

Metamodel Definition

Fig. 1 illustrates the components of the meta-model generated during the first phase of DAMP’s execution. The meta-model contains all the essential information about a microservice-based system required for accurately detecting microservice anti-patterns. It includes information at both the system level and the individual microservice level. At the system level, the meta-model captures key elements such as the total number of microservices, system-wide dependency files, configuration files, environment files, and deployment files.

At the microservice level, the meta-model provides more granular details, including source code elements used to extract provided URIs, class fields, methods, provided APIs, and existing annotations. It also includes information about the type of database used by each microservice, the type of HTTP requests, imported libraries, and Docker images.

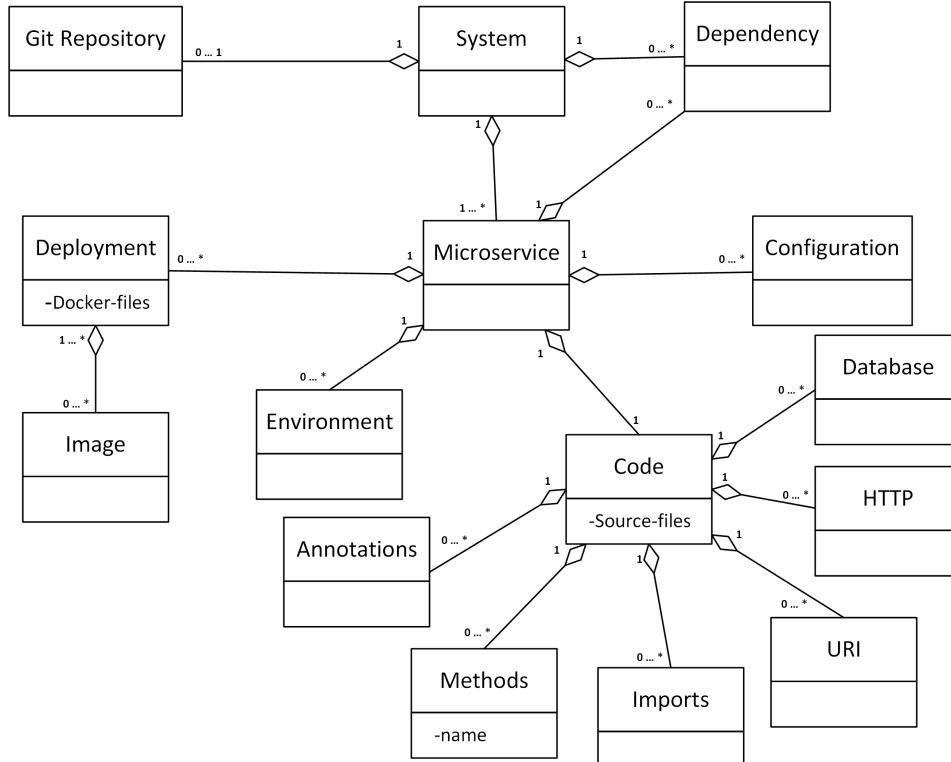


Fig. 1: Components of the Meta-model generated by DAMP.

The use of a meta-model offers a key advantage to the detection algorithm by enabling access to all necessary information without relying on system resources at runtime. The introduction of the meta-model facilitates the extension of DAMP to support the detection of additional anti-patterns and the incorporation of new structural components.

Furthermore, relying on the meta-model ensures that the execution of DAMP remains technology-agnostic. For example, by abstracting system-level and microservice-level dependencies into a unified **Dependency** component, the solution becomes independent of specific build tools such as Maven, Gradle, or others.

The **Configuration** component is essential for detecting various anti-patterns. It enables the identification of endpoints by searching for URLs within configuration files (for detecting Hardcoded Endpoints), the extraction of CI/CD indicators such as “CD Pipeline” or “Java CI” (for detecting No CI/CD), and the discovery of versioning details (for detecting No API Versioning). These capabilities, in conjunction with the **Code** and **Dependency** components, are critical for accurate anti-pattern detection.

The model defined in DAMP is composed of fundamental components organized hierarchically. At the highest level, the **System** component serves as the root of the model. This component is initialized by importing a microservice system from either a Git repository or a local directory specified as input to DAMP.

As the top-level entity in the model, the **System** component encompasses several subcomponents. These subcomponents capture key information about the system, including the individual microservices (represented by the **Microservice** component), the interactions between microservices along with their HTTP method types (e.g., POST, PUT, GET, DELETE), the system’s dependency files (captured in the **Dependencies** component), and configuration files (represented by the **Configuration** component). In the following each of these subcomponents is described in greater detail.

Microservice. This component encapsulates comprehensive information about each actual microservice within the system. This information is structured using various subcomponents, each storing key-value pairs that represent relevant characteristics of the microservice. These subcomponents include the service name, number of lines of code, programming language,

dependencies, imports, annotations, declared-annotations, methods, HTTP requests, databases used, configuration files, Docker files, images, environment files, and provided APIs.

Imports. This component stores all import statements found within the source code files of the microservice.

Annotations. This component contains a complete list of annotations used in the microservice's source code.

Declared-Annotations. If the developer has defined custom annotations (e.g., within interface files), these are automatically identified by DAMP and stored under this subcomponent.

Methods. This subcomponent includes a list of method names used in the microservice code, along with their respective return types.

HTTP. All HTTP requests defined in the source code of the microservice are collected and listed in this subcomponent.

Database. If the microservice interacts with a database, this subcomponent stores details such as the database type and name.

Dependencies. The Dependencies component is utilized at both the System and Microservice levels, as dependency files are typically defined for both scopes using build tools such as Maven or Gradle. This component represents the dependencies used by the system and its microservices to achieve their intended functionalities, such as monitoring, load balancing, and other operational aspects. Additionally, it may include dependencies on core internal services within the system.

Configuration. The Configuration component also exists as a subcomponent under both the System and Microservice components, containing critical information extracted from configuration files. Information from this component is vital for detecting certain anti-patterns. For example, it includes settings related to hardcoded endpoints. Furthermore, versioning of projects, particularly in microservice systems that utilize message queuing, is generally handled through these configuration files.

Environment. The Environment component contains data related to environment variables, which are typically used for the dynamic injection of variables into a system during runtime.

Deployment. The Deployment component refers to files that include information necessary for the deployment and maintenance of microservices. This includes configuration files such as Dockerfile and docker-compose.yml, which hold data essential for microservice deployment purposes. The Deployment component is defined as a subcomponent under both the System and Microservice components and is populated as a dictionary containing the required deployment-related information.