TYPHON
POLYGLOT AND HYBRID PERSISTENCE ARCHITECTURES FOR BIG DATA ANALYTICS

Project Number 780251

D3.4 Hybrid Polystore Deployment Language (Final Version)

Version 1.0
9 July 2020
Final

Public Distribution

ATB

Project Partners: **Alpha Bank, ATB, Centrum Wiskunde & Informatica, CLMS, Edge Hill University, GMV, OTE, SWAT.Engineering, The Open Group, University of L'Aquila, University of Namur, University of York, Volkswagen**

PROJECT PARTNER CONTACT INFORMATION

Alpha Bank Vasilis Kapordelis 40 Stadiou Street 102 52 Athens Greece Tel: +30 210 517 5974 E-mail: vasileios.kapordelis@alpha.gr	**ATB** Sebastian Scholze Wiener Strasse 1 28359 Bremen Germany Tel: +49 421 22092 0 E-mail: scholze@atb-bremen.de
Centrum Wiskunde & Informatica Tijs van der Storm Science Park 123 1098 XG Amsterdam Netherlands Tel: +31 20 592 9333 E-mail: storm@cwi.nl	**CLMS** Antonis Mygiakis Mavrommataion 39 104 34 Athens Greece Tel: +30 210 619 9058 E-mail: a.mygiakis@clmsuk.com
Edge Hill University Yannis Korkontzelos St Helens Road Ormskirk L39 4QP United Kingdom Tel: +44 1695 654393 E-mail: yannis.korkontzelos@edgehill.ac.uk	**GMV Aerospace and Defence** Almudena Sánchez González Calle Isaac Newton 11 28760 Tres Cantos Spain Tel: +34 91 807 2100 E-mail: asanchez@gmv.com
OTE Theodoros E. Mavroeidakos 99 Kifissias Avenue 151 24 Athens Greece Tel: +30 697 814 7618 E-mail: tmavroeid@ote.gr	**SWAT.Engineering** Davy Landman Science Park 123 1098 XG Amsterdam Netherlands Tel: +31 633754110 E-mail: davy.landman@swat.engineering
The Open Group Scott Hansen Rond Point Schuman 6, 5th Floor 1040 Brussels Belgium Tel: +32 2 675 1136 E-mail: s.hansen@opengroup.org	**University of L'Aquila** Davide Di Ruscio Piazza Vincenzo Rivera 1 67100 L'Aquila Italy Tel: +39 0862 433735 E-mail: davide.diruscio@univaq.it
University of Namur Anthony Cleve Rue de Bruxelles 61 5000 Namur Belgium Tel: +32 8 172 4963 E-mail: anthony.cleve@unamur.be	**University of York** Dimitris Kolovos Deramore Lane York YO10 5GH United Kingdom Tel: +44 1904 325167 E-mail: dimitris.kolovos@york.ac.uk
Volkswagen Behrang Monajemi Berliner Ring 2 38440 Wolfsburg Germany Tel: +49 5361 9-994313 E-mail: behrang.monajemi@volkswagen.de	

DOCUMENT CONTROL

Version	Status	Date
0.1	Template creation	17.06.2020
0.2	Refinement of the document structure and first content	25.06.2020
0.3	First complete set of contributions collected	29.06.2020
0.4	Final version ready for internal proof-reading	01.07.2020
0.5	Sent out to project partners for external proof-reading	02.07.2020
0.6	Contributions from project partners integrated	06.07.2020
0.7	Additional updates from partner reviews	08.07.2020
1.0	Final version for EC submission	09.07.2020

TABLE OF CONTENTS

TABLE OF FIGURES

TABLE OF LISTINGS

EXECUTIVE SUMMARY

This deliverable presents the final version of the TyphonDL modelling language. In particular it presents the language concepts and architecture, its meta-model as well as its implementation.

TyphonDL – Hybrid Polystore Deployment Language – is a modular language aiming to bridge the conceptual gap between high-level polystore design models (expressed in TyphonML and developed in Work Package 2), and low-level virtual image configuration and assembly tools such as Docker and Kubernetes.

In this deliverable an overview of the deployment technologies that were studied in the scope of the TyphonDL design and development is given and the requirements that were outlined for TyphonDL are revisited.

Further, the final version of TyphonDL is introduced, in particular the language architecture along with language syntax and implementation using the EMF-based technologies. Furthermore, an example TyphonDL model is illustrated in concrete syntax.

The deliverable is concluded with an evaluation of the requirements outlined for TyphonDL together with major points with respect to the strengths and limitations of the language as well as its future outlook.

1. INTRODUCTION

1.1 OVERVIEW

This document presents the work done in task T3.1 Hybrid Polystore Deployment Language (TyphonDL) Design. The main objective of T3.1 is to define the syntax and the semantics of TyphonDL, a modular language aiming to bridge the conceptual gap between high-level polystore design models (expressed in TyphonML and developed in work package 2), and low-level virtual image configuration and assembly tools such as Docker-Compose[1] and Kubernetes[2] (see section 2). TyphonDL will be specified in an iterative manner by executing two subsequent steps:

- elicitation of new concepts from the domain of virtual image configuration and assembly tools domain, and
- validation of the elicited concepts by specifying concrete hybrid polystore deployments.

This document extends the first version of TyphonDL given in D3.1 and presents the final version of TyphonDL. It elaborates on the language features resulted from a progression --- starting from the analysis of the TyphonML development and the requirements collected during the requirements analysis phase of the project (Work Package 1) to create the syntax and semantics of TyphonDL.

