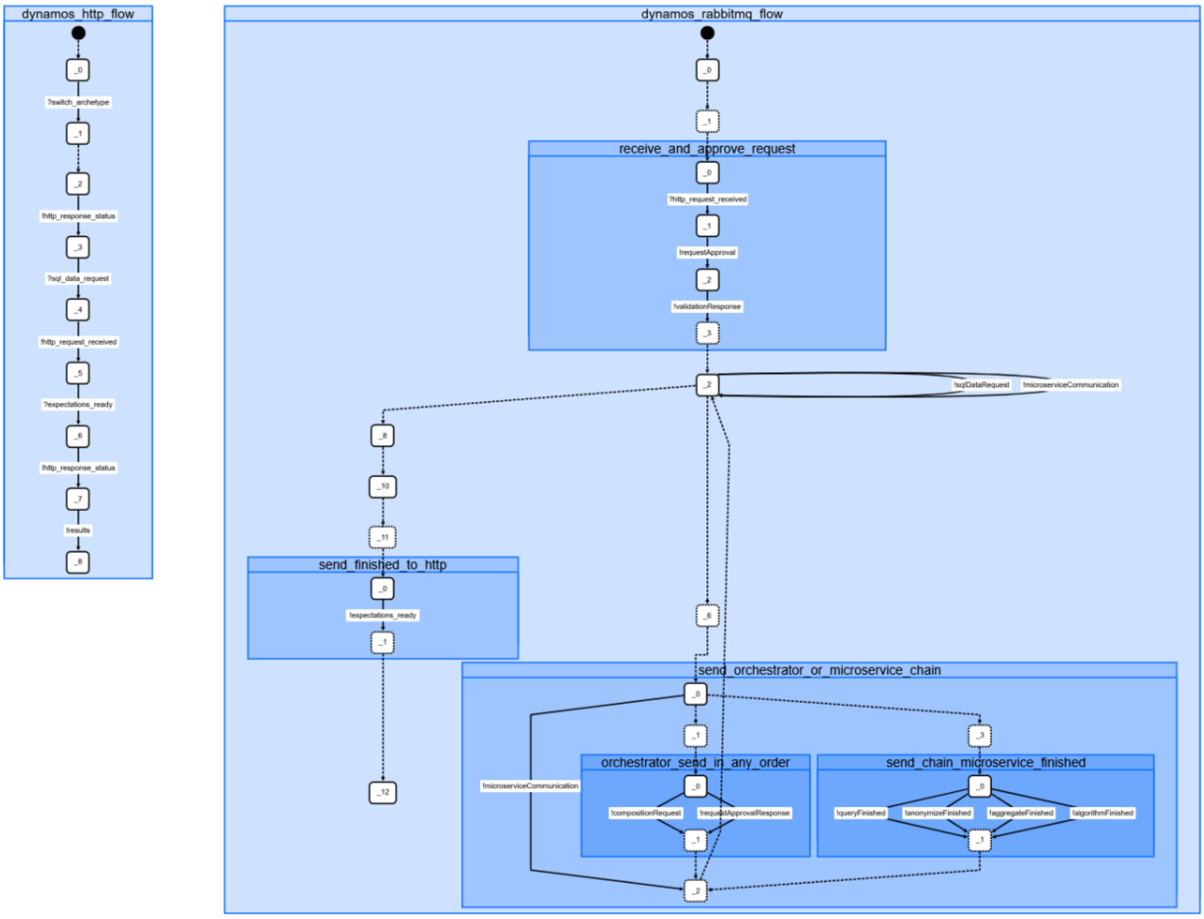


Figure 4.4: The dataThroughTtp model STS.

4.3.5.3 `combinedArchetypes` Model

The `combinedArchetypes` model serves as a complementary construction that merges the behavior of both core archetypes under a unified abstraction. Unlike the two previous models, which assume that the archetype is statically defined in the configuration, this one is designed to demonstrate that archetype selection can also occur dynamically within a single execution. In doing so, it provides evidence that adaptive switching between archetypes is not only feasible but expressible at the symbolic level within

AML.

The original DYNAMOS work [26] states that if the system's load exceeds a certain threshold, specifically 80 percent, an appropriate archetype is selected to better balance execution. As such, system load could be a factor influencing adaptive behavior. However, since our study focuses on MBT rather than, for example, load testing, the only way to verify whether this mechanism exists and functions correctly in practice would be to issue a high volume of parallel requests. Given that this is not part of the standard testing approach, we instead choose to reflect this selection directly within the model itself.

Structurally, the model reuses both the HTTP and RabbitMQ processes but generalizes several behaviors to accommodate variation in the execution path. For example, it includes transitions that cover both the `computeToData` and `dataThroughTtp` sequences, with appropriate constraints to ensure that only one path is taken at runtime, based on the received archetype. This is achieved through relevant parameters and conditional logic that route the flow accordingly.

While not intended to be as precise as the two dedicated models, `combinedArchetypes` plays a valuable role in exploratory testing. It allows for broader test scenarios and configuration diversity, all within a single structure. Moreover, it provides a useful abstraction when adaptivity itself is the subject under test, rather than any specific behavior associated with a single archetype.

Nonetheless, the model trades some of the strictness of the dedicated variants in favor of flexibility. Certain constraints and counters may be defined more loosely, and the termination condition is relaxed to accommodate different paths through the system. As such, it is not meant to replace the more rigorous models, but rather to complement them by enabling experimentation.

4.3.5.4 Scalability of the Modeling Method

The modeling approach adopted in this work scales naturally as new archetypes are added to the system. Each archetype can be represented by a separate model, in the same way as the existing `computeToData` and `dataThroughTtp` variants. These models reuse the shared `dynamos_http_flow` process, while implementing a custom `dynamos_rabbitmq_flow` process to capture their internal behavior. In practice, much of the structure and logic is borrowed from existing models, with only localized adjustments to specific behaviors and configuration options.

When integrating a new archetype into the `combinedArchetypes` model, the changes required are also minimal, primarily involving configuration handling and the definition of constraints or conditions that determine the correct path. This remains true as long as the added archetype does not introduce fundamentally different execution logic. In cases where more substantial divergence occurs, the number of modifications naturally increases, but the underlying modeling strategy remains consistent and tractable.

4.3.6 Supporting Models

In addition to the primary models that capture the expected behavior of the system, a set of supporting models is developed to assist with error diagnosis, behavioral observation, and focused scenario testing. These models do not aim to provide full coverage of the system's functionality, but rather complement the main test process by isolating and controlling specific cases.

Following the methodology outlined in the official documentation, we distinguish between three categories of models. The working model, referred to as M_{work}, is designed to express only conforming behavior as defined by the system's specification. In contrast, the bugs model M_{bugs} contains targeted traces that intentionally reproduce known violations, thereby making it possible to confirm and isolate faulty executions. Finally, the witness model M_{witness} records the observed behavior of the system when it deviates from the specification, enabling a better understanding of how the system behaves in non-conforming states.

This separation allows each model to serve a distinct purpose in the testing workflow. When a test case fails, we can use M_{bugs} to reproduce and isolate the faulty behavior, and M_{witness} to understand how the system actually behaved instead. Meanwhile, any test that passes under M_{work} can be trusted as a valid confirmation of the specification. This approach simplifies analysis, increases confidence in the test verdicts, and supports clearer reasoning when retesting system updates.

4.3.6.1 `bugs` Model

M_{bugs} is designed to fail. It is defined as a single process with one unified behavior, `bug_options`, which contains several mutually exclusive traces that simulate invalid client requests. Each option alters a specific part of the input payload, such as an empty or malformed `request_type`, an invalid SQL query, or a configuration where the `graph` option is enabled, which is an identified bug, and observes whether the system proceeds to generate non-empty results regardless.

These deviations are implemented as separate branches within a `choice` construct, each using a modified version of the `sql_data_request` constraint. Following the reception of one such input, the model enters a repetition block that sends the full expected sequence of internal messages. The presence of a final `results` response, combined with a successful HTTP status code and a non-empty payload, is the desired outcome that is going to be violated. This is explicitly encoded in the final constraint, which checks that the array of responses is not empty and contains no empty strings.

The goal is to verify that incorrect or malformed requests do not silently pass through the system and produce output. Each path is short, focused, and designed to fail, ensuring that these bugs are both observable and reproducible. The model thus acts as a fault probe, testing how the system reacts to a variety of structurally invalid inputs while maintaining an otherwise valid execution path.

4.3.6.2 `witness` Model

Conversely, M_{witness} is designed to succeed. Every trace must pass, notwithstanding the fact that the behavior does not align with the specification.

All bugs are defined within a single behavior, `witness_options`, using the same `choice` construct and options as in M_{bugs}.

What distinguishes this model is the constraint on the final `results` message. Here, the model explicitly allows the response to contain only empty strings, using the function `all_empty_strings`. This reflects a situation where the system processes the request and replies with an HTTP 200 status, yet returns a response that is semantically void. While this does not violate the syntax of the protocol, it reveals a behavior that diverges from the expected.

By recording these defects in a separate passing model, we preserve information about the system's real-world response to edge cases, even if they are not desirable. This helps detect whether the same behavior persists across versions, without interfering with the correctness claims made by the main model.

4.3.6.3 Scenarios

A final group of supporting models are `scenarios`. These consist of short blocks that assert or reject specific use cases within a test execution trace. Unlike the other models, they are a subset of allowed behavior and do not define new processes but instead refer to transitions that should or should not occur. Each one focuses on a single interaction, such as a particular label being sent or a transition happening anywhere in the trace. They are useful both for fine-grained validation and for ensuring that certain requirements are being exercised during test runs.

In our case, the defined scenarios primarily capture high-level behavioral expectations tied to the successful execution of key components in each archetype. These include the presence of specific response patterns and the verification of minimal delivery guarantees. While limited in number, they serve as supplementary formal confirmation.

For reference, the primary AML models used in this study are included in Appendix A.

Chapter 5

Case Study Experiments

The purpose of this chapter is to demonstrate the methodology via which we aim to evaluate the functionality and effectiveness of the proposed testing approach. Overall, we intend to verify that the integration mechanism operates correctly and that the developed models are both executable and representative of the system's behavior. We also aim to investigate the extent to which this approach can detect deviations from expected behavior. The experiments are designed to systematically cover different execution flows and to assess how well the MBT setup performs across both valid and erroneous configurations.

5.1 Experimental Setup

5.1.1 System Configuration

All experiments are executed on an ASUS Vivobook S16 laptop equipped with an AMD Ryzen 9 8945HS processor and 16 GB of RAM. DYNAMOS runs inside a Docker Desktop environment with Kubernetes enabled through Windows Subsystem for Linux (WSL) 2 [1], using Ubuntu 24.04 as the underlying Linux distribution. The container runtime is allocated 8 CPU cores and approximately 8 GB of memory. The Kubernetes cluster consists of a single node, running all static and ephemeral components of the system. The adapter is deployed as a regular pod in the same cluster and communicates with the AMP via a dedicated WebSocket channel.

5.1.2 Test Campaign Configuration

Test runs are launched through the AMP web interface. Each model is executed independently with its own configuration profile, and results are collected via the reporting interface.

To achieve consistency and allow variation, a set of configuration parameters is defined and applied uniformly across all test runs. These include both structural and behavioral options that influence the way test cases are generated and executed. The key settings used during the experiments are as follows:

- **Strategy:** `GlobalTransitionCoverage`. This strategy aims to cover all transitions across the entire model.
- **Maximum number of steps:** 30 steps per test case. This upper bound limits the depth of each individual trace.
- **Trailing responses wait time:** 5.0 seconds. This parameter defines the time window for receiving any additional stimuli or responses after a test case concludes.
- **Quiescence wait time:** 5.0 seconds. The tester waits this amount of time at the end of a test case to detect whether the system enters a quiescent state.
- **Randomize paths:** `true`. This setting introduces variation into the selection of unexplored transitions, allowing the generation of diverse test cases across different runs.
- **Solver:** GNU Prolog [2]. This constraint solver is used to evaluate guard conditions and determine valid data values during symbolic execution.

In addition, as explained in section 4.3, we implement custom configuration to control the activation

[1] `https://learn.microsoft.com/en-us/windows/wsl/about`
[2] `http://www.gprolog.org/`

of behavioral paths within the model using dedicated flags. These flags selectively enable or disable functionality such as good and bad weather flows, as well as optional microservice behavior. Combined with the symbolic parameters in the models, this setup allows the assessment of different runtime scenarios in a controlled and reproducible way.

