TYPHON
POLYGLOT AND HYBRID PERSISTENCE ARCHITECTURES FOR BIG DATA ANALYTICS

Project Number 780251

D3.5 Optimized Hybrid Polystore VM Assembly Tools

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	Document Structure	02.06.2020
0.2	First Content	02.06.2020
0.3	Content update	16.06.2020
0.4	Content ready for internal review	18.06.2020
0.5	Internal review comments integrated and ready for external review	29.06.2020
0.6	Integration of external review	07.07.2020
1.0	Final version	09.07.2020

TABLE OF CONTENTS

TABLE OF FIGURES

EXECUTIVE SUMMARY

This deliverable presents the work done in Task T3.4 Assembly of Optimised Hybrid Polystore VMs from Deployment Models. This task produces tools for generating installation and configuration scripts for deploying modelled hybrid polystores by targeting selected virtual machine image assembly technologies. The generation of the VMs assembly is optimized by taking into account both the characteristics of the modelled polystores, and the considered deployment contexts, e.g., hardware configuration, costs, workloads, performance, costs, and storage size. The produced virtual machines are directly deployable on cloud infrastructure.

This deliverable presents the tools able to generate configuration scripts to assemble Hybrid polystore VMs from source TyphonDL models. Also, the modelling tools supporting the creation of TyphonDL models are presented.

1. INTRODUCTION

1.1 OVERVIEW

This deliverable, D3.5, presents the work done in Task T3.4 Assembly of Optimised Hybrid Polystore VMs from Deployment Models and is the final version of the deliverable D3.2 TyphonDL Tools (TYPHON Consortium, 2018). This task has produced tools for generating installation and configuration scripts for deploying modelled hybrid polystores by targeting selected virtual machine image assembly technologies. The generation of the VMs assembly is optimized by taking into account both the characteristics of the modelled polystores, and the considered deployment contexts, e.g., hardware configuration, costs, workloads, performance, costs, and storage size. The produced virtual machines are directly deployable on cloud infrastructure.

This deliverable presents the tools able to generate configuration scripts to assemble Hybrid polystore VMs from source TyphonDL models. Also, the modelling tools supporting the creation of TyphonDL models are presented.

The chosen technology for virtualising the polystore components is Docker[1] containers, used either with Docker Compose[2], Docker Swarm[3] or Kubernetes[4].

1.2 POLYSTORE OVERVIEW

The Polystore - and therefore the TyphonDL model - consists of the following components:

- Typhon API
- Typhon UI
- Typhon Metadata Database
- Typhon QL
- Optional Typhon Analytics
- The User Databases

The user is only able to edit the DL model for the user databases and the analytics component, the other configuration parameters are provided by the respective components and are not editable.

To create a Polystore, the TyphonDL Tools generate a TyphonDL model from a given TyphonML (or ML) model. After completing the TyphonDL (or DL) model, scripts are generated and the Polystore is started. When the Polystore is started, the ML and DL

[1] https://www.docker.com/
[2] https://docs.docker.com/compose/
[3] https://docs.docker.com/engine/swarm/
[4] https://kubernetes.io/

model are uploaded to the Typhon Metadata Database automatically. The Typhon API parses the DL model and provides the other components with connection information about all Polystore components. This way the DL model contains "addresses" to all the Polystore components.

This procedure will be explained in more detail in the following sections.

1.3 STRUCTURE OF THE DOCUMENT

The document is organised as follows:

- Section 2 presents the description of usage of the TyphonDL tools that help create a TyphonDL model and generate deployment scripts.
- Section 3 shortly describes the implementation of the TyphonDL tools.
- Section 4 presents the conclusion of the document.

The Typhon Deployment Language concepts, presented in D3.4 Hybrid Polystore Deployment Language (Final Version) (TYPHON Consortium, 2020) will be used and therefore not further explained in the present document.

2. USAGE OF TYPHONDL TOOLS

In this section the usage of the modelling tools including script generation is presented. After creating a TyphonML model with the help of the TyphonML modelling tools ((TYPHON Consortium, D3.2 TyphonDL Modeling Tools, 2019) and (TYPHON Consortium, 2019)) a TyphonDL model can be created with the help of the TyphonDL Wizard (see 2.2) from the ML model. The wizard uses the previously defined (default or use-case specific) templates (see 2.1) and creates a TyphonDL model file and additional model files for every database that can be edited with the textual and/or graphical editor (see 2.3). When the DL model is ready, the TyphonDL Script Generator can be used to generate technology dependent deployment scripts (see 2.4).