1.2 STRUCTURE OF THE DELIVERABLE

The deliverable is structured as follows:

- Section 2 gives an overview of the deployment technologies that were studied in the scope of the TyphonDL design and development, and revisits the requirements that were outlined for TyphonDL.
- Section 3 introduces the language design aspects and architecture of TyphonDL,
- Section 4 presents the meta-model in particular its abstract syntax in detail,
- Section 5 focuses on the implementation of TyphonDL, in particular presenting the textual language in terms of concrete syntax, using the EMF-based technologies (Xtext[3]), and illustrates an example TyphonDL model given in concrete syntax, and presents validation support for TyphonDL implemented in Xtend[4],
- Section 6 concludes the document and evaluates the requirements for TyphonDL.

[1] https://docs.docker.com/compose
[2] https://kubernetes.io
[3] https://www.eclipse.org/Xtext/
[4] https://www.eclipse.org/xtend/

2. DEPLOYMENT TECHNOLOGIES

This section introduces the fundamental concepts in the context of deployment technologies and the state-of-the-art tools used for them.

2.1 OVERVIEW

An *application software* (or *application* for short) is a single or a group of software programs designed for end-users. In the context of deployment modelling in this document, the individual software programs that comprise an application will be referred to as *services*. Figure 1 illustrates a simple example of an application named Weather Warning application that consists of four databases services: Two database services based on MariaDB[5] and two database services based on MongoDB[6].

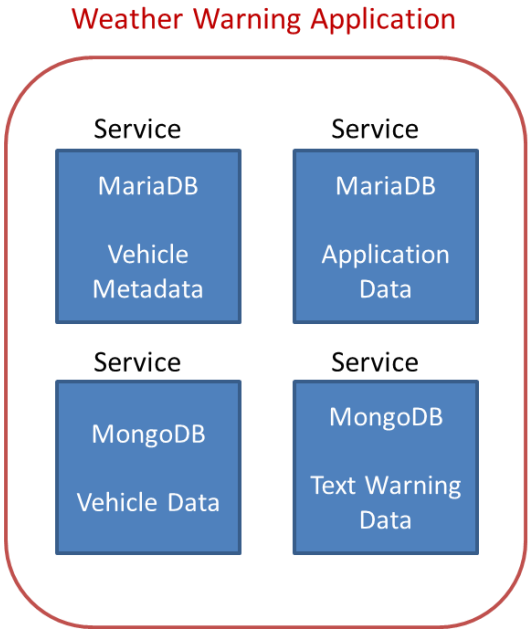

Figure 1: An Example Application

2.2 CLOUD PLATFORMS

Cloud platforms are platforms on the internet that provide cloud computing services and offer computation power, database storage, content delivery and other functionalities to businesses to support their product development. The cloud infrastructure is maintained by the platform provider and not by the individual platform user, which allows businesses and other application developers to focus completely on the product they are creating without any concerns on the underlying infrastructure to run their applications.

Cloud platforms offer a wide range of benefits from providing flexibility in terms of 1) the scaling of infrastructure on demand in order to accommodate for varying workload; 2) public, private or hybrid storage options to meet the required security standards; 3)

[5] www.mariadb.org
[6] www.mongodb.com

tool selection, to allow for accessibility of applications and data virtually from any device connected to the internet. This makes cloud platforms increasingly popular and useful.

Amazon Web Services[7], Microsoft Azure[8], Google Cloud[9], IBM Cloud[10] and Oracle Cloud[11] are amongst the most leading cloud platform providers.

2.3 CONTAINERISED APPLICATIONS USING DOCKER

A *container* is a standard unit of software that packages up code and all its dependencies, so the application runs quickly and reliably from one computing environment to another.

A Docker[12] container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings[13]. Contrary to virtual machines, multiple containers can share the OS kernel with other containers, thus taking up less space.

Images can be pulled from Docker-Hub[14], company specific registries[15] either local or externally accessible (authentication with apache and nginx possible) or the Docker Trusted Registry (DTR)[16], which is the enterprise-grade image storage solution from Docker.

Running multiple containers can be configured with the tool Docker-Compose[17]. All needed parts of an application (services) are defined in a YAML[18] file, which is used to create and start all needed containers with a single command.

In Docker-Compose, it is possible to specify the interplay between components in a deployment configuration – either linking to other services inside or even outside the *docker-compose.yaml*. Deployment properties (e.g., the computing power/number of CPUs to be used, amount of memory to be used, path to persistent storage, shared networks, etc.) can easily be added and edited.

The deployment of the explanatory Weather Warning application can be exemplified as in Figure 2:

[7] https://aws.amazon.com
[8] https://azure.microsoft.com
[9] https://cloud.google.com
[10] https://www.ibm.com/cloud/
[11] https://cloud.oracle.com
[12] https://www.docker.com/
[13] https://www.docker.com/resources/what-container
[14] https://hub.docker.com/
[15] https://docs.docker.com/registry/
[16] https://docs.docker.com/datacenter/dtr/2.1/guides/
[17] https://docs.docker.com/compose/
[18] https://yaml.org/