5.2 Evaluation Strategy

The evaluation is carried out in two phases. In the first phase, we assess whether the testing setup functions as intended under different configurations, and whether the constructed models are able to capture the behavior of the SUT. In the second phase, we introduce deliberate faults into the system and evaluate the extent to which the testing infrastructure is able to detect and isolate them.

5.2.1 Execution Scenarios

Each core model is executed independently in test runs consisting of a maximum of *three* test cases. We verify that each model is executable and stable, and that the system responds with observable and conformant behavior. This ensures that the adapter, modeling platform, and system under test interact as expected and that DYNAMOS adheres to its requirements. In addition to normal operation, we activate bad weather behavior to assess how the system handles incorrect cases. We deliberately cause this behavior by alternating the `aggregate` flag to the opposite value per archetype.

To support evaluation, we leverage the auxiliary models defined in section 4.3, namely the `bugs` and `witness` models. The reason we have already identified defects and populated these models stems from our interaction with the system during the modeling phase. This is not only expected, but in fact fully aligned with the goals of model-based testing, particularly its emphasis on discovering bugs early, during the construction of the models rather than after their completion. Naturally, these supporting models can be effortlessly extended to incorporate additional faults as they are discovered. At this stage, we have identified and captured *four* specific bugs that are included in our evaluation (see Table 5.1). We also assess the outcome of `scenarios` to assert the presence or absence of individual requirements across test runs.

Finally, we observe the transition and state coverage of each model, which are mainly affected by the inclusion of optional services, use of varied data values, and weather selection.

Bug Condition	**Description**
`data_request[options][graph] == true`	A request is issued with graph option enabled.
`request_type == ""`	An empty string is supplied as the request type.
`request_type == "invalidString"`	An invalid string is supplied as the request type.
`data_request[query] == "invalidString"`	An invalid string is supplied as the SQL query.

Table 5.1: Bug scenarios defined in the `bugs` model.

5.2.2 Evaluation Under Fault Injection

To further examine the fault detection capabilities of the testing framework, we conduct a second round of experiments in which the SUT is modified to include a set of known defects. This allows us to assess not only whether the models are expressive enough to describe conforming behavior, but also whether they can expose subtle deviations from it.

Initially, we explored the possibility of using mutation testing tools, including `go-mutesting` [3], `Gremlins` [4], and `Stryker Mutator` [5] as a means of introducing faults into the codebase. However, these tools either lacked language support, sufficient maturity, or did not provide fine-grained control over the location and nature of the injected mutations. Moreover, the typical mutation approach focuses on low-level code perturbations that are often unrelated to the behavioral level targeted by MBT. For this reason, we turn to an alternative technique based on fault injection using `failpoint` [6].

[3] `https://github.com/avito-tech/go-mutesting`
[4] `https://github.com/go-gremlins/gremlins`
[5] `https://stryker-mutator.io/`
[6] `https://github.com/pingcap/failpoint`

The `failpoint` package is a lightweight mechanism enabling controlled fault injection in Go applications during runtime. It is distributed under the *Apache 2.0* license and is widely used in large-scale systems. A failpoint defines an injection point in the code, whose associated logic is only executed when explicitly enabled at runtime via the `GO_FAILPOINTS` environment variable. This design ensures that the fault code does not affect the regular behavior of the system when disabled and incurs zero runtime cost in normal operation.

In our experiments, we define *six* failpoints, where each one is inserted at a strategic location within a component, introducing a specific erroneous behavior. These bugs are carefully selected to target both control flow and data integrity across the system. An instance of a simple failpoint declaration is shown in Listing 10.

```
failpoint.Inject("forceWrongReturnAddress", func() {
    returnAddress = "someRandomQueue"
})
```

Listing 10: Overriding return address with hardcoded value.

For our purposes, failpoints are enabled one at a time by uncommenting the appropriate line in the central Dockerfile. The system is then rebuilt and redeployed with the fault injected. This procedure allows for precise control over the presence and scope of each fault without requiring permanent modifications to the application logic or deployment structure. An example of a failpoint flag is depicted below:

```
ENV GO_FAILPOINTS="main/forceDenyApproval=return"
```

Each test run is repeated against a version of the system containing one active failpoint, confirming whether the model accurately encodes the specification and detects behavioral violations. We conduct *two* test runs per failpoint, one for each archetype.

Table 5.2 presents a summary of the failpoints used in our experiments, including their name, target components, and the type of deviation they are intended to introduce.

Failpoint Identifier	Affected Component	Description
`forceWrongReturnAddress`	Agent	Override the return address with a hardcoded queue name unrelated to the current request causing potential misrouting.
`skipSendingToProviders`	API Gateway	Skip all communication with data providers and return a fabricated empty response.
`forceDenyApproval`	Policy Enforcer	Ignore the list of valid data providers and force the request to be rejected regardless of correctness.
`malformCompositionRequest`	Sidecar (Generic)	Corrupt the outgoing composition request message by replacing it with non-decodable data.
`overrideMsCommDestination`	Sidecar (Generic)	Replace the intended message destination with a fake queue leading to delivery to the wrong recipient.
`duplicateFinishedMessage`	sql-algorithm	Send a duplicate `algorithmFinished` message and repeat the downstream communication.

Table 5.2: Injected failpoints and their expected effect on behavior.

Chapter 6

Results

This chapter presents the results of the experiments introduced in chapter 5. Each set of results corresponds to a distinct part of the evaluation strategy and reflects the observed behavior of the SUT. All test runs are executed independently, and the system is freshly deployed before each one to ensure accuracy and consistency. This step is necessary because DYNAMOS tends to degrade in performance over time, with some Kubernetes jobs occasionally failing to terminate correctly and interfering with subsequent executions.

6.1 Functional Test Results

This section reports the outcomes of the functional validation phase, in which each of the core models is executed independently against the system. The goal is to verify that the models are executable, the testing infrastructure is responsive, and the system exhibits behavior consistent with its intended functionality. All tests are executed through the AMP interface, and coverage information is collected via the reporting view.

6.1.1 Initial System Validation

The adapter is recognized by the AMP interface, and the relevant labels are correctly initialized. The first test run is conducted using the `computeToData` model. A summary of the test run is shown in Table 6.1.

Test Run Summary	
Model	`computeToData`
Test run verdict	Pass
Number of test cases	3
Total duration (min)	∼1
Transition coverage (%)	100%
State coverage (%)	100%
Scenarios satisfied	Yes

Test Case Results			
Case	**Verdict**	**Steps**	**Duration (sec)**
1	Pass	22	12.98
2	Pass	17	12.46
3	Pass	24	16.32

Table 6.1: Results for the `computeToData` model test run.

Figure 6.1 shows the AMP test run view and its corresponding coverage chart.

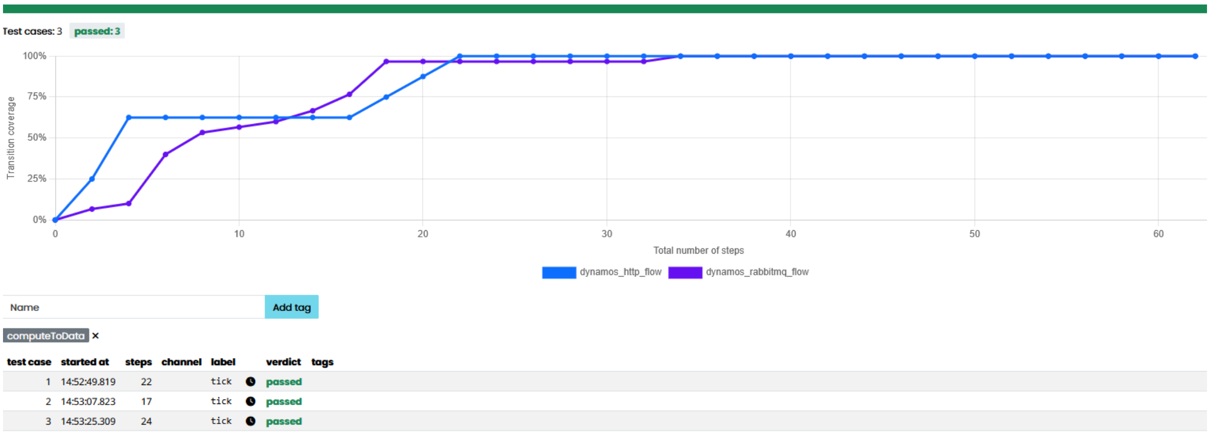
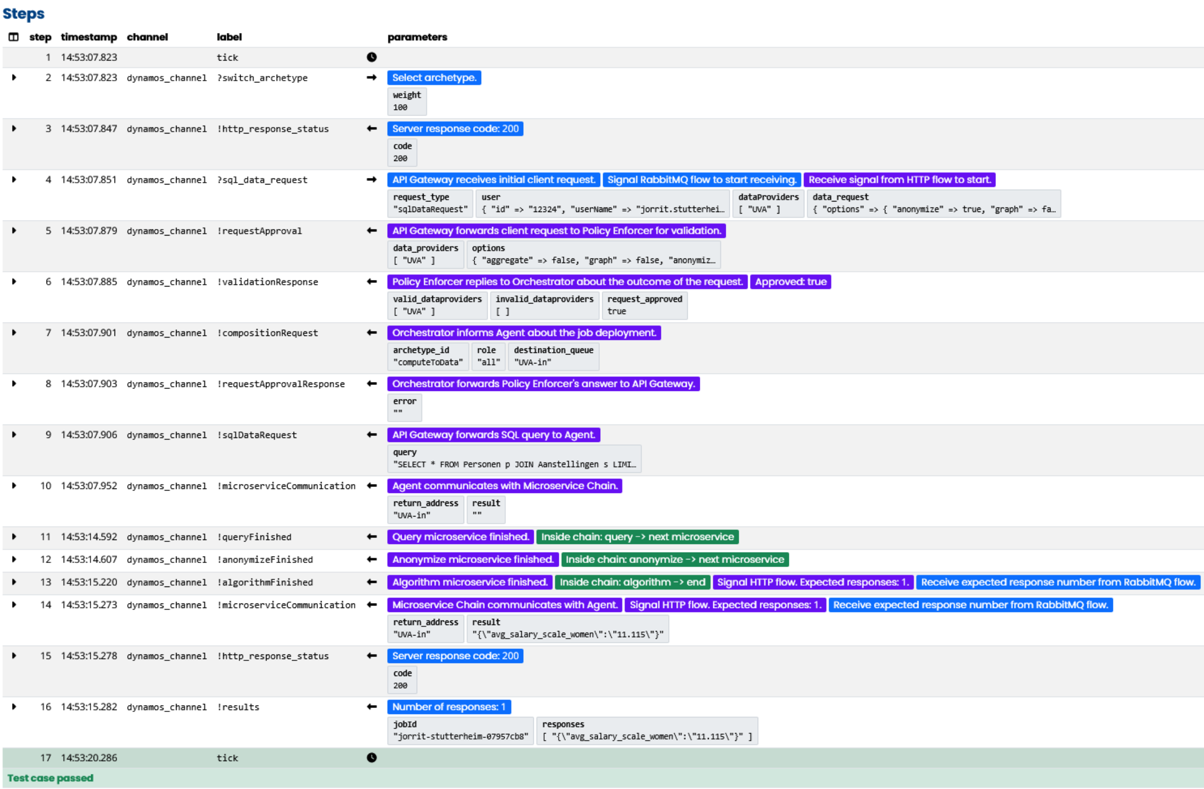


Figure 6.1: AMP test run view for the `computeToData` model.

Figure 6.2 presents a detailed trace of a passing test case from the same run.