For the general reability of this section, TyphonDL meta-class objects, presented using the font `font`, can be revisited in D3.4, Hybdrid Polystore Deployment Language (Final Version) (TYPHON Consortium, 2020).

2.1 TYPHONDL TEMPLATES

The TyphonDL plugin comes with a set of default `DB` and `DBType` templates, that can be viewed, imported, exported and edited in *Eclipse* → *Window* → *Preferences* → *TyphonDL* → *Templates* (see Figure 1). Here, additional templates can be added, or company specific `DB` settings can be defined and used for creating a new Polystore deployment.

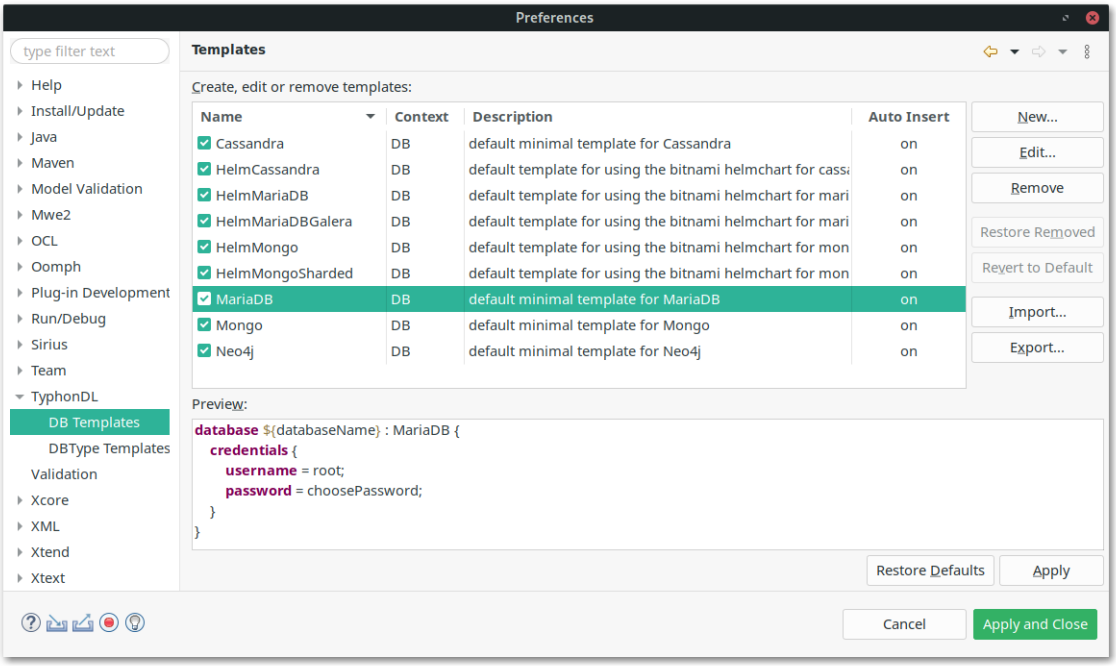

Figure 1: TyphonDL DB Template preferences

The default DB Templates include:

- MariaDB with `DBType` MariaDB[5] containing `Credentials` with *username* = *root* and a *password* to be set by the user.
- Mongo with `DBType` Mongo[6] containing `Credentials` with *username* and *password* to be set by the user.
- Cassandra with `DBType` Cassandra[7] containing an `Environment` to set the *maximum heap size* and the *amount of heap memory allocated to newer objects*[8].
- Neo4j with `DBType` Neo4j[9] containing `Credentials` with *username* = *neo4j* and a *password* to be set by the user.
- HelmMariaDB with `DBType` MariaDB containing a `HelmList`[10] using bitnami/mariadb[11] and `Credentials` with *username* = *root* and a *password* to be set by the user.
- HelmMariaDBGalera with DBType mariadbgalera containing a `HelmList` using bitnami/mariadb-galera[12] and `Credentials` with *username* = *root* and a *password* to be set by the user.

[5] https://hub.docker.com/_/mariadb
[6] https://hub.docker.com/_/mongo
[7] https://hub.docker.com/_/cassandra
[8] https://docs.datastax.com/en/ddac/doc/datastax_enterprise/operations/opsConHeapSize.html
[9] https://hub.docker.com/_/neo4j
[10] A HelmList includes the helm repository's name and address, and the helm chart's name, see [2].
[11] https://github.com/bitnami/charts/tree/master/bitnami/mariadb

- HelmMongo with `DBType` Mongo containing a `HelmList` using bitnami/mongodb[13] and `Credentials` with *username=root* and a *password* to be set by the user.