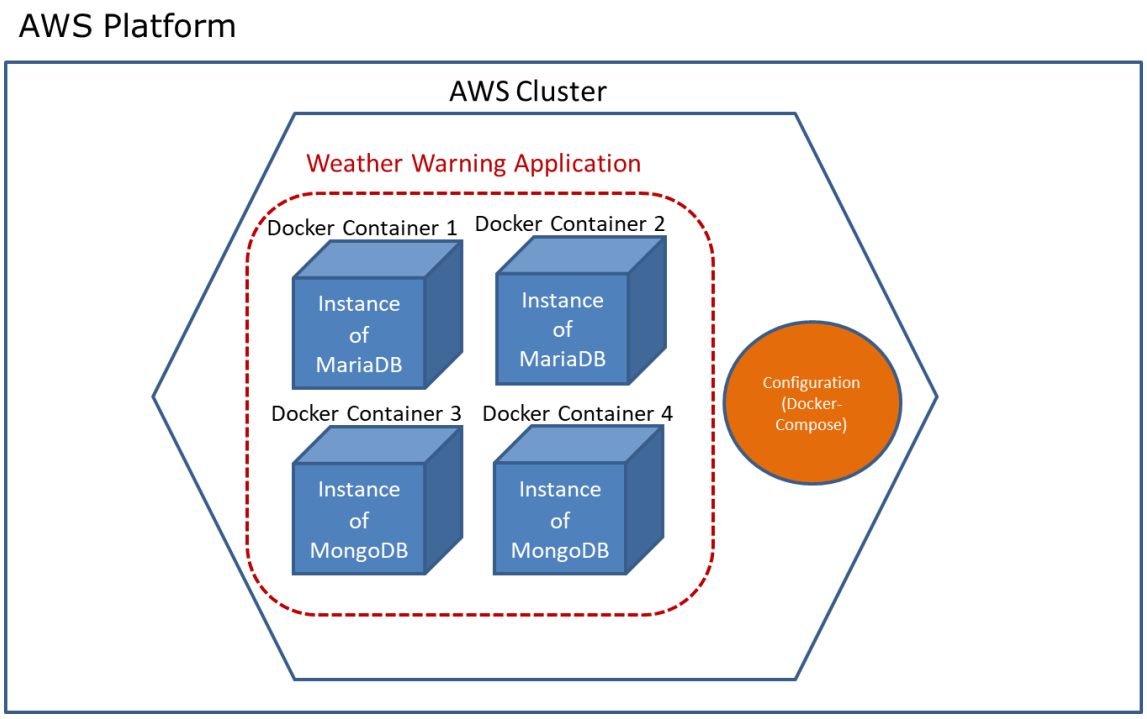

Figure 2: Deployment Example for an Application

Each database service in the Weather Warning application in Figure 1 is deployed as service instance separately in a Docker container on the AWS cloud platform.

Figure 3 shows the corresponding deployment configuration specification for this application, where Docker Container 1 is named as *vehiclemetadatadb*, Docker container 2 is named *appdata*, Docker Container 3 is named *textwarningdata* and the Docker Container 4 is named as *vehicledatadb*. A service codifies the way an image runs. For example, the service *vehiclemetadatadb* uses the standard MariaDB image from the Docker-Hub

https://hub.docker.com/_/mariadb/

plus a configuration-file, which changes the default database charset to utf8. Both services use gitlab as private container registry.

```
version: '3.7'

services:
  vehiclemetadatadb:
    image: mariadb:latest
    environment:
      MYSQL_ROOT_PASSWORD: Tm20Mel65SOhedML
    ports:
      - target: 3306
        published: 3306
    deploy:
      resources:
        limits:
          cpus: '0.5'
          memory: 512M
        reservations:
          cpus: '0.25'
          memory: 256M
appdata:
    image: mariadb:latest
    environment:
      MYSQL_ROOT_PASSWORD: Mt9Z45daWtwO4d
    ports:
      - target: 3306
        published: 3306

  vehicledatadb:
    image: mongo:latest
    command: mongod --replSet vehicledatadbReplset
  vehicledatadb-replica1:
    image: mongo:latest
    command: mongod --replSet vehicledatadbReplset
  vehicledatadb-replica2:
    image: mongo:latest
    command: mongod --replSet vehicledatadbReplset
  vehicledatadb-replica3:
    image: mongo:latest
    command: mongod --replSet vehicledatadbReplset
  vehicledatadb-rsinit:
    build:
      context: .
      dockerfile: vehicledatadb/rsinit
    entrypoint: [
      'sh',
      '-c',
      'init_set.sh'
      ]
textwarningdata:
    image: mongo:latest
    environment:
      MONGO_INITDB_ROOT_USERNAME: user
      MONGO_INITDB_ROOT_PASSWORD: Jdk4g3kK0gtO2gkq
```

Figure 3: Configuration Script Example in Docker-Compose

2.4 CONTAINER MANAGEMENT USING KUBERNETES

In large-scale applications comprising hundreds of containers spread across multiple hosts, containers need to be managed and connected to the outside world for tasks such as scheduling, load balancing, and distribution. Docker images can be deployed and

managed by container management solutions, such as Apache Mesos[19], Docker Swarm[20], or Kubernetes[21].

As a first step in TyphonDL, Kubernetes is used as a container management system in the deployment of hybrid polystores, which can be extended to other container management systems in the future if desired. Kubernetes is an open source tool to orchestrate and manage containers. Furthermore, Kubernetes has good compatibility with Docker. It has been developed by Google and is one of the most used instruments for this purpose. Kubernetes allows removing many of the manual processes involved in the deployment and scalability of containerised applications and manage easily and efficiently clusters of hosts on which containers are executed (see D1.1 [1]).

3. TYPHONDL DESIGN AND ARCHITECHTURE

In this section the design and architecture of TyphonDL is presented.

3.1 LANGUAGE DESIGN

While TyphonML models represent the high-level infrastructure of a hybrid polystore in terms of the conceptual entities to be managed and the corresponding database systems that are involved, TyphonDL models represent the deployment infrastructure of that polystore in terms of the specific cloud platform and deployment tools employed. The general approach of how TyphonDL is used for modelling the deployment infrastructure of a hybrid polystore is illustrated in Figure 4. A TyphonDL model requires two sources of input:

- A TyphonML model, from which database specific information is extracted for the TyphonDL model (e.g., which individual databases are used to manage the modelled data entities and relationships).
- Deployment specific values that instantiate configuration parameters, which generate a ready-to-use configuration file for the actual deployment task on a cloud platform.