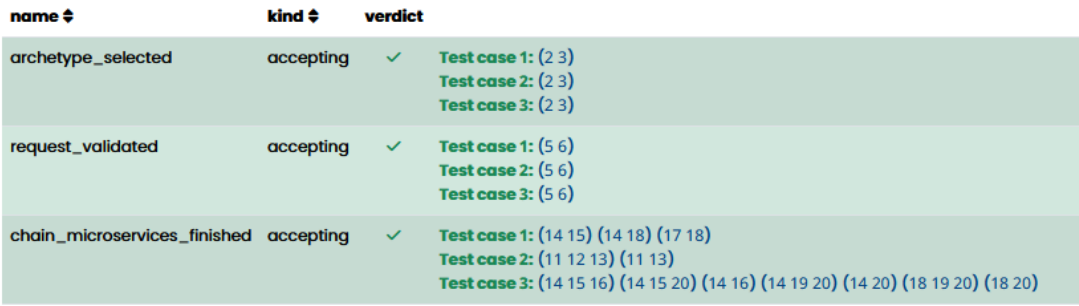
Steps

step	timestamp	channel	label	parameters
1	14:53:07.823		tick	
2	14:53:07.823	dynamos_channel	?switch_archetype	Select archetype. weight 100
3	14:53:07.847	dynamos_channel	!http_response_status	Server response code: 200 code 200
4	14:53:07.851	dynamos_channel	?sql_data_request	API Gateway receives initial client request. Signal RabbitMQ flow to start receiving. Receive signal from HTTP flow to start. request_type "sqlDataRequest" user { "id" => "12324", "userName" => "jorrit.stutterhei… dataProviders ["UVA"] data_request { "options" => { "anonymize" => true, "graph" => fa…
5	14:53:07.879	dynamos_channel	!requestApproval	API Gateway forwards client request to Policy Enforcer for validation. data_providers ["UVA"] options { "aggregate" => false, "graph" => false, "anonymiz…
6	14:53:07.885	dynamos_channel	!validationResponse	Policy Enforcer replies to Orchestrator about the outcome of the request. Approved: true valid_dataproviders ["UVA"] invalid_dataproviders [ ] request_approved true
7	14:53:07.901	dynamos_channel	!compositionRequest	Orchestrator informs Agent about the job deployment. archetype_id "computeToData" role "all" destination_queue "UVA-in"
8	14:53:07.903	dynamos_channel	!requestApprovalResponse	Orchestrator forwards Policy Enforcer's answer to API Gateway. error ""
9	14:53:07.906	dynamos_channel	!sqlDataRequest	API Gateway forwards SQL query to Agent. query "SELECT * FROM Personen p JOIN Aanstellingen s LIMI…
10	14:53:07.952	dynamos_channel	!microserviceCommunication	Agent communicates with Microservice Chain. return_address "UVA-in" result ""
11	14:53:14.592	dynamos_channel	!queryFinished	Query microservice finished. Inside chain: query -> next microservice
12	14:53:14.607	dynamos_channel	!anonymizeFinished	Anonymize microservice finished. Inside chain: anonymize -> next microservice
13	14:53:15.220	dynamos_channel	!algorithmFinished	Algorithm microservice finished. Inside chain: algorithm -> end Signal HTTP flow. Expected responses: 1. Receive expected response number from RabbitMQ flow.
14	14:53:15.273	dynamos_channel	!microserviceCommunication	Microservice Chain communicates with Agent. Signal HTTP flow. Expected responses: 1. Receive expected response number from RabbitMQ flow. return_address "UVA-in" result "{\"avg_salary_scale_women\":\"11.115\"}"
15	14:53:15.278	dynamos_channel	!http_response_status	Server response code: 200 code 200
16	14:53:15.282	dynamos_channel	!results	Number of responses: 1 jobId "jorrit-stutterheim-07957cb8" responses ["{\"avg_salary_scale_women\":\"11.115\"}"]
17	14:53:20.286		tick	

Test case passed

Figure 6.2: Test case trace in the `computeToData` test run.

Figure 6.3 depicts the verdict of scenarios from the same run.



name	kind	verdict	
archetype_selected	accepting	✓	Test case 1: (2 3) Test case 2: (2 3) Test case 3: (2 3)
request_validated	accepting	✓	Test case 1: (5 6) Test case 2: (5 6) Test case 3: (5 6)
chain_microservices_finished	accepting	✓	Test case 1: (14 15) (14 18) (17 18) Test case 2: (11 12 13) (11 13) Test case 3: (14 15 16) (14 15 20) (14 16) (14 19 20) (14 20) (18 19 20) (18 20)

Figure 6.3: Execution of scenarios in the `computeToData` test run.

The second test run is executed using the `dataThroughTtp` model. A summary of the test run is shown in Table 6.2.

Test Run Summary	
Model	`dataThroughTtp`
Test run verdict	Pass
Number of test cases	3
Total duration (min)	∼1
Transition coverage (%)	100%
State coverage (%)	100%
Scenarios satisfied	Yes

Test Case Results

Case	Verdict	Steps	Duration (sec)
1	Pass	29	17.12
2	Pass	23	13.56
3	Pass	29	15.79

Table 6.2: Results for the `dataThroughTtp` model test run.

The third test run uses the `combinedArchetypes` model. A summary of the test run is shown in Table 6.3.

Test Run Summary	
Model	`combinedArchetypes`
Test run verdict	Pass
Number of test cases	3
Total duration (min)	∼1
Transition coverage (%)	100%
State coverage (%)	100%
Scenarios satisfied	Yes

Test Case Results

Case	Verdict	Steps	Duration (sec)
1	Pass	22	17.67
2	Pass	29	15.11
3	Pass	17	12.54

Table 6.3: Results for the `combinedArchetypes` model test run.

6.1.2 Execution Under Bad Weather Conditions

Variations of the `computeToData` and `dataThroughTtp` models are tested with the `bad weather` flag enabled. Table 6.4 and Table 6.5 summarize the results.

Test Run Summary	
Model (bad weather)	`computeToData`
Test run verdict	Pass
Number of test cases	3
Total duration (min)	~2
Transition coverage (%)	60%
State coverage (%)	69%
Scenarios satisfied	No

Test Case Results			
Case	**Verdict**	**Steps**	**Duration (sec)**
1	Pass	14	25.16
2	Pass	13	25.17
3	Pass	13	25.09

Table 6.4: Results for the `computeToData` model test run under bad weather conditions.

Test Run Summary	
Model (bad weather)	`dataThroughTtp`
Test run verdict	Pass
Number of test cases	3
Total duration (min)	~2
Transition coverage (%)	60%
State coverage (%)	73%
Scenarios satisfied	No

Test Case Results			
Case	**Verdict**	**Steps**	**Duration (sec)**
1	Pass	14	25.19
2	Pass	13	25.12
3	Pass	14	25.12

Table 6.5: Results for the `dataThroughTtp` model test run under bad weather conditions.

6.1.3 Supplementary Model Results

To support the functional evaluation, we execute the auxiliary `bugs` and `witness` models. Table 6.6 and Table 6.7 present the test run verdict for each model.

Test Run Summary	
Model	`bugs`
Test run verdict	Fail
Number of test cases	4
Total duration (min)	<1
Transition coverage (%)	73%
State coverage (%)	100%
Scenarios satisfied	No

Test Case Results			
Case	**Verdict**	**Steps**	**Duration (sec)**
1	Fail	13	0.17
2	Fail	13	0.29
3	Fail	11	0.09
4	Fail	11	0.09

Table 6.6: Results for the `bugs` model test run.

Test Run Summary	
Model	`witness`
Test run verdict	Pass
Number of test cases	4
Total duration (min)	~2
Transition coverage (%)	77%
State coverage (%)	100%
Scenarios satisfied	No

Test Case Results			
Case	**Verdict**	**Steps**	**Duration (sec)**
1	Pass	14	25.16
2	Pass	14	25.10
3	Pass	12	25.11
4	Pass	12	25.09

Table 6.7: Results for the `witness` model test run.

6.2 Results Under Fault Injection

We execute a series of test runs against faulty system variants. Each variant includes a single activated failpoint, as described in chapter 5. Table 6.8 summarizes the verdicts across all experiments, while Table 6.9 presents a concise summary of the trace-level observations.

Failpoint Identifier	**computeToData**	**dataThroughTtp**
`forceWrongReturnAddress`	Pass	Fail
`skipSendingToProviders`	Fail	Fail
`forceDenyApproval`	Fail	Fail
`malformCompositionRequest`	Fail	Fail
`overrideMsCommDestination`	Fail	Fail
`duplicateFinishedMessage`	Fail	Fail

Table 6.8: Test run verdicts under activated failpoints.

Failpoint Identifier	**computeToData**	**dataThroughTtp**
`forceWrongReturnAddress`	Test cases pass with expected number of responses.	List of responses contains only empty strings. No `microserviceCommunication` responses from microservice completions. Test cases fail due to unexpected HTTP responses.
`skipSendingToProviders`	List of responses empty. No `microserviceCommunication` responses present. Test cases fail due to unexpected HTTP responses.	List of responses empty. No `microserviceCommunication` responses present. Test cases fail due to unexpected HTTP responses.
`forceDenyApproval`	Request denied by `Policy Enforcer`. No `microserviceCommunication` responses observed. Test cases fail due to unexpected HTTP responses.	Request denied by `Policy Enforcer`. No `microserviceCommunication` responses observed. Test cases fail due to quiescence or unexpected HTTP responses.
`malformCompositionRequest`	List of responses contains only empty strings. No `microserviceCommunication` responses observed. Test cases fail due to unexpected HTTP responses.	List of responses contains only empty strings or no response received. No `microserviceCommunication` responses observed. Test cases fail due to quiescence or unexpected HTTP responses.
`overrideMsCommDestination`	List of responses contains only empty strings. No `microserviceCommunication` responses observed. Test cases fail due to quiescence or unexpected HTTP responses.	List of responses contains only empty strings. No `microserviceCommunication` responses observed. Test cases fail due to unexpected HTTP responses.
`duplicateFinishedMessage`	List of responses contains expected number of entries. Each `microserviceCommunication` response duplicated. Test cases fail due to repeated completion signals.	List of responses contains only empty strings. No `microserviceCommunication` responses observed. Test cases fail due to unexpected HTTP responses.

Table 6.9: Trace-level observations under each activated failpoint.

Chapter 7

Discussion

This chapter reflects on the outcomes of the study and places them within the context of the original research objectives. We begin by interpreting the results of the case study experiments and formulating judgments about the SUT and the proposed method. We proceed to answer the research questions directly, drawing on evidence from the testing infrastructure and experiments. Finally, we examine the limitations of the approach and threats to the validity of the research.

7.1 Interpretation of Results

System Behavior & Observed Instability

A first and critical observation that does not appear explicitly in our reported outcomes is that DYNAMOS often exhibits unpredictable and unreliable behavior. This means that even though a request may be valid, the system might still respond with an empty result. Such seemingly non-deterministic behavior is more commonly observed, but not exclusively so, directly after an invalid request has been made or an internal error has occurred. This flakiness becomes evident to the tester when a test run fails despite being based on a correct model that passes under identical conditions. What we observe in the test trace is that immediately after receiving the initial HTTP stimulus, and without any of the usual internal messages being recorded beforehand, the system appears to send responses corresponding to later stages of execution, such as `microserviceCommunication` messages.

This anomaly leads us to focus our attention on how the system handles malformed or invalid user input. Indeed, upon inspecting the state of the cluster during such cases, we observe that certain ephemeral jobs do not terminate correctly and remain active during the execution of subsequent requests. These lingering components are likely to interfere with the internal logic that instructs behavior based on the number of ephemeral services currently alive. While this remains a speculation, it is a strong candidate for the root cause of the observed issues. What matters most, however, is that our testing mechanism is capable of detecting the presence of such issues reliably, provided that they alter the system's observable and modeled behavior.

Execution Results of the Core Models

We now turn to the interpretation of the results we gathered, which, as previously noted, are free from the aforementioned unstable behavior. Starting from the fundamentals, we observe that no issues were encountered in the connection between the adapter and AMP, nor in the registration of the supported labels. This ensures reliability and contributes to a frictionless testing process, which always executes without interruption. Moving to the core models, we find that they execute successfully and without incident. In particular, the `computeToData` model, as shown in Table 6.1, yields a successful test run, with all three test cases passing. The total duration of the test run, approximately one minute, along with the successful satisfaction of the defined `scenarios` (see Figure 6.3), is in line with expectations. The same applies to the transition and state coverage, both of which reach 100%. This is because the model traverses every possible transition and state in its STS.