- HelmMongoSharded with `DBType` mongoshareded containing a `HelmList` using bitnami/mongodb-sharded[14] and `Credentials` with *username=root* and a *password* to be set by the user.

- HelmCassandra containing a `HelmList` using bitnami/cassandra[15] and `Credentials` with *username* and *password* to be set by the user.

- HelmNeo4j containing a `HelmList` using neo4j-helm[16] and `Credentials` with *username=neo4j* and a *password* to be set by the user.

2.2 TYPHONDL WIZARD

To create a TyphonDL model from a TyphonML model the TyphonDL Wizard has to be started by selecting the given ML model and selecting *Create TyphonDL model* in the Typhon context menu (see D3.2 (TYPHON Consortium, 2018)).

On the first page of the wizard (see Figure 2) the name for the TyphonDL model has to be entered and a deployment technology such as Docker Compose, or Kubernetes has to be chosen from a dropdown menu. The selected technology will be included in the model in the form of `Clustertype` which is used when defining a `Cluster`:

```
clustertype DockerCompose

cluster clusterName: DockerCompose …
```

The Analytics component (TYPHON Consortium, 2019) can be activated and deployment scripts can be created to either run it alongside the other Polystore components, or to run it on a different machine. An already running Analytics component can also be added to the model by giving its URI. For the UI to be reachable by the API, the API URI (consisting of host and port) has to be given to the Wizard. If Swarm Mode or Kubernetes is used, it is possible to scale the stateless parts of the Polystore, i.e. the API and the QL server.

[12] https://hub.helm.sh/charts/bitnami/mariadb-galera
[13] https://github.com/bitnami/charts/tree/master/bitnami/mongodb
[14] https://hub.helm.sh/charts/bitnami/mongodb-sharded
[15] https://hub.helm.sh/charts/bitnami/cassandra
[16] https://github.com/neo4j-contrib/neo4j-helm

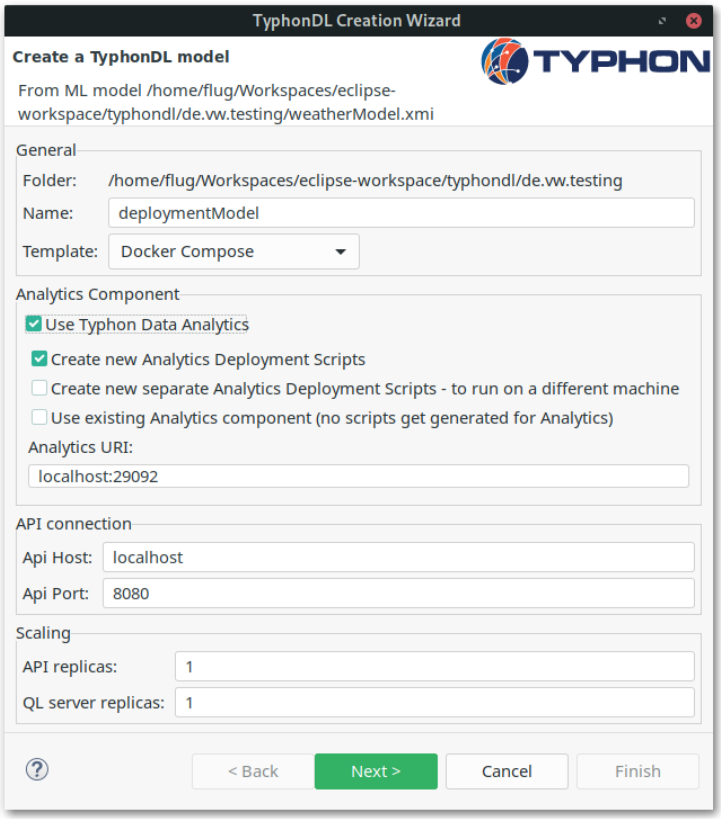

Figure 2: TyphonDL Creation Wizard: page one

If the Analytics component is to be generated, an optional page (see Figure 3) appears after the first one. Here, the Analytics component can be configured.