The deployment specific configuration parameters can be supplied by a modeller or by a user of the polystore in textual form directly in the TyphonDL model itself (see section 5) or via a graphical editor that provides configuration-specific input automatically in the TyphonDL model (see D3.5 [2]).

[19] http://mesos.apache.org/
[20] https://docs.docker.com/engine/swarm/
[21] https://kubernetes.io/

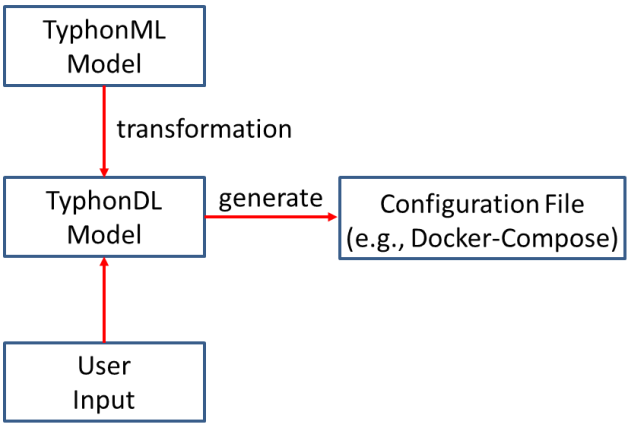

Figure 4: TyphonDL Approach

The remaining of this section is organised as follows: In section 3.2 the transformation from TyphonML models to TyphonDL models is elaborated. In section 3.3 configuration-specific parameters required for the creation of TyphonDL models are described.

3.2 TyphonML to TyphonDL Transformation

Recalling that a TyphonML model is a high-level specification of the database infrastructure of a hybrid polystore, TyphonDL follows the principle that each individual database installation will be deployed on a cloud platform in a separate container. Furthermore, based on what type of database systems are being modelled, the configuration parameters for those specific database systems will vary in the deployment model. In order to automatically generate the respective set of configuration parameters for different types of databases, the transformation of a TyphonML model extracts the database type for each database declaration and their names in a TyphonML model and adds this information in the TyphonDL model respectively (see details in deliverable D3.3 [3]).

3.3 Configuration Parameters

In order to generate a ready-to-use configuration script for the deployment of a polystore, the following parameters are necessary to be provided either by the modeller/polystore user:

- *Cloud platform provider:* The user can choose a specific cloud platform provider such as AWS, Google Cloud, Microsoft Azure, etc.
- *Deployment configuration format:* The user can choose a specific deployment configuration format such as Docker-Compose or Kubernetes.
- *Container format:* The user can choose which containerisation technology should be used, such as Docker, or rkt[22].

[22] https://coreos.com/rkt/

- *DBMSs:* Based on the type of database retrieved from a TyphonML model, the user can choose a specific database system such as MariaDB[23] or MySQL[24] for a relational database, MongoDB[25] as a document-based database, Cassandra[26] as a column-based database, and Neo4j[27] as a graph-oriented database.
- *Configuration parameters:* From a list of standard configuration parameters, the user can choose which ones should be included in the TyphonDL model such as storage space or computing power required for each container, container access, which network a container is connected to, data storage or exchange in/amongst containers, database-specific parameters such as credentials or access URI.

4. TYPHONDL METAMODEL

This section introduces the metamodel that formalises the concepts that constitute the language primitives of TyphonDL. Meta-classes are shown in Figure 5 and described below and are presented using the font as in `font`:

`DeploymentModel` represents the root container of each TyphonDL specification and consists of two distinct elements:

- `MetaModel`: It represents the set of operators on TyphonDL models.
- `Model`: It represents the set of concepts that will be used in a TyphonDL model.

These elements are further defined as follows:

`MetaModel` consists of the import operation that allows a TyphonDL model to include the contents of another TyphonDL model.

`Model` consists of the following classes that categorise the components of a TyphonDL model:

- `Type`: It represents the collection of all types that are used in a TyphonDL model.
- `Services` represents the collection of deployable software services.
- `Platform` represents the logical units in a deployment environment.

`Type` consists of the following types:

- `PlatformType` represents the set of different types of platforms that can be used in a deployment task in the cloud. Example platform types are the Amazon Web Services cloud services platform, Microsoft Azure, Google Cloud etc.
- `ClusterType` represents the set of different types of schemes to govern over a cluster of containers.

[23] https://mariadb.org/
[24] https://www.mysql.com/
[25] https://www.mongodb.com/
[26] http://cassandra.apache.org/
[27] https://neo4j.com/

- `ContainerType` represents the set of different types of containerisation software that can be used in a deployment task. Example container types are Docker, rkt, VirtualBox, VMWare, etc.

- `DBType` represents the collection of different database management systems such as MariaDB, MongoDB, Neo4j, Cassandra, etc.

`Services` distinguish between of database services `DB` and all other software services `Software`.

- Database services `DB` are named elements typed by a database type defined by `DBType`.
- `Software` is a named element and consists of a list of configuration parameters including image, URI, environment and properties.

`Platform` is a named element and typed by a platform type defined by `PlatformType`. It permits to model an individual platform space on a specific platform provider. It consists of a list of cluster declarations.

`Cluster` is a named element and typed by a cluster type defined by *ClusterType*. It consists of a list of application declarations.

`Application` is a named element that represents a software-based application that is possibly composed of several smaller software components that are deployed in individual containers.

`Container` is a named element that is typed by a container type defined by `ContainerType`. It represents a container or a virtual machine and consists of a list configuration elements that are part of the TyhonDL metamodel or other container specific properties that are defined by `Property`.

Configuration elements consist of a set of pre-selected standard deployment configuration parameters. These parameters are the following ones.