As seen in the detailed test case results, the number of steps per test case is not fixed. This is due to the fact that the solver selects different input values each time, which leads the execution down distinct paths. For instance, in the third test case, anonymization is enabled and more than one provider is present, resulting in a higher total number of steps compared to the other test cases in the same run. Furthermore, the duration of each execution falls well within the expected range. In Figure 6.1, we observe the two processes executing in parallel and achieving full transition coverage, both independently and as a whole. Figure 6.2 illustrates a representative successful test case and shows the sequence of labels in the order they appeared, each annotated with a note offering explanatory context. The colors correspond to the processes depicted in Figure 6.1, with the addition of green to indicate messages originating from microservices in the chain.

At the end of each test case, we observe that the SUT responds with the expected outputs. Once the final message has been sent, the model waits for the configured timeout period in order to declare quiescence. At that point, it can safely determine that the test case has passed.

The second test run concerns the `dataThroughTtp` model, which also completes successfully. All previously stated observations apply here as well. As shown in Table 6.2, the number of steps varies across test cases and is, on average, higher than that of the `computeToData` model. This is due to the fact that this archetype includes an additional chain microservice, `aggregate`, as well as a greater number of `microserviceCommunication` messages, resulting from the involvement of the trusted third party. Accordingly, the execution durations are slightly longer, which is fully expected.

Finally, the combined model `combinedArchetypes` also executes successfully, as shown in Table 6.3. This model primarily serves as a mechanism for granting the tester increased observational freedom, since its rules are not as strict as those in the archetype-specific models. An additional purpose is to demonstrate that archetype activation in DYNAMOS can be modeled dynamically, without being predefined statically. In doing so, it becomes clear that MBT can be adapted to suit the needs of an adaptive system. As discussed in section 4.3, it becomes evident that additional archetypes could easily be integrated into the modeling approach, should they be implemented in the future.

Behavior Under Bad Weather Conditions

The next point of interest concerns the behavior of the system and the models under bad weather conditions. This helps us understand how the system responds when circumstances deviate from the expected flow, and how well the models capture these alternative paths.

The first results in this category relate to the `computeToData` model. As shown in Table 6.4, the model passes successfully. It is important here to interpret this outcome correctly. The success of the run does not indicate that an erroneous situation went undetected, but rather that under certain conditions, the system is capable of handling malformed input gracefully. In this particular execution, the model runs for approximately two minutes, a duration consistent with the fact that an intentionally invalid request was issued, which the system attempts to process internally. Furthermore, the failure of the `scenarios` is expected. Coverage, both transition and state, is partial because the model is never able to traverse the happy flow paths, as the necessary conditions are absent. This is a direct consequence of the fact that the bad weather path is selected, which is also reflected in the number of steps executed in each test case. The execution durations are also higher, as the model awaits internal stimulation from the `dynamos_rabbitmq_flow` process which never arrives.

The second set of results pertains to the `dataThroughTtp` model (see Table 6.5). The observations are very similar to the previous case, which is entirely expected, given that both models share the `dynamos_http_flow` process that handles the bad weather logic.

Evaluation of Supporting Models

We now turn to the interpretation of the results from the supporting models, which are of significant importance. Beginning with the `bugs` model, we observe that the test run fails (see Table 6.6). This outcome is expected and desirable by design, as the model contains predefined faulty scenarios where correct behavior is asserted but, by definition, not observed. The number of test cases is four, matching the number of injected bugs. The total duration is under one minute due to early termination. The `scenarios` fail as expected, and transition coverage is incomplete, since not all responses are received. Each test case fails quickly because the satisfaction condition, which requires responses to be non-empty, is not met in any of the provided traces.

Next is its counterpart, the `witness` model. As shown in Table 6.7, this model passes successfully, as anticipated. The reason lies in the satisfaction condition, which checks whether the system indeed returned empty responses. This is confirmed, as the test cases reproduce the exact same defects as the `bugs` model, but in this case, the model is defined to accept them as valid outcomes.

Fault Injection & Detection Capabilities

The final and highly significant point concerns the interpretation of results when the SUT has been mutated. As seen in Table 6.8, unexpected behavior is detected in slightly over 91% of the cases we examined. The only instance where such behavior goes unnoticed is when a `failpoint` has been introduced in a location that affects only the `dataThroughTtp` archetype.

Turning our attention to Table 6.9, we gain a comprehensive view of the effectiveness of our modeling approach in identifying the source of the divergent behavior. Focusing on `forceWrongReturnAddress`, in one case we observe no detection, while in the other, although `microserviceCommunication` messages are present, none correctly involve the trusted third party. Moving on to `skipSendingToProviders`, we confirm that no `sqlDataRequest` messages appear in the trace for either archetype, which is consistent with expectations. Regarding `forceDenyApproval`, both models correctly reflect that the Policy Enforcer rejected the request, which is highly informative. As for `malformCompositionRequest`, while the trace does not directly indicate a malformed composition request, the model fails in every case, albeit for different reasons. In the case of `overrideMsCommDestination`, we observe that the expected `microserviceCommunication` messages are missing, as are the internal messages from the chain. Finally, injecting the `duplicateFinishedMessage` fault results in the detection of a duplicated `algorithmFinished` message by the `computeToData` model, while `dataThroughTtp` fails due to the absence of `microserviceCommunication` messages.

In conclusion, our method proves effective in detecting the majority of system faults. An additional advantage is its ability to assist with bug localization, though this does not apply in all cases and depends on the specific model condition being violated.

7.2 Answers to Research Questions

In this section, we present the findings of this research and connect them directly to the initial research questions posed in chapter 1. Each question is addressed individually, based on the evidence gathered from the design, implementation, and experimental evaluation of the proposed approach.

7.2.1 Research Question 1 (RQ1)

How can the behavior of an adaptive microservice system be systematically observed and exposed to enable model-based testing?

The first step is to define both the scope and depth at which testing is to be conducted. This decision depends on the desired level of detail as well as the degree of visibility that can be achieved into the SUT. In this thesis, we adopt a *gray-box E2E* testing strategy. This choice reflects our intention to thoroughly test the entire system, including every microservice, which is made possible by having unrestricted access to all internal components. The method is not characterized as black-box because we are concerned with behavior at the component level, which grants insight into the internal functionality and message exchange within the system. Given that this selection is both feasible and effective in our setting, we arrive at Finding 1.

> **Finding 1:** The proposed approach for testing an adaptive microservice system using Model-based Testing is gray-box E2E, with a focus on the communication layer.

Once this initial strategy is established, a mechanism must be defined in order to observe behavior in a systematic way. Since our focus is on inter-component communication, the chosen mechanism should be tightly coupled to that layer, without disrupting normal execution. This naturally leads to a passive observation approach. Given that the system already relies on the AMQP protocol, we introduce a dedicated queue for testing purposes. This queue receives real-time copies of all messages exchanged across components, which are consumed by the adapter, which is the intermediate component of the MBT infrastructure.

Because capturing messages from all possible locations may be challenging, one viable strategy is to identify the system's primary message dispatch mechanism, if one exists. This reduces the number of instrumentation points required, making the setup more readable and maintainable. In our case study, we leverage the sidecar, which is deployed alongside each component and handles communication on its behalf. This provides a natural point of integration for observation. When systems rely on multiple protocols, such as gRPC and HTTP, communication is not always centralized, and more involved adjustments are required to enable full instrumentation. This is particularly relevant for ephemeral components whose behavior and lifecycle are dynamic and protocol-specific. These also require the message to be sent synchronously to the queue, as they may terminate before transfer can be completed. Nevertheless, in the system we examine, all inter-component communication eventually passes through the RabbitMQ broker, offering a centralized mechanism for message capture. The above is summarized in Finding 2.

> **Finding 2:** Systematic observation of adaptive microservice behavior can be achieved by passively capturing inter-component communication through a centralized dispatch layer, such as a message broker. When access to this layer is provided via sidecar components, instrumentation becomes more modular and less intrusive. However, in systems involving multiple communication protocols, additional adaptations may be required to maintain consistent visibility.

Once a suitable message collection method is established, it becomes necessary to determine how these messages should be structured in a uniform way. The decision to represent each message as a pair consisting of a `type` and a `body` ensures compatibility and provides the contextual information needed to trace the message's origin during later stages. These messages, however, do not travel through the medium in their original form. To ensure their integrity, a serialization step is required. The message is first converted into a binary format, which is then encoded as a base64 string. Only in this form can it be safely embedded within a JSON object and transmitted.

The recipient of these messages is the adapter, whose main role is to translate the communication between AMP and the SUT, in both directions. To maximize accuracy during testing, the adapter must be placed as close to the system as possible. Since DYNAMOS is deployed within a Kubernetes cluster, we deploy the adapter as a pod. In terms of architecture and implementation, we follow Axini's Plugin adapter protocol. Among its responsibilities, the adapter handles the initialization phase and subscribes to the testing queue. As messages arrive from DYNAMOS, they must be forwarded immediately to the platform. To do this, the adapter must validate all relevant fields, handle errors gracefully to avoid breaking the flow, and deserialize the body of incoming messages, reversing the encoding process described earlier.

Additionally, it must accept and process messages originating from AMP and intended for the SUT, using different underlying protocols. These requirements lead us to the conclusion captured in Finding 3.

> **Finding 3:** Unlike in typical reactive systems where the adapter remains relatively uncomplicated, in adaptive microservice environments the adapter must be more elaborate, as it is responsible for handling multiple heterogeneous and intricate message structures.

7.2.2 Research Question 2 (RQ2)

What modeling strategies can be developed to specify the behavior of an adaptive microservice system?

In order to make the system amenable to modeling, one must first decide how the testing platform will receive the behavioral observations in the form of labels. Regarding the delivery strategy, we choose to send each message to the platform as soon as it is received, rather than batching multiple messages together. This approach complies with the established methodology and provides more direct control over the testing process.

As for the message structure, while some degree of freedom exists in formatting, we opt to preserve the original structure intact. The only exception is that for certain message types, we forward only a subset of their internal fields to AMP, in order to reduce model clutter. In practice, the `type` of the message serves as the label, while the decoded `body` serves as its parameters. This method ensures accuracy, clarity in the resulting models, and ease of future extension when new message types are introduced. The essence of this mapping is captured in Finding 4.

> **Finding 4:** Messages exchanged within the SUT and their respective fields correspond directly to model labels and their parameters.

In our case study, we needed to account for two archetypes, meaning two distinct behavioral patterns triggered by incoming request characteristics and current conditions. While these could have been modeled within a single artifact, we chose to separate them into two dedicated models, one per archetype, for reasons of readability and scalability. Both models share a common process, `dynamos_http_flow`, which captures the external behavior of the system as observed through the HTTP protocol. Each model defines its own `dynamos_rabbitmq_flow` process to describe the internal messaging flow, which occurs over AMQP and, in part, over gRPC.

These two processes execute in parallel and communicate, enabling modularity, maintainability, and clarity, while also allowing for responses to be accepted from either process, as long as permitted by the model. This design proves especially useful for testing bad weather scenarios. Notably, the fact that labels can uniformly represent messages regardless of the underlying protocol, even when they originate from microservices within a spawned chain, further enhances abstraction. These characteristics are reflected in Finding 5.

> **Finding 5:** The main communication flows of the system can be separated by protocol and modeled as parallel synchronizing processes, whose labels are agnostic to the underlying transport protocol, thereby promoting uniformity.

In modeling system behavior, two primary factors must be considered, namely order and state. Order refers to the required sequence in which messages must occur for an implementation to be deemed conforming, as dictated by system requirements. In the case of DYNAMOS, although some message types must precede others, the system exhibits a strong degree of asynchrony. This behavior is expected, given that the system relies on the AMQP protocol, and presents a modeling challenge in terms of accurately representing the timing and order of message exchanges.

To address this, we found that the `choice` and, by extension, the `repeat` construct in AML are essential for correctly modeling such exchanges without always enforcing strict sequencing. In many cases, and in accordance with the specification, certain data may arrive in slightly varying orders. As for state, this refers to the internal condition of the system at a given point in time. It is managed through variables and constraints. In our study, it was necessary to introduce domain-specific notions such as providers, roles, and other context-aware constructs that directly influence the labels and parameters, and in turn, enable or disable transitions under certain conditions.

Some of these constraints pertain specifically to the adaptive nature of the system and its obligation to respond differently depending on which archetype is active and how the user's request is configured. These insights are summarized in Finding 6.