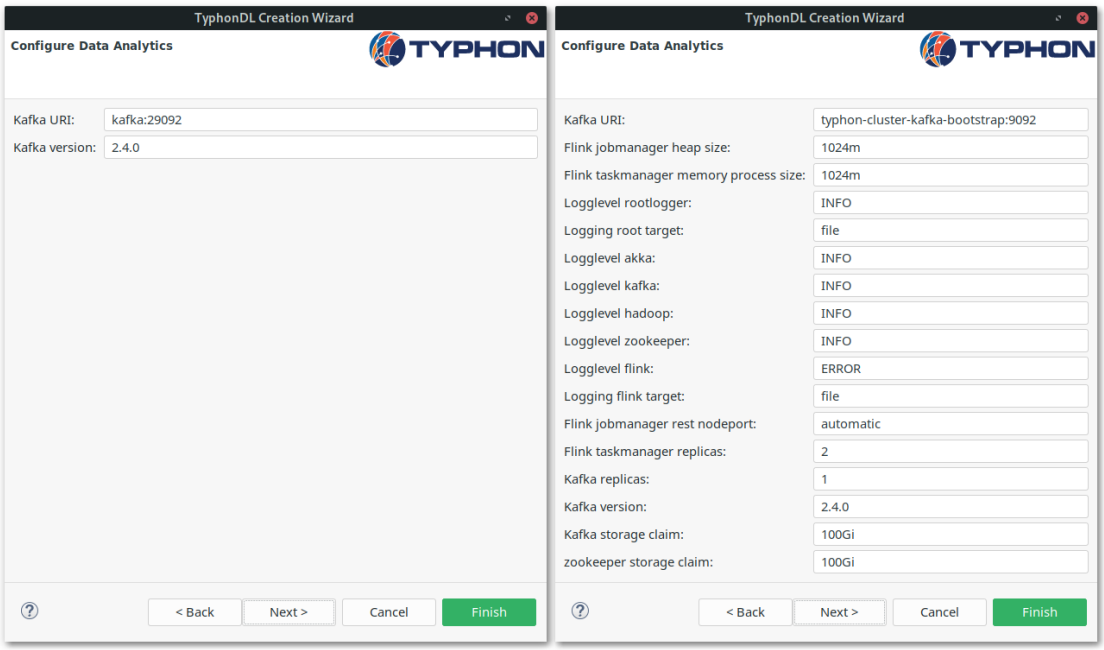

Figure 3: TyphonDL Creation Wizard: Configuring the Analytics component Docker Compose vs. Kubernetes)

TyphonML provides an XMI representation of the ML model that is parsed by the TyphonDL Wizard and that filters out the databases to be deployed by TyphonDL. For each database the second page of the wizard (see Figure 4) provides the possibility to choose one of the following options:

1. Use a pre-existing `DB` model file[17] if a file with the name *<databasename>.tdl* exists in the project folder.

2. Create a new `DB` model object by choosing a template (shown in 2.1) from the drop down menu.

3. Use an existing externally running database. A `DB` model object with the flag `external`, an `URI` and the `DBType` of the selected template is created.

4. If Kubernetes is chosen on the first page, the option to use a Helm Chart[18] is added. Here, one of the templates already containing a `HelmList` should be chosen, their names all start with "Helm". Otherwise a new default `HelmList` using bitnami[19] as *Helm Repo* is created.

In each of the above cases, the resulting `DB` model object is cached in the Creation Wizard for further configuration on the next pages.

[17] (examples in Listing 3, Listing 4 and Listing 5)
[18] https://hub.helm.sh/
[19] https://bitnami.com/stacks/helm