- `Image:` The image that contains a set of instructions for creating a container.
- `HelmList:` The list of specifications to use Helm charts[28] to define the setup configuration of database deployments. In particular, the name of the Helm chart, the repository name and the repository address of the respective Helm chart are specified.
- `Environment:` The environment parameters used in the setup configuration of database deployments.
- `Credentials:` The credentials to be defined in the setup configuration of database deployments.

[28] www.helm.sh/docs/topics/charts/

- `URI:` The URI for a database or a container through which they are accessed by Typhon
- `deploys:` The link between a service specification and the respective container it is deployed in.
- `depends_on:` The dependency relation between two containers.
- `Networks:` The network parameters to which a container is part of are specified.
- `Ports:` The parameters that publish a container to be reachable outside of a polystore network are specified. These parameters are typically a target port for the container, and a published port that makes the container available outside of the polystore.
- `Resources:` The parameters that control or limit the resources allocated to a container, such as CPU and memory.
- `Replication:` The parameters to define replicated instances of a container based on a specific replication mode including master-slave (primary-replica), replica set and stateless replication.
- `Volumes:` The mount parameters for the directories in a container to save data or share data between containers. The parameters are the volume name, the mount path, the volume type and any other technology specific parameters for a volume.

`Property` is a set of three different kinds of configuration declarations in the form of:

- Key-value pairs (`Key_Values`),
- key and array of values (`Key_ValueArray`),
- list of key-value pairs (`Key_KeyValueList`)

and permit to represent any other configuration properties that are specific to individual containerisation technologies.

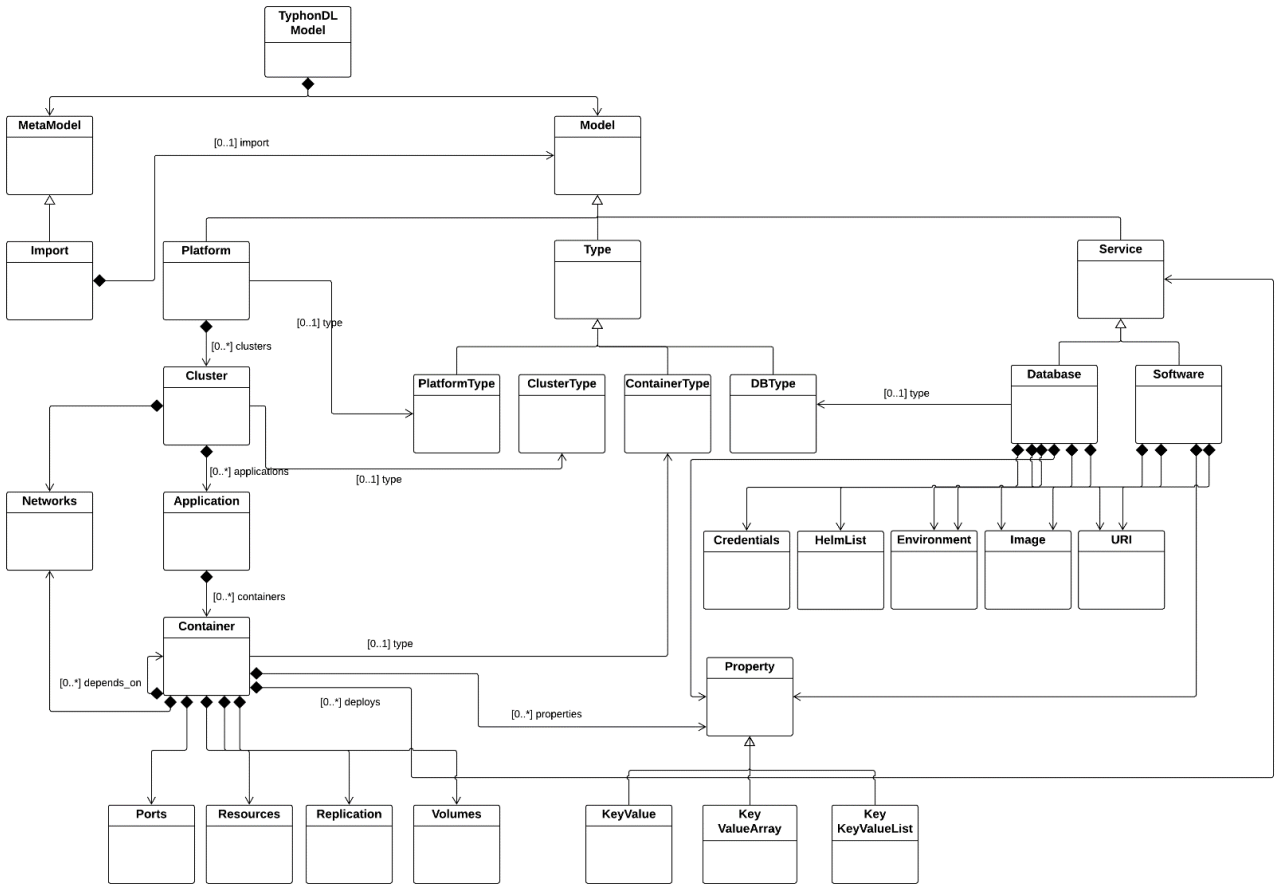

Figure 5: The TyphonDL Meta-Model

5. TYPHONDL IMPLEMENTATION

5.1 OVERVIEW

In this section the implementation of the metamodel of TyphonDL is presented. The implementation of TyphonDL is given as an EMF/Ecore [4] model.

In parallel to this implementation, the TyphonDL Wizard is implemented, which allows polystore designers and users to create TyphonDL models from user input through a graphical editor (see D3.5 [2]). The need for both a textual and a graphical editor for TyphonDL was acknowledged by the TYPHON project partners in the early stages of the project.

The TyphonDL textual editor is developed in XText[29], an Eclipse project for the design and development of domain-specific languages. The implementation of TyphonDL in XText follows from the grammar definition for the concrete syntax for TyphonDL.