> **Finding 6:** The `choice` and `repeat` constructs provide the essential mechanism for modeling asynchronous behavior. This form of behavior is especially relevant for responses in adaptive microservice systems, where such labels can also represent internal communication, beyond their typical use in modeling interactions with the external environment.

7.2.3 Practical Usage Considerations

As established in chapter 1, our primary objective has been to examine the applicability of Model-based Testing in adaptive microservice systems, with the goal of assessing their behavioral correctness. Based on the experimental procedure we followed, we interpret the results as strong indications that this is indeed feasible. On the one hand, conforming behavior can be accurately modeled, and on the other, deviations from it can be detected to a satisfactory degree.

Beyond this evaluation, the value of our study also lies in considering how the developed testing infrastructure can be used in practice. Our proposal is to integrate it into the CI/CD pipeline, specifically as part of the system's integration workflows. This choice is motivated by the fact that, due to the nature of our models, test failures often signal the presence of an issue without necessarily revealing its exact cause. By executing and evaluating the core models upon each push, we strengthen fault localization, as failures can then be correlated with recent changes, provided the models passed prior to those changes.

In practice, our testing mechanism produces two key artifacts, namely the adapter logs and the test run results. Together, they constitute actionable insight for developers. When the result of a test execution is a failure, a GitHub Issue [1] can be automatically created to include these artifacts, along with

[1] `https://docs.github.com/en/issues/tracking-your-work-with-issues/about-issues`

any additional debugging information. The automation of this process can be achieved using GitHub Actions [2]. This positions MBT not only as an evaluation tool but as a live diagnostic utility within the development cycle.

7.3 Limitations

While the approach proved effective within the context of this study, certain limitations are observed that define the boundaries of its current applicability. These provide important context for identifying areas where future refinement may be beneficial.

Limitation to a Research Tool The case study examined in this thesis is not a production-grade system. As a result, in several instances, we had to apply workarounds and limit the range of values and paths explored during testing, which in turn affects the thoroughness of the method. Examples include the placement of hardcoded values where arbitrary choices could theoretically have been allowed, and the restriction of the modeling to just two archetypes. A further consequence of not using a commercial system is the inability to evaluate the approach under real-world conditions.

Implementation Assumptions & Generalizability A central assumption in this study concerns the ability of the communication layer to render the system sufficiently observable. If hidden or internal state must be accessed to determine correctness, then the proposed approach cannot be considered holistic, as it captures only part of the system's behavior. Another point relevant to modeling and the feasibility of effective E2E testing is the use of a single stimulus with variations. This choice is justified by the operational logic of the system. However, had we examined case studies involving richer interactions, we might have gained a more comprehensive perspective on how to best approach modeling for adaptive microservice systems, regardless of their particular characteristics.

With respect to the architecture, during the design phase we sought to leverage the system's existing structures and mechanisms as much as possible. The aim was to craft a solution tailored to the case at hand while also enabling reusability. However, this implies that in systems lacking similar infrastructures, our solution may diverge significantly, thereby affecting generalizability. A characteristic example of such an assumption is the presence of the sidecar, which plays a central role in the implementation. Without an equivalent component, instrumentation code would need to be injected at more points throughout the system, rendering the integration more intrusive. The same applies to the use of RabbitMQ, which enabled us to take advantage of a pre-existing setup, which is something that cannot be assumed in environments relying on different internal communication protocols. If the approach had been tested on multiple systems, a more general-purpose observation and behavior extraction mechanism might have emerged.

Adaptability to System Evolution Since the system was examined at a specific point in time, it is only natural that the implementation was tied to that version. During the development of our solution, DYNAMOS evolved, meaning that bugs were fixed, functionality was extended, and various internal communication mechanisms were modified. The inevitable binding meant that the solution remained tailored to the current version, while significant adjustments would be required for equivalent testing to be performed in the future. Changes concerning the modeling layer are relatively straightforward to implement, as the overall behavior largely remains the same. However, adjustments to the testing infrastructure are more involved. Even a minor modification in a critical area may necessitate a partial redesign of the integration mechanism.

Restricted Demonstration of Usability Although the solution can be considered successful in fulfilling its primary purpose, time constraints prevented further attempts at integrating it into the system's development cycle. Had such integration been achieved, the practical usefulness of the solution would have been more clearly demonstrated, as automated model execution would provide concrete evidence of its ease of use and effectiveness in fault detection.

[2] `https://docs.github.com/en/actions/get-started/understanding-github-actions`

7.4 Threats to Validity

In order to ensure a well-rounded evaluation of the proposed approach, we consider potential threats to validity that may affect the interpretation or generalization of the results. Following established guidelines, we organize these threats into categories, namely construct, internal, external, and conclusion validity.

Construct Validity

The testing and hence the evaluation approach is based on observing system messages and matching them to abstract labels, capturing only the externally visible layer of behavior. Internal state changes that are not reflected in messages remain outside the scope. In such cases, metrics like transition and state coverage may overstate the actual thoroughness of the tests if the models abstract away important aspects.

The adapter maps only selected fields from messages. This simplifies integration, but it may omit information relevant to behavioral correctness. When certain fields are excluded or generalized, deviations may go undetected. In effect, the scope of what is tested is shaped not only by the model but also by what the adapter chooses to expose, which limits how fully system behavior is evaluated.

Internal Validity

During testing, occasional instability in the system made it unclear whether some failures reflected actual faults or temporary conditions. Cases involving delayed responses or leftover jobs sometimes produced unexpected behavior. This introduces ambiguity about whether a failed test points to a system error or to uncontrolled side effects during execution.

Another limitation is the use of a single configuration for all test runs. While this was sufficient for demonstration, it may restrict the variety of behaviors observed. If some faults only emerge under specific conditions, running the models with one setup makes it harder to isolate the causes of success or failure, and may hide behaviors that would otherwise surface.

External Validity

Our study evaluates the method on a single system. Although DYNAMOS includes many elements common to adaptive microservice architectures, it remains uncertain whether our research findings would generalize to systems with different structures, protocols, or deployment setups.

Additionally, the system was tested under ideal conditions in a local development environment. This setup ensures control and reproducibility but does not reflect real-world complexity. Factors like concurrent usage, network latency, or production-scale deployment could significantly affect both system behavior and performance of the testing mechanism.

Conclusion Validity

Each test run included only three test cases. While this was enough to demonstrate basic functionality, it is not sufficient to capture rare or subtle faults. In complex systems such as DYNAMOS, a larger and more diverse test suite would be required to draw confident conclusions about correctness.

The number of injected faults was also limited. These were introduced in specific components based on predefined assumptions. Other categories of faults, such as network errors, were not explored. Combined with the small number of modeled scenarios, this reduces the strength of any general claim about the method's overall fault detection effectiveness.

Chapter 8

Related Work

This chapter includes previous research and tools related to this study. Specifically, it mentions works concerning both microservices testing and MBT, which either provide a foundation for this project or offer an overview of the research landscape and the state-of-the-art.

Microservices Testing

Regarding microservices testing, Waseem et al. [4] find that five research themes characterize testing approaches in MSA-based applications across the 33 studies examined, of which less than 10% involve MBT. They also recognize the overall lack of dedicated tools to support testing for MSA-based applications, which are difficult to test due to their complex nature and dynamic behavior. This gap is what our approach attempts to bridge by connecting microservices with MBT, while at the same time suggesting a way to address the complexity of this architecture. This need is also confirmed by Ghani et al. [28], who state that few studies discuss E2E testing and that there is a call for more practical experimentation.

The majority of existing approaches focuses on specific quality attributes. These works demonstrate the broad interest in improving microservices testing but do not address behavioral correctness. For example, concerning resilience, Heorhiadi et al. [29] introduce Gremlin to analyze the failure-handling capabilities of microservices. Gremlin leverages the fact that microservices use common message-exchange patterns, allowing operators to design and conduct tests by modifying network-layer interactions between services. This resembles the testing of bad weather behavior that we conduct, despite the fact that we do not operate at the network level. Regarding performance, Camilli et al. [30] apply load testing in order to learn the behavior of the target system under several configurations. The process refines the initial model to obtain a complete specification from runtime evidence and then conducts automated verification. Gazzola et al. [31] present ExVivoMicroTest, an approach that observes services at runtime to generate test cases that can be applied to subsequent versions. It implements lightweight monitoring and tracing to record executions that can be turned into regression test cases that capture how services are used. As for contract testing, Wu et al. [32] introduce CCTS, a tool that shares some elements with our approach. It records the transitions of event exchange between services and retrieves the possible paths among them. Also, it analyzes the event logs from the system to determine whether they conform with the specified transitions and retrieved paths. Pact, an industry-standard tool, performs consumer-driven contract testing, also focusing on interaction points rather than full system behavior.

Model-based Testing

Model-based testing is a mature field that has been methodically researched for more than two decades. Marques et al. [33] contrast it with traditional testing strategies, noting among other things that MBT offers reusable, shareable, and consistent test suites that help validate system behavior and specification, which was one of our motivations for examining it in the microservices context. Additionally, Mlynarski et al. [23] claim that abstract models can support communication, documentation, and testing in agile processes, which closely aligns with what we attempt to demonstrate. Utting et al. [34] present a paper that characterizes different MBT approaches, which helped us pick the optimal strategy for our case. There is also a wide range of tools, some of which are open-source. For example, Intel only recently archived fMBT [1], which can be applied to testing a wide range of components. TorXakis [35] is an experimental tool for on-the-fly model-based testing, implementing test generation for Symbolic Transition Systems. Although it is powerful in theory, it has practical limitations, such as usability. GraphWalker [36] is an MBT tool that reads models in the shape of directed graphs and generates test paths from them. Zafar et al. [24] apply it on an industrial cyber-physical system to evaluate MBT. Our work is based on the works of Tretmans [18][37] that introduce the IOCO theory, which is of fundamental importance. Additionally, [20] played a significant role in how we structured our supporting models, such as the `bugs` and the `witness`.

Model-based Testing of Microservices

To the best of our knowledge, little literature has specifically focused on the intersection of MBT and microservices and only a few notable works can be identified, reinforcing the novelty of our approach. Van den Brink et al. [5] were among the first to systematically explore the applicability of MBT to microservices. They developed a formal foundation for applying compositional MBT to microservices by modeling them as STSs. Their work focused on preserving IOCO conformance under composition through completion algorithms, such as demonic and protocol completion, and evaluated the approach with a proof-of-concept tool on an open-source system. While demonstrating that MBT can effectively test individual microservices, the study reported severe state-space explosion and did not yet provide a practical solution for large-scale or asynchronous systems.

Booy [6] extended previous work by proposing model-based mocking, a theoretically grounded method to enable compositional testing of microservices. The approach replaces the underlying services of the SUT with mocks generated directly from their specifications, aiming to make component-wise testing feasible. It was implemented in AMP and evaluated using the same system as the previously mentioned study, where 20 mutants were introduced. Mocking detected more than half of the bugs significantly faster than other methods and was never slower, demonstrating its potential for improving testing efficiency. However, the method assumes that the specifications of the mocked services are correct and complete, and it has not been validated on highly dynamic or adaptive systems. Thus, its applicability to real-world microservice ecosystems remains limited.

Ganzevoort [7] applied MBT directly to the DYNAMOS middleware, integrating van den Brink's proof-of-concept tool into a real deployment. Their work focused exclusively on testing the Agent component, both in isolation and within job chains, leaving the other core components outside the scope of testing. The `dataThroughTtp` archetype was modeled but not tested, limiting the experiments to the simpler `computeToData` scenario. The main challenges they identified included handling asynchronous outputs, creating input-enabled specifications, dealing with quiescence bottlenecks, and correctly isolating the SUT. To improve stability, a mocked job-chain setup was introduced, which increased testing performance but masked certain types of faults, demonstrating the difficulty of fully isolating microservices in a realistic environment. In contrast, our work extends beyond Agents to the entire DYNAMOS architecture, modeling and testing interactions across all components and archetypes, thereby enabling a more realistic assessment of adaptive behavior under production-like conditions.

[1] `https://github.com/intel/fMBT`

Chapter 9

Conclusion

Microservice systems introduce unique challenges for testing, as their distributed and complex nature makes it difficult to assess correctness. Traditional strategies, such as unit or contract testing, focus on narrow aspects and often fail to capture interactions and behavior holistically. These challenges become even more pronounced in self-adaptive systems, where services are instantiated and terminated on demand, creating highly transient behavior. Model-based Testing (MBT) has been proposed as a promising solution for such environments because of its ability to verify conformance against a specification in an automated and systematic way. In this context, DYNAMOS was selected as a representative case study due to its adaptive architecture and its role as a research platform for data-sharing systems, making it an ideal candidate to evaluate the feasibility of MBT in a realistic yet controlled setting.

To address the first research question, a passive observation mechanism was introduced to make behavior accessible for testing in a reliable and minimally intrusive way. Our infrastructure leveraged the existing sidecar architecture and extended the communication layer with a dedicated testing queue in RabbitMQ. An adapter was developed to consume the observed messages, translate them into abstract labels, and forward them to the Axini Modeling Platform (AMP). This setup allowed message exchanges to be collected centrally without interfering with normal execution and enabled the detection of interactions involving ephemeral components. By observing communication rather than internal logic, it provided a gray-box, End-to-End view of the system that could be directly used for model-based testing.

The second research question examined how to model this behavior. Each message was forwarded to the testing platform as soon as it was received, preserving its original structure to maintain accuracy, while forwarding only selected internal fields for certain types to reduce complexity. The type of each message served as the label, with the decoded body used as its parameters. Two primary models were developed in the Axini Modeling Language (AML), one for each archetype, sharing a common process for external interactions and defining separate processes for internal messaging. These flows executed in parallel and synchronized when required, promoting modularity and enabling responses to be accepted from either process, especially in bad weather scenarios. To correctly model asynchronous behavior, specific AML constructs proved to be of great importance and were used extensively, while variables and context-aware constraints guided transitions, reflecting the system's responses under different conditions.

Our experimental evaluation confirmed that the proposed approach can be effectively applied to a microservice environment. The models achieved full coverage and detected the majority of injected faults, demonstrating that MBT can reveal deviations from the expected behavior, even in highly dynamic scenarios. These results indicate that model-based testing is a practical and valuable technique for increasing confidence in the correctness of adaptive microservice systems, particularly when used to complement existing strategies. As Dijkstra famously stated, "Program testing can be used to show the presence of bugs, but never to show their absence." In line with this principle, the goal of this research was not to prove absolute correctness, but to systematically uncover behavioral discrepancies and offer a structured means to reason about the system as a whole.

9.1 Future Work

While this thesis provides a concrete step toward applying MBT to adaptive microservice systems, it does not aim to be exhaustive. Several aspects remain unexplored or could be extended, creating opportunities for future research to enhance its applicability and effectiveness in broader contexts.

Automating Test Execution in CI/CD The current setup relies on manual test execution, which limits its use in realistic development workflows. A desired next step would be to integrate the testing process into a CI/CD pipeline. Concretely, after each redeployment and commit, a GitHub Action could trigger a new test run through AMP's HTTP API. If the verdict is *pass*, the workflow continues, but if not, a new GitHub Issue could be automatically created, attaching the adapter logs and test run results as artifacts. This would allow systematic evaluation of both applicability and efficiency in a continuous development setting.

Reducing Instrumentation Overhead The current instrumentation modifies the sidecar and several other points directly in the code. While effective, this approach requires maintenance whenever components change. A future improvement would be to explore even less invasive ways of collecting messages. For instance, instead of injecting function calls in every service, a single interception point closer to the network layer could be used, as long as it still captures both static and ephemeral components correctly.

Richer Models & Stronger Validation The current models can detect when behavior deviates from the specification but can often be less helpful in identifying the exact cause. They could therefore be extended with more validation of input parameters and bad weather cases, especially by exploring more invalid combinations of stimuli. Testing additional archetypes as they are introduced in DYNAMOS would also strengthen the completeness of the evaluation, as it would confirm that the modeling strategy remains applicable under different execution patterns. Together, these improvements would make the results more informative.

Deeper Exploration of Adaptive Behavior The adaptivity of the system is tested through static archetype selection. A focused model dedicated to adaptivity could be combined with test runs involving multiple simultaneous requests. This would allow observing whether archetype changes occur as expected and whether the system behaves correctly under higher load.

Generalization to Other Microservice Systems A natural next step would be to evaluate whether the findings of this work hold in other microservice systems. To achieve this, the adapter needs to be decoupled from DYNAMOS-specific implementation details, for instance by avoiding direct assumptions about message structures and service naming. On the modeling side, the set of labels and the conditions under which they are accepted should be defined in a way that reflects general interaction patterns rather than system-specific behaviors. These changes would allow the same methodology to be applied with minimal modifications, enabling a broader validation of the research findings.

Acknowledgments

First and foremost, I would like to express my sincere gratitude to my academic supervisor, Dr. Ana Oprescu, for her consistent guidance and thoughtful input throughout this thesis. Her comments were always focused and relevant, offering exactly the kind of direction I needed to move forward with confidence. Her involvement had a clear influence on the evolution and coherence of this work.

I would also like to thank my daily supervisor, Aleandro Mifsud, for his continued engagement and practical assistance across all stages of the project. His openness, careful observations, and constructive suggestions helped me tackle many of the key challenges. There were several points where his support made it possible to keep the work moving when progress would otherwise have stalled.

My appreciation extends to Axini for providing the opportunity to carry out this research within their organization. The positive and welcoming atmosphere created a space where I could work with focus, and the feedback I received along the way helped me navigate the more technical aspects of the project.

Finally, I am truly thankful to my family, friends, and partner for being present through every stage of this journey. Their encouragement helped me push through the difficult moments, and their belief in me gave me strength when I needed it most. I am especially grateful for their support and for enduring my ups and downs along the way, always with kindness and without complaint. Without them, none of this would have been possible.

Bibliography

[1] S. Newman, *Building Microservices*, 2nd. O'Reilly Media, Inc., Aug. 2021, p. 612, ISBN: 978-1-492-03402-5. [Online]. Available: https://www.oreilly.com/library/view/building-microservices-2nd/9781492034018/.

[2] M. Fowler and J. Lewis, *Microservices*, Mar. 2014. [Online]. Available: https://martinfowler.com/articles/microservices.html.

[3] A. A. L. Fernandes, "Automation of Microservices Testing: Effective Strategies and Tools," *International Journal of Science and Research (IJSR)*, vol. 13, no. 9, pp. 29–37, Sep. 2024, ISSN: 2319-7064. DOI: 10.21275/ES24809100005. [Online]. Available: https://www.ijsr.net/archive/v13i9/ES24809100005.pdf.

[4] M. Waseem, P. Liang, G. Márquez, and A. Di Salle, "Testing Microservices Architecture-based Applications: A Systematic Mapping Study," in *Proceedings of the 2020 27th Asia-Pacific Software Engineering Conference (APSEC)*, IEEE, 2020, pp. 119–128. DOI: 10.1109/APSEC51365.2020.00020. [Online]. Available: https://ieeexplore.ieee.org/document/9359275.

[5] B. van den Brink, T. Zameni, U. Tikhonova, L. Vinke, and A.-M. Oprescu, "Leveraging composability in model-based testing for microservices," in *Proceedings of the 21st Belgium-Netherlands Software Evolution Workshop (BENEVOL '22)*, CEUR-WS.org, Sep. 2022, p. 11. [Online]. Available: https://benevol2022.github.io/papers/BasVanDenBrink.pdf.

[6] J. J. Booy, *Model-Based Mocking: Compositional Model-Based Testing for Microservices using the Axini Modeling Platform*, Nov. 2023. [Online]. Available: https://research.tue.nl/files/319274449/Booy_J.pdf.

[7] P. Ganzevoort, *Model-Based Testing & Microservices: Examining challenges through integration with DYNAMOS*, Jan. 2025. [Online]. Available: https://dspace.uba.uva.nl/server/api/core/bitstreams/5a15ac8e-bfc7-4e00-b111-e4c403450c6e/content.

[8] N. Dragoni *et al.*, "Microservices: Yesterday, Today, and Tomorrow," in *Present and Ulterior Software Engineering*, Springer International Publishing, 2017, pp. 195–216, ISBN: 978-3-319-67425-4. DOI: 10.1007/978-3-319-67425-4{_}12. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-67425-4_12.

[9] *O'Reilly's Microservices Adoption in 2020 Report Finds that 92% of Organizations are Experiencing Success with Microservices*, Jul. 2020. [Online]. Available: https://www.oreilly.com/pub/pr/3307.

[10] *Do you utilize microservices within your organization?* 2021. [Online]. Available: https://www.statista.com/statistics/1236823/microservices-usage-per-organization-size/.

[11] P. Jamshidi, C. Pahl, N. C. Mendonca, J. Lewis, and S. Tilkov, "Microservices: The Journey So Far and Challenges Ahead," *IEEE Software*, vol. 35, no. 3, pp. 24–35, May 2018, ISSN: 1937-4194. DOI: 10.1109/MS.2018.2141039. [Online]. Available: https://ieeexplore.ieee.org/document/8354433.

[12] M. Waseem, P. Liang, M. Shahin, A. Di Salle, and G. Márquez, "Design, monitoring, and testing of microservices systems: The practitioners' perspective," *Journal of Systems and Software*, vol. 182, p. 44, Dec. 2021, ISSN: 0164-1212. DOI: 10.1016/j.jss.2021.111061. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121221001588.

[13] J. Figueira and C. Coutinho, "Developing self-adaptive microservices," *Procedia Computer Science*, vol. 232, pp. 264–273, Jan. 2024, ISSN: 1877-0509. DOI: 10.1016/J.PROCS.2024.01.026. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050924000267?via%3Dihub.

[14] M. Perry, *Testing Microservices at Scale: Using Ephemeral Environments*, Sep. 2023. [Online]. Available: https://www.qovery.com/blog/testing-microservices-at-scale-using-ephemeral-environments/.

[15] S. Whitesell, R. Richardson, and M. D. Groves, "Testing Microservices," in *Pro Microservices in .NET 6*, Apress, Berkeley, CA, 2022, pp. 171–207, ISBN: 978-1-4842-7833-8. DOI: 10.1007/978-1-4842-7833-8{_}7. [Online]. Available: https://link.springer.com/chapter/10.1007/978-1-4842-7833-8_7.

[16] M. Cohn, *Succeeding with Agile*. Addison-Wesley Professional, Oct. 2009, p. 504, ISBN: 9780321579362. [Online]. Available: https://www.oreilly.com/library/view/succeeding-with-agile/9780321660534/.

[17] MTS, *Methods for Testing and Specification (MTS); Model-Based Testing (MBT); Requirements for Modelling Notations*, Jul. 2011. [Online]. Available: https://www.etsi.org/deliver/etsi_es/202900_202999/202951/01.01.01_60/es_202951v010101p.pdf.

[18] J. Tretmans, "Model Based Testing with Labelled Transition Systems," in *Formal Methods and Testing*, 1st, vol. 4949, Springer, Berlin, Heidelberg, 2008, p. 38, ISBN: 978-3-540-78917-8. DOI: 10.1007/978-3-540-78917-8{_}1. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-540-78917-8_1.

[19] B. Legeard, "Model-based Testing: Next Generation Functional Software Testing," in *Dagstuhl Seminar Proceedings (DagSemProc)*, Jun. 2010, p. 13. DOI: 10.4230/DagSemProc.10111.9. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/DagSemProc.10111.9.

[20] A. Kramer and B. Legeard, *Model-Based Testing Essentials: Guide to the ISTQB® Certified Model-Based Tester: Foundation Level*. John Wiley & Sons, Inc, Apr. 2016, p. 276, ISBN: 9781119130161. DOI: 10.1002/9781119130161. [Online]. Available: https://onlinelibrary.wiley.com/doi/book/10.1002/9781119130161.

[21] T. Ruys, *Model Based Testing with the Axini Modeling Platform*, 2024.

[22] T. Ruys and M. van der Bijl, "Effective Model-Based Testing," in *Principles of Verification: Cycling the Probabilistic Landscape*, ser. Lecture Notes in Computer Science, vol. 15262, Springer, Cham, 2025, pp. 234–251, ISBN: 978-3-031-75778-5. DOI: 10.1007/978-3-031-75778-5. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-031-75778-5_11.

[23] M. Mlynarski, B. Güldali, and S. Weißleder, "Model-Based Testing: Achievements and Future Challenges," *Advances in Computers*, vol. 86, p. 39, 2012, ISSN: 0065-2458. DOI: 10.1016/B978-0-12-396535-6.00001-6. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780123965356000016.