[29] https://www.eclipse.org/Xtext/

Upon the compilation of the TyphonDL grammar, XText generates an Ecore-model, the internal infrastructure for the parsing, linking, type-checking of TyphonDL models written textually in XText.

5.2 TYPHONDL CONCRETE SYNTAX

In this section, the concrete syntax of TyphonDL is described using an example TyphonDL model as illustrated in Figure 6.

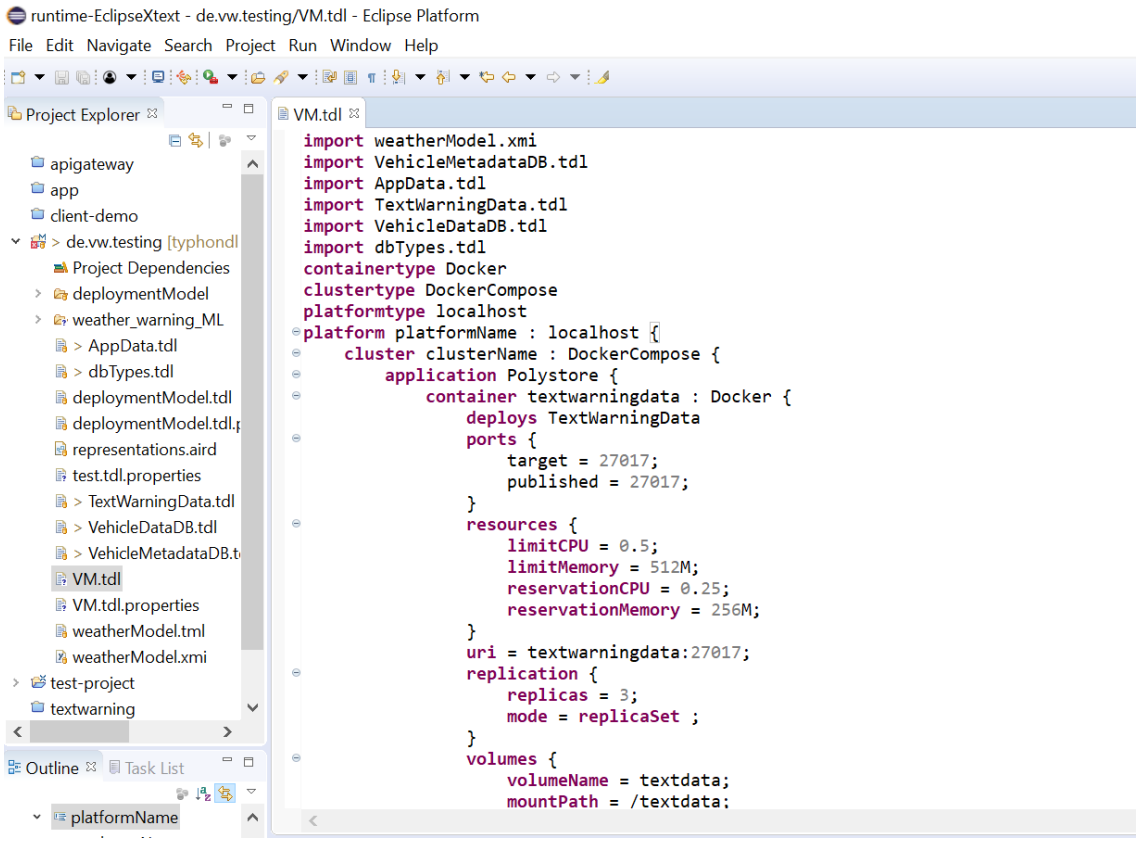

Figure 6: TyphonDL Model Example for Docker-Compose

This TyphonDL model is generated using the TyphonDL Wizard (see D3.5 [2] for more details) by taking a TyphonML model and user inputs for deployment configuration parameters. It consists of a list of imports of automatically generated TyphonDL models based on the TyphonML model input. These imports are declared using the TyphonDL keyword **`import`** for import.

Database types are given in the *dbTypes.tdl* file, which consists of the following database type declarations in Listing 1, where each database type is declared using the TyphonDL keyword **`dbtype`**.

```
dbtype Mongo {
      default image = mongo:latest;
}
dbtype MariaDB {
      default image = mariadb:latest;
}
```

Listing 1: TyphonDL Model Example - Database Type Declarations

Furthermore, each database type declaration must consist of a default image from which the respective database system can be deployed in a container.

The remaining *.tdl* files consist of specific database declarations based on the database models extracted from the initial TyphonML model.

After introducing these two database systems in TyphonDL as **dbtype** declarations, individual databases can be declared using the TyphonDL constructor **database** as in the example of Listing 2.

```
database TextWarningData : Mongo {
      credentials {
            username = user;
            password = TT3s=RA1spIKALJA;
      }
}
external database AppData : MariaDB {
      uri = 192.168.11.40:8080;
      credentials {
            username = root;
            password = OdgU9gP0bosqDle9;
      }
}
```

Listing 2: TyphonDL Model Example using Docker-Compose – Database Specifications

The first database specification above is a Mongo database named `TextWarningData` and typed by the database type `Mongo`. The database configuration for `TextWarningData` introduces credentials (username and password) for a user using the TyphonDL keyword **credentials**.

Database specifications can optionally use the **external** attribute to mark a database to be external. This is used to annotate that the database that is being introduced is already an existing database installation that runs outside of the Typhon polystore and is accessible at the given URI.