[24] M. Zafar, W. Afzal, E. Enoiu, A. Stratis, A. Arrieta, and G. Sagardui, "Model-Based Testing in Practice: An Industrial Case Study using GraphWalker," in *ISEC '21: Proceedings of the 14th Innovations in Software Engineering Conference (formerly known as India Software Engineering Conference)*, Association for Computing Machinery, Apr. 2021, p. 11, ISBN: 978-1-4503-9046-0. DOI: 10.1145/3452383.3452388. [Online]. Available: https://dl.acm.org/doi/10.1145/3452383.3452388.

[25] S. Shakeri, L. Veen, and P. Grosso, "Evaluation of Container Overlays for Secure Data Sharing," in *2020 IEEE 45th LCN Symposium on Emerging Topics in Networking (LCN Symposium)*, Institute of Electrical and Electronics Engineers Inc., Nov. 2020, pp. 99–108, ISBN: 978-1-7281-8314-5. DOI: 10.1109/LCNSYMPOSIUM50271.2020.9363266. [Online]. Available: https://ieeexplore.ieee.org/document/9363266.

[26] J. Stutterheim, A. Mifsud, and A. Oprescu, "DYNAMOS: Dynamically Adaptive Microservice-based OS A Middleware for Data Exchange Systems," in *21st IEEE International Conference on Software Architecture*, Jun. 2024, p. 82. [Online]. Available: https://conf.researchr.org/details/icsa-2024/icsa-2024-software-architecture-in-practice-track/1/DYNAMOS-dynamic-microservice-composition-for-data-exchange-systems-lessons-learned.

[27] *Axini Modeling Platform - Documentation*. [Online]. Available: https://docs.axini.com/tech/.

[28] I. Ghani, W. M. Wan-Kadir, A. Mustafa, and M. I. Babir, "Microservice Testing Approaches: A Systematic Literature Review," *International Journal of Integrated Engineering*, vol. 11, no. 8, pp. 65–80, Dec. 2019, ISSN: 2600-7916. DOI: 10.30880/ijie.2019.11.08.008. [Online]. Available: https://publisher.uthm.edu.my/ojs/index.php/ijie/article/view/3856.

[29] V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. Reiter, and V. Sekar, "Gremlin: Systematic Resilience Testing of Microservices," in *2016 IEEE 36th International Conference on Distributed Computing Systems*, 2016, pp. 57–66. DOI: `10.1109/ICDCS.2016.11`. [Online]. Available: `https://ieeexplore.ieee.org/document/7536505`.

[30] M. Camilli, A. Janes, and B. Russo, "Automated test-based learning and verification of performance models for microservices systems," *The Journal of Systems & Software*, vol. 187, p. 22, May 2022, ISSN: 0164-1212. DOI: `10.1016/j.jss.2022.111225`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0164121222000061`.

[31] L. Gazzola *et al.*, "ExVivoMicroTest: ExVivo Testing of Microservices," *Journal of Software: Evolution and Process*, vol. 35, no. 4, p. 23, Apr. 2022. DOI: `10.1002/smr.2452`. [Online]. Available: `https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2452`.

[32] C. F. Wu, S. P. Ma, A. C. Shau, and H. W. Yeh, "Testing for Event-Driven Microservices Based on Consumer-Driven Contracts and State Models," in *2022 29th Asia-Pacific Software Engineering Conference (APSEC)*, vol. 2022-December, IEEE, 2022, pp. 467–471, ISBN: 9781665455374. DOI: `10.1109/APSEC57359.2022.00064`. [Online]. Available: `https://ieeexplore.ieee.org/document/10043304`.

[33] A. Marques, F. Ramalho, and W. L. Andrade, "Comparing Model-Based Testing with Traditional Testing Strategies: An Empirical Study," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, IEEE, 2014, pp. 264–273, ISBN: 978-1-4799-5790-3. DOI: `10.1109/ICSTW.2014.29`. [Online]. Available: `https://ieeexplore.ieee.org/document/6825672`.

[34] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing approaches," *Journal of Software Testing, Verification and Reliability*, vol. 22, no. 5, pp. 297–312, 2011, ISSN: 1099-1689. DOI: `10.1002/stvr.456`. [Online]. Available: `https://onlinelibrary.wiley.com/doi/full/10.1002/stvr.456`.

[35] J. Tretmans and P. van de Laar, "Model-Based Testing with TorXakis: The Mysteries of Dropbox Revisited," in *Proceedings of the Central European Conference on Information and Intelligent Systems*, Zagreb : Faculty of Organization and Informatics, University of Zagreb, Oct. 2019, pp. 247–258. [Online]. Available: `https://repository.ubn.ru.nl/handle/2066/214659`.

[36] R. Wang, Y. Li, H. Xie, Y. Xu, and J. C. S. Lui, "GraphWalker: An I/O-Efficient and Resource-Friendly Graph Analytic System for Fast and Scalable Random Walks," in *Proceedings of the 2020 USENIX Annual Technical Conference*, Jul. 2020, pp. 559–571, ISBN: 978-1-939133-14-4. DOI: `10.5555/3489146.3489184`. [Online]. Available: `https://dl.acm.org/doi/10.5555/3489146.3489184`.

[37] J. Tretmans, "Test generation with inputs, outputs, and quiescence," in *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 1055, Springer, Berlin, Heidelberg, 1996, pp. 127–146, ISBN: 978-3-540-49874-2. DOI: `10.1007/3-540-61042-1{\_}42`. [Online]. Available: `https://link.springer.com/chapter/10.1007/3-540-61042-1_42`.

Acronyms

AMdEX Amsterdam Data Exchange. 12
AML Axini Modeling Language. 1, 10, 34, 37, 54, 59
AMP Axini Modeling Platform. 1, 5, 10, 11, 19, 20, 27–32, 35, 37, 39, 42, 45, 46, 50, 53, 58–60
AMQP Advanced Message Queuing Protocol. 1, 7, 13, 17, 19, 23, 25, 26, 31, 33, 34, 36, 52, 54
API Application Programming Interface. 4, 8, 17
ASCII American Standard Code for Information Interchange. 25
BFF Backend for Frontend. 6
CI/CD Continuous Integration/Continuous Delivery. 54, 60
DAG Directed Acyclic Graph. 13, 26, 38
DDM Digital Data Marketplace. 12
DORA DevOps Research and Assessment. 6
DSL Domain-Specific Language. 10
E2E End-to-End. 1, 8, 16, 18, 26, 33, 52, 55, 57, 59
ESB Enterprise Service Bus. 6
ETSI European Telecommunications Standards Institute. 8
FSM Finite State Machine. 8, 28
HTTP Hypertext Transfer Protocol. 7, 13, 17, 26, 29–31, 33–36, 38, 40, 41, 49, 50, 53, 54, 60
IOCO Input-Output Conformance. 9, 10, 58
JSON JavaScript Object Notation. 25, 29–31, 35, 53
LTS Labeled Transition System. 8, 9
MBT Model-based Testing. 1, 4–6, 8–12, 16–19, 21, 22, 40, 42, 43, 51, 52, 54, 55, 57–59
MSA Microservices Architecture. 1, 4, 6–8, 11, 57
RPC Remote Procedure Call. 14
SOC Service-oriented Computing. 6
SQL Structured Query Language. 15, 26, 37, 40, 43
STS Symbolic Transition System. 8–10, 19, 21, 39, 50, 58
SUT System Under Test. 1, 8–12, 16–20, 22, 23, 27, 28, 32, 33, 43, 45, 50–53, 58
TTP Trusted Third Party. 15
UI User Interface. 17
UNL University of the Netherlands. 12, 14
URL Uniform Resource Locator. 32
WSL Windows Subsystem for Linux. 42

Appendix A

Primary AML Models

The full AML models can be found at: `https://github.com/michalispap/MBT-Models`

```
# computeToData archetype.

external 'dynamos_channel'
internal 'sync_channel'

include 'scenarios/archetype_selected.aml'
include 'scenarios/request_validated.aml'
include 'scenarios/chain_microservices_finished.aml'

include 'shared/shared_functions.aml'

# Override default configuration.
include 'shared/default_config'
@config[:selected_weight] = 100
@config[:aggregate] = false
@config[:implementation] = :COMPUTETODATA

# process('dynamos_http_flow')
include 'shared/shared_process.aml'

process('dynamos_rabbitmq_flow') {

  timeout 20
  include 'shared/interface.aml'

  # Internal channel.
  include 'shared/shared_internal_channel.aml'

  include 'shared/shared_variables.aml'

  include 'shared/shared_behaviors.aml'

  # Notify upon chain microservice processing termination.
  behavior('send_chain_microservice_finished') {
    send 'queryFinished', constraint: 'query_count < length(validated_providers)',
    ↪ update: 'query_count += 1', note: ["#indigo Query microservice finished.",
    ↪ "#green Inside chain: query -> next microservice"]
    optionally { send 'anonymizeFinished', constraint: 'is_anonymize_enabled &&
    ↪ anonymize_count < length(validated_providers)', update: 'anonymize_count +=
    ↪ 1', note: ["#indigo Anonymize microservice finished.", "#green Inside chain:
    ↪ anonymize -> next microservice"] }
```

```
    send 'algorithmFinished', constraint: 'algorithm_count <
    ↪  length(validated_providers)', update: 'algorithm_count += 1', note: ["#indigo
    ↪  Algorithm microservice finished.", "#green Inside chain: algorithm -> end"]
  }

  call 'receive_and_approve_request'

  repeat {

    o { send 'sqlDataRequest', note: "#indigo API Gateway forwards SQL query to
    ↪  Agent." }
    o { send 'microserviceCommunication', constraint: "result == '' && return_address
    ↪  in queues && !(return_address in responded_queues)", update:
    ↪  'responded_queues += [return_address]', note: "#indigo Agent communicates
    ↪  with Microservice Chain." }
    o { call 'send_orchestrator_or_microservice_chain' }
    # Do not expect additional messages.
    o {
      constraint "number_of_composition_requests == length(validated_providers) &&
      ↪  number_of_request_approval_responses == 1 && length(responded_queues) ==
      ↪  length(validated_providers) && compare_queues(queues, validated_providers)
      ↪  && compare_queues(responded_queues, validated_providers) && response_error
      ↪  == '' && query_count == number_of_composition_requests && algorithm_count
      ↪  == number_of_composition_requests && anonymize_count <=
      ↪  length(validated_providers)"
      update "expected_number_of_responses = number_of_composition_requests"
      stop_repetition
    }

  }

  call 'send_finished_to_http'

}
```

Listing 11: computeToData.aml

```
# dataThroughTtp archetype.

external 'dynamos_channel'
internal 'sync_channel'

include 'scenarios/archetype_selected.aml'
include 'scenarios/request_validated.aml'
include 'scenarios/chain_microservices_finished.aml'

include 'shared/shared_functions.aml'

# Override default configuration.
include 'shared/default_config'
@config[:selected_weight] = 300
@config[:aggregate] = true
@config[:implementation] = :DATATHROUGHTTP

# process('dynamos_http_flow')
include 'shared/shared_process.aml'
```

```
process('dynamos_rabbitmq_flow') {

  timeout 20
  include 'shared/interface.aml'

  # Internal channel.
  include 'shared/shared_internal_channel.aml'

  include 'shared/shared_variables.aml'

  include 'shared/shared_behaviors.aml'

  # Notify upon chain microservice processing termination.
  behavior('send_chain_microservice_finished') {
    choice {
      o { send 'queryFinished', constraint: 'query_count <
      ↪ length(validated_providers)', update: 'query_count += 1', note: ["#indigo
      ↪ Query microservice finished.", "#green Inside chain: query -> next
      ↪ microservice"] }
      o { send 'anonymizeFinished', constraint: 'is_anonymize_enabled &&
      ↪ anonymize_count < length(validated_providers)', update: 'anonymize_count +=
      ↪ 1', note: ["#indigo Anonymize microservice finished.", "#green Inside
      ↪ chain: anonymize -> next microservice"] }
      o { send 'aggregateFinished', constraint: 'is_aggregate_enabled &&
      ↪ aggregate_count < 1', update: 'aggregate_count += 1', note: ["#indigo
      ↪ Aggregate microservice finished.", "#green Inside chain: aggregate -> next
      ↪ microservice"] }
      o { send 'algorithmFinished', constraint: 'algorithm_count < 1', update:
      ↪ 'algorithm_count += 1', note: ["#indigo Algorithm microservice finished.",
      ↪ "#green Inside chain: algorithm -> end"] }
    }
  }

  call 'receive_and_approve_request'

  repeat {

    o { send 'sqlDataRequest', constraint: 'number_of_sqlDataRequest_responses <=
    ↪ length(validated_providers)', update: 'number_of_sqlDataRequest_responses +=
    ↪ 1', note: "#indigo API Gateway forwards SQL query to Agent." }
    o { send 'microserviceCommunication', constraint: "result == '' && return_address
    ↪ in queues", note: "#indigo Agent communicates with Microservice Chain or
    ↪ another Agent." }
    o { call 'send_orchestrator_or_microservice_chain' }
    # Do not expect additional messages.
    o {
      constraint "number_of_composition_requests == length(validated_providers) + 1
      ↪ && number_of_request_approval_responses == 1 && compare_queues(queues -
      ↪ ['SURF-in'], validated_providers) && response_error == '' && query_count ==
      ↪ length(validated_providers) && anonymize_count <=
      ↪ length(validated_providers) && algorithm_count == 1 && aggregate_count ==
      ↪ 1"
      update "expected_number_of_responses = length(responded_queues_with_response)"
      stop_repetition
    }

  }
```

```
  call 'send_finished_to_http'

}
```