The deployment example used in this TyphonDL model uses Docker containers, Docker-Compose as the container configuration format, and Amazon Web Services (AWS) as the platform provider. More specifically, this is modelled by the type declarations as can be seen in Listing 3.

```
containertype Docker
clustertype DockerCompose
platformtype AWS
```

Listing 3: TyphonDL Model Example using Docker-Compose - Type Declarations

The next part in this example is the specification of the deployment platform and the containers to be deployed as given in
Listing 4.

```
containertype Docker
clustertype DockerCompose
platformtype localhost
platform platformName : localhost {
      cluster clusterName : DockerCompose {
            application Polystore {
                  container textwarningdata : Docker {
                        deploys TextWarningData
                        ports {
                              target = 27017;
                              published = 27017;
                        }
                        resources {
                              limitCPU = 0.5;
                              limitMemory = 512M;
                              reservationCPU = 0.25;
                              reservationMemory = 256M;
                        }
                        uri = textwarningdata:27017;
                        replication {
                              replicas = 3;
                              mode = replicaSet;
                        }
                        volumes {
                              volumeName = textwarningdata;
                              mountPath = /textwarningdata;
                              volumeType = volume;
                              volume {
                                    nocopy = true;
                              }
                        }
                  }
                  Volumes textwarningdata;
            }
      }
}
```

Listing 4: TyphonDL Model Example using Docker-Compose - Deployment Platform and Container Specification

The TyphonDL keyword `platform` declares a platform of type `AWS` named `platformName`, which consists of a `cluster` declaration named `clusterName` and is

of cluster type `DockerCompose`. The cluster `clusterName` consists of an **`application`** declaration named `Polystore`.

Inside the application `Polystore` one Docker container is specified: The TyphonDL keyword **`container`** declares a container named `textwarningdata` and typed by the container type `Docker`. In this container declaration the following configuration parameters are specified:

- Keyword **`deploys`** links the container `textwarningdata` to the Mongo database specification `TextWarningData`,
- Keyword **`ports`** specifies the ports at which the container `textwarningdata` will be accessible. In this example, the target port and the published port for this container are specified.
- Keyword **`resources`** specifies limit and reservation properties on the CPU and memory allocation for the container `textwarningdata`. The limit properties are given using **`limitCPU`** and **`limitMemory`**, and the reservation properties are given using **`reservationCPU`** and **`reservationMemory`**.
- Keyword **`uri`** specifies the URI of the container `textwarningdata`.
- Keyword **`replication`** specifies the number of replicas to be created including the container `textwarningdata` itself using the primary-replica replication. It uses the keyword **`replica`** to set the replica number and **`mode`** for the specific kind of replica setup, which is in this example **`replicaSet`**.
- Keyword **`volumes`** specifies the name of the volume using the keyword **`volumeName`**, its mount directory using keyword **`mountPath`**, and the volume type using the keyword **`volumeType`**. Additionally, in this example, further volume specific configuration parameter `nocopy` is specified to be true.

These configuration parameters are then translated by the Typhon Script Generator (see D3.5 [2]) to the respective Docker-Compose specification in the yaml-format.

The above TyhonDL examples used Docker-Compose as the underlying containerisation technology. Alternatively, using Kubernetes as the underlying containerisation technology, the same polystore example can be modelled as illustrated in Listing 5 through Listing 7.

In the database specification in Listing 5, Kubernetes specific configuration details for the database `TextWarningData` are specified using a Helm chart. The keyword **`helm`** introduces the Helm chart parameters such as the repository name and address using the keywords **`repoName`** and **`repoAddress`**, respectively, as well as the name of the chart using **`chartName`**. The remaining of the specifications in `TextWarningData` is credentials.

```
database TextWarningData : Mongo {
	helm {
		repoName = bitnami;
		repoAddress = https://charts.bitnami.com/bitnami;
		chartName = mongodb;
	}
	credentials {
		username = user;
		password = KY92flr05L1bma3m;
	}
}
```

Listing 5: TyphonDL Model Example using Kubernetes – Database Specification

The following TyphonDL type declarations in Listing 6 are necessary to choose Kubernetes as the underlying containerization technology in this example.

```
containertype Docker
clustertype Kubernetes
platformtype minikube
```

Listing 6: TyphonDL Model Example using Kubernetes – Type Declarations

In Listing 7, the platform and container specifications are given. It is important to note that this model differentiates from the TyphonDL model example in Listing 4 only in terms of platform type and cluster type used, which are given above, and the volumes specification in the container `textwarningdata`. In particular, the volumes declaration uses a volume type that is specific to Kubernetes that allows persistent volumes and an additional configuration declaration required by this volume type.

```
platform platformName : minikube {
      cluster clusterName : Kubernetes {
            application Polystore {
                  container textwarningdata : Docker {
                        deploys TextWarningData
                        ports {
                              target = 27017;
                              published = 27017;
                        }
                        resources {
                              limitCPU = 0.5;
                              limitMemory = 512M;
                              reservationCPU = 0.25;
                              reservationMemory = 256M;
                        }
                        uri = textwarningdata:27017;
                        replication {
                              replicas = 3;
                              mode = replicaSet;
                        }
                        volumes {
                              volumeName = textwarningdata;
                              mountPath = /textwarningdata;
                              volumeType = persistentVolumeClaim;
                              claimName = true;
                        }
                  }
            }
      }
}
```

Listing 7: TyphonDL Model Example using Kubernetes – Deployment Platform and Container Specification

5.3 TYPHONDL VALIDATION

Validation is a necessary step to prevent a modeller from adding semantically meaningless content in a model, although it might still be syntactically well-formed. It is a standard approach in language design and implementation to define the syntax of the language and a set of validation rules for the correct use of syntax.

In TyphonDL, Xtend[30] is used to introduce validation rules for this purpose. In particular, the validation rules in TyphonDL are categorised as follows:

- **Unique Declarations:** It is not allowed to have multiple occurrences of the same declaration in a TyphonDL model.
- **Technology Specific Keywords:** It is not allowed to use a random keyword in a TyphonDL model that does not belong to the set of keywords in the chosen

[30] www.eclipse.org/xtend/

containerisation technology. This is implemented specifically for in Docker-Compose and Kubernetes.

- **Database Systems:** It is checked in the declaration of a database system using `dbtype` whether the image of that database does mention the database name.

6. CONCLUSION

This document presents results of the work done in WP3 and details the TyphonDL language for the modelling and deployment of the envisioned polystore systems. It includes the concepts required to generate deployable solutions based on TyphonML models.

TyphonDL is defined along a list of technology and industrial use case requirements that are set at the beginning of the project in D1.1 [1]. An evaluation of TyphonDL with respect to those requirements is given in Table 1 and Table 2. Further evaluation of TyphonDL with respect to its strengths and limitations is presented and a future outlook is discussed below.

Table 1: TyphonDL Technology Requirements

ID	Requirement	Priority	Status
12	TyphonDL models shall allow for specification of the components in deployment configuration.	SHALL	Implemented
13	TyphonDL models shall allow for specification of interplay between components in deployment configuration.	SHALL	Implemented
14	TyphonDL models shall allow for specification of deployment operations on the components.	SHALL	Implemented
15	TyphonDL shall be adaptable to the de facto standard virtual image configuration technique Docker.	SHALL	Implemented
16	TyphonDL models shall allow for the definition of deployment properties.	SHALL	Implemented
17	TyphonDL shall allow for the definition of individual nodes.	SHALL	Implemented
18	TyphonDL shall allow for the definition of standard configuration concepts.	SHALL	Implemented
19	The Hybrid Polystore Deployment shall support scalability to large amounts of data.	SHALL	Implemented (using Kubernetes)

ID	Requirement	Priority	Status
20	The Hybrid Polystore Deployment component shall develop tools and services to define (and edit) deployment specifications.	SHALL	Implemented
21	TyphonDL should support templates for creation of Polystore Deployments.	SHOULD	Implemented
22	TyphonDL should allow defining the level of redundancy for the database instance so that some consistency checks on the data can be supported.	SHOULD	Implemented for some DBMS
23	The Polystore Deployment should be compatible with several cloud platform providers.	SHOULD	Implemented for cloud platforms supporting Docker
24	TyphonDL should allow for the definition of collection/cluster of nodes	SHOULD	Implemented
25	TyphonDL may be adaptable to other virtual image configuration techniques.	MAY	The tools are prepared to be extended Prototype implementation is for Docker/Kubernetes
26	TyphonDL may support heterogeneous cloud platforms.	MAY	Implemented
27	The hybrid polystore shall support the deployment and execution of text processing pipelines.	SHALL	Part of Analytics Deployment

Table 2: TyphonDL Industrial Use Case Requirements

ID	Requirement	Priority	Status
30	The polystore deployment language shall allow to define distributed topologies with database master and slave nodes	SHALL	Implemented
31	The polystore deployment language shall allow to define the level of redundancy for the database instance	SHALL	Implemented

ID	Requirement	Priority	Status
32	The polystore deployment language should be based on standard file formats (JSON, XML, etc.)	SHOULD	Implemented
33	The polystore deployment language shall allow to size each individual node storage space	SHALL	Implemented
34	The polystore deployment language should allow to embed the configuration of the underlying cloud manager so that a single configuration file is handled to generate the final virtual machines	SHOULD	Implemented
35	The polystore deployment language should allow to embed the configuration of the underlying database servers (e.g. based on SQL, Lucene etc.) so that the overall instance can be tuned at deployment time	SHOULD	Implemented
36	The polystore deployment language shall allow to define collections of nodes without requiring each individual node to be configured in a separate section	SHALL	Implemented
37	The polystore deployment language should allow a mechanism to deal with elastic instances, in which nodes are started and stopped when needed	SHOULD	Implemented
38	The polystore deployment language shall support adding additional datastores	SHALL	Implemented
39	The polystore deployment language should provide an API that is accessible through .NET	SHOULD	Not implemented Access to polystore will be handled via polystoreAPI only

- **Strengths:**
  - TyphonDL is designed as a meta-containerisation language. It is technology-independent and abstracts away from concrete language and implementation details of containerisation technologies. This permits TyphonDL to model various different specific containerisation technologies besides the most commonly used ones.

  - The declarative nature of TyphonDL is especially useful to specify concrete database or software implementations in a flexible way that are part of the modelled polystore and allows the possibility to introduce individual systems or new versions of existing systems in the model itself on demand.

  - TyphonDL is an extensible language. New language features can be easily added to TyphonDL to support possible future forms of polystore architectures.

- **Limitations:** TyphonDL does not provide support for aliases or language-specific notations used in individual containerisation technologies as these are not an essential part of deployment modelling and are typically custom-defined in each specific technology. This has the consequence that TyphonDL-generated configuration scripts always use one default syntax of the respective containerisation technology. Any desired notations or aliases can be however always manually added in the generated configuration script.
- **Future Outlook:** The strengths mentioned above makes TyphonDL scalable to a wide range of application domains. In the long-term perspective, TyponDL can be extended to support, along with the strong modelling features of TyphonML, a user community of Typhon-based polystore applications.

7. REFERENCES

[1] TYPHON Consortium, „D1.1 Project Requirements,“ 2018.

[2] TYPHON Consortium, „D3.5 Optimized Hybrid Polystore VM Assembly Tools,“ 2020.

[3] TYPHON Consortium, „D3.3 TyphonML to TyphonDL Model Transformation Tools,“ 2019.

[4] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick und T. Grose., Eclipse Modeling Framework, Addison wesley, 2003.

[5] TYPHON Consortium, „D7.1 Architectural Guidelines Report,“ 2018.