Listing 12: dataThroughTtp.aml

```
# Common process used by computeToData and dataThroughTtp archetypes.

include 'shared/shared_macros.aml'

# Override configuration.
@config[:bad_weather] = false
@config[:ugly_weather] = false

process('dynamos_http_flow') {

  timeout 20
  include 'shared/interface.aml'

  # Internal channel.
  channel('sync_channel') {
    stimulus 'expectations_ready', { "_count" => :integer }
    response 'http_request_received'
  }

  # Configuration is overriden in implementation/archetype model.
  var 'selected_weight', :integer, @config.fetch(:selected_weight)
  var 'aggregate', :boolean, @config.fetch(:aggregate)

  var 'expected_responses_from_sync', :integer
  var 'observed_responses', :integer

  # Perform initial client HTTP POST request.
  def perform_sql_data_request(request_properties)
    receive 'sql_data_request', constraint: request_properties, note: "#blue API
    ↪  Gateway receives initial client request."
  end

  # Receive HTTP PUT request to activate archetype change.
  # computeToData weight: 100, dataThroughTtp weight: 300
  receive 'switch_archetype', constraint: 'weight == selected_weight', note: "#blue
  ↪  Select archetype."

  # Archetype weight may be incorrect.
  if bad_weather?
    optionally { send 'http_response_status', constraint: '!(selected_weight in [100,
    ↪  300])', note: "#red Archetype weight is incorrect." }
  end

  # Archetype weight is correct.
  constraint 'selected_weight in [100, 300]'

  # Send HTTP PUT response status code.
  send 'http_response_status', note: "#blue Server response code: $code"

  # Receive HTTP POST request to trigger SUT.
```

```
  perform_sql_data_request(
    "request_type == 'sqlDataRequest' && " \
    "user['id'] == '12324' && " \
    "user['userName'] == 'jorrit.stutterheim@cloudnation.nl' && " \
    "(dataProviders == ['VU','UVA','RUG'] || dataProviders == ['UVA']) && " \
    "data_request['type'] == 'sqlDataRequest' && " \
    "data_request['query'] == 'SELECT * FROM Personen p JOIN Aanstellingen s LIMIT
    ↪  1000' && " \
    "data_request['algorithm'] == 'average' && " \
    "data_request['options']['graph'] == false && " \
    "data_request['options']['aggregate'] == aggregate && " \
    "(data_request['options']['anonymize'] == true ||
    ↪  data_request['options']['anonymize'] == false)"
  )

  # Signal RabbitMQ flow that the HTTP POST request is received and it can begin its
  ↪  sequence.
  send 'http_request_received', on: 'sync_channel', note: "#blue Signal RabbitMQ flow
  ↪  to start receiving."

  choice {

    o {
      # RabbitMQ flow is complete. Receive number of expected responses.
      receive 'expectations_ready', on: 'sync_channel', update:
      ↪  'expected_responses_from_sync = _count', note: "#blue Receive expected
      ↪  response number from RabbitMQ flow."

      # Send HTTP POST response status code.
      send 'http_response_status', note: "#blue Server response code: $code"

      # Send results to client.
      send 'results', constraint: 'length(responses) ==
      ↪  expected_responses_from_sync', update: 'observed_responses =
      ↪  length(responses)', note: '#blue Number of responses: $observed_responses'
    }
    # RabbitMQ flow may be interrupted prematurely.
    if bad_weather?
      o {
        send 'http_response_status'
        send 'results', constraint: 'is_empty(responses) || include(responses, "")',
        ↪  note: '#red Internal system error. Empty results.'
      }
    end

  }

}
```

Listing 13: shared_process.aml

```
# Stimuli and response labels and respective parameters.

channel('dynamos_channel') {

  def sql_data_request_properties
    {
```

```
    "request_type" => :string,
    'user' => {
      'id' => :string,
      'userName' => :string
    },
    'dataProviders' => [:string],
    'data_request' => {
      'type' => :string,
      'query' => :string,
      'algorithm' => :string,
      'options' => {
        'graph' => :boolean,
        'aggregate' => :boolean,
        'anonymize' => :boolean
      },
      'requestMetadata' => {}
    }
  }
end

def switch_archetype_properties
  {"weight" => :integer}
end

def request_approval_properties
  { "data_providers" => [:string], "options" => { "graph" => :boolean, "aggregate"
  ↪  => :boolean, "anonymize" => :boolean } }
end

def validation_response_properties
  { "valid_dataproviders" => [:string], "invalid_dataproviders" => [:string],
  ↪  "request_approved" => :boolean }
end

def composition_request_properties
  { "archetype_id" => :string, "role" => :string, "destination_queue" => :string }
end

def request_approval_response_properties
  { "error" => :string }
end

def microservice_communication_properties
  { "return_address" => :string, "result" => :string }
end

def results_properties
  { "jobId" => :string, "responses" => [:string] }
end

def http_response_status_properties
  { "code" => :integer }
end

def sql_data_request__response_properties
  {"query" => :string}
end
```

```
  stimulus 'sql_data_request', sql_data_request_properties
  stimulus 'switch_archetype', switch_archetype_properties
  stimulus 'reset'
  response 'requestApproval', request_approval_properties
  response 'validationResponse', validation_response_properties
  response 'compositionRequest', composition_request_properties
  response 'requestApprovalResponse', request_approval_response_properties
  response 'microserviceCommunication', microservice_communication_properties
  response "anonymizeFinished"
  response "algorithmFinished"
  response "aggregateFinished"
  response "queryFinished"
  response "sqlDataRequest", sql_data_request__response_properties
  response 'http_response_status', http_response_status_properties
  response 'results', results_properties
}
```

Listing 14: interface.aml

```
# Common variables used by process('dynamos_rabbitmq_flow') in computeToData and
↪  dataThroughTtp archetypes.

var 'all_data_providers', [:string], ['UVA','VU']
var 'validated_providers', [:string]

var 'query_count', :integer, 0
var 'anonymize_count', :integer, 0
var 'algorithm_count', :integer, 0
var 'aggregate_count', :integer, 0

var 'is_anonymize_enabled', :boolean, false
var 'is_aggregate_enabled', :boolean, false

var 'approved', :boolean

var 'archetypes', [:string], ['computeToData', 'dataThroughTtp']
var 'provider_roles', [:string], ['all', 'dataProvider', 'computeProvider']

var 'queues', [:string], []
var 'responded_queues', [:string], []
var 'responded_queues_with_response', [:string], []

var 'response_error', :string

var 'number_of_composition_requests', :integer, 0
var 'number_of_request_approval_responses', :integer, 0
var 'number_of_sqlDataRequest_responses', :integer, 0

var 'expected_number_of_responses', :integer
```

Listing 15: shared_variables.aml

```
# Common internal channel used by process('dynamos_rabbitmq_flow') in computeToData
↪  and dataThroughTtp archetypes.

channel('sync_channel') {
```

```
  response 'expectations_ready', { "_count" => :integer }
  stimulus 'http_request_received'
}
```

Listing 16: shared_internal_channel.aml

```
# Common behaviors used by computeToData and dataThroughTtp archetypes.

# Receive internal signal and approve request.
behavior('receive_and_approve_request') {

  # Receive signal from HTTP flow to start receiving RabbitMQ messages.
  receive 'http_request_received', on: 'sync_channel', note: "#indigo Receive signal
  ↪ from HTTP flow to start."
  send 'requestApproval', update: "is_anonymize_enabled = options['anonymize'];
  ↪ is_aggregate_enabled = options['aggregate']", note: '#indigo API Gateway
  ↪ forwards client request to Policy Enforcer for validation.'
  send 'validationResponse', constraint:
  ↪ 'check_valid_data_providers(valid_dataproviders, all_data_providers) &&
  ↪ check_invalid_data_providers(invalid_dataproviders, all_data_providers)',
  ↪ update: 'validated_providers = valid_dataproviders; approved =
  ↪ request_approved', note: ['#indigo Policy Enforcer replies to Orchestrator
  ↪ about the outcome of the request.', '#indigo Approved: $approved']

}

# Send message from Orchestrator.
behavior('orchestrator_send_in_any_order') {

  choice {
    o { send 'compositionRequest', constraint: 'archetype_id in archetypes && role in
    ↪ provider_roles && number_of_composition_requests <=
    ↪ length(validated_providers)', update: 'queues += [destination_queue];
    ↪ number_of_composition_requests += 1', note: '#indigo Orchestrator informs
    ↪ Agent about the job deployment.' }
    o { send 'requestApprovalResponse', constraint:
    ↪ 'number_of_request_approval_responses < 1', update: 'response_error = error;
    ↪ number_of_request_approval_responses += 1', note: "#indigo Orchestrator
    ↪ forwards Policy Enforcer's answer to API Gateway." }
  }

}

# Send message from Orchestrator or Microservice Chain.
behavior('send_orchestrator_or_microservice_chain') {

  choice {
  o { call 'orchestrator_send_in_any_order' }
  o { call 'send_chain_microservice_finished' }
  o { send 'microserviceCommunication', constraint: "result != '' && return_address
  ↪ in queues && !(return_address in responded_queues_with_response)", update:
  ↪ 'responded_queues_with_response += [return_address]', note: "#indigo
  ↪ Microservice Chain communicates with Agent." }
  }

}
```

```
# Signal HTTP flow to send final results.
behavior('send_finished_to_http') {

  # Send number of expected responses to HTTP flow.
  send 'expectations_ready', on: 'sync_channel', constraint: '_count ==
   ↪ expected_number_of_responses', note: "#indigo Signal HTTP flow. Expected
   ↪ responses: $expected_number_of_responses."

}
```

Listing 17: shared_behaviors.aml

```
# Common functions used by computeToData and dataThroughTtp archetypes.

# Ensure all given data providers are present in full list.
function('check_valid_data_providers', [[:string], [:string]] => :boolean) do
↪  |valid_dataproviders, all_data_providers|
  (valid_dataproviders - all_data_providers).empty?
end

# Ensure no invalid data provider is present in full list.
function('check_invalid_data_providers', [[:string], [:string]] => :boolean) do
↪  |invalid_dataproviders, all_data_providers|
  (invalid_dataproviders & all_data_providers).empty?
end

# Ensure providers match the validated ones.
function('compare_queues', [[:string], [:string]] => :boolean) do |queues,
↪  validated_providers|
  prefixes = queues.map { |q| q.split('-').first }
  validated_providers.sort == prefixes.sort
end
```

Listing 18: shared_functions.aml

```
def bad_weather?
  @config.fetch(:bad_weather)
end

def ugly_weather?
  @config.fetch(:ugly_weather)
end
```

Listing 19: shared_macros.aml

```
@config = {
  selected_weight: 100,
  aggregate: false,
  implementation: nil,
  bad_weather:  true,
  ugly_weather: true
}
```

Listing 20: default_config