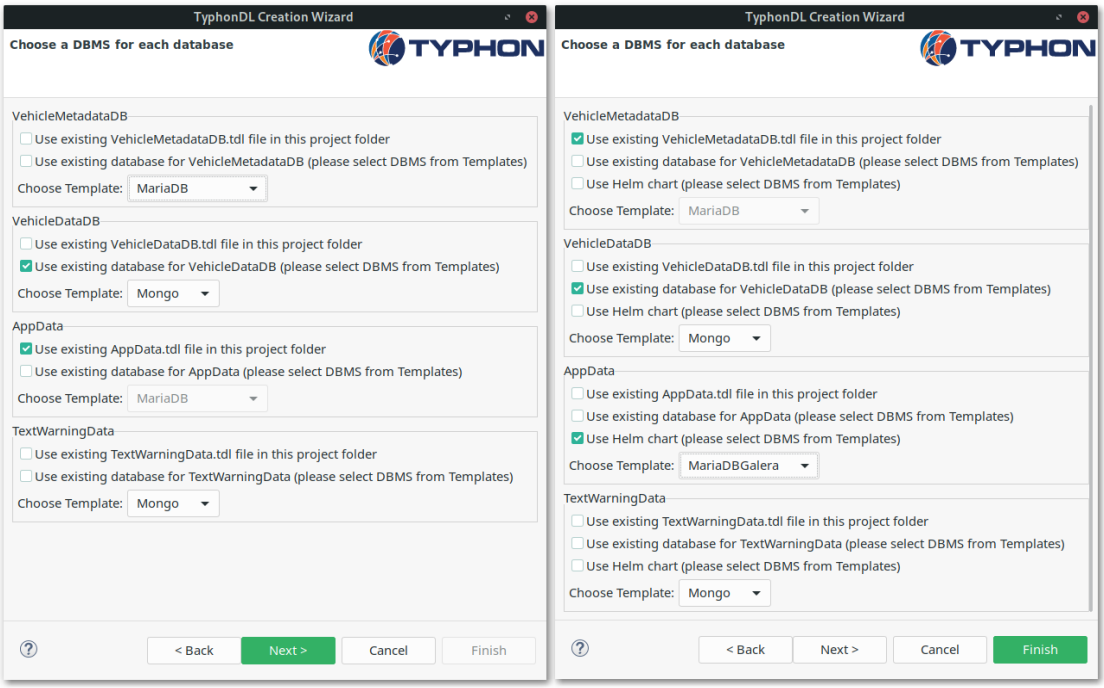

Figure 4: TyphonDL Creation Wizard: Choosing the DBMS for each database (Docker Compose vs. Kubernetes)

In Figure 5 two example `DB` configurations are shown. If the `DB` is not set to external, a `Container` model object for each database is created and cached together with the `DB` object in the Wizard. The `Container` gets an `URI` object with the value *<containerName>:<containerPort>*. This `URI` is parsed by the API to know where to reach each database.

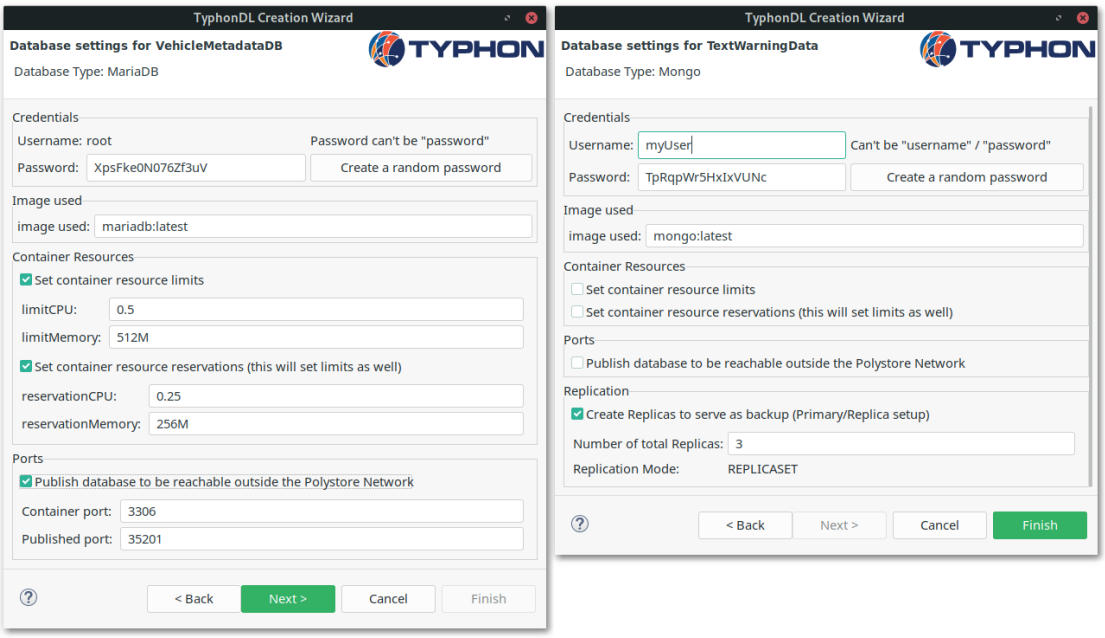

Figure 5: TyphonDL Creation Wizard: Further database configuration (MariaDB container vs. MongoDB container).

On the left side of Figure 5 database settings for VehicleMetadataDB are presented. The previously chosen `DBType` is shown on the top of the page – here MariaDB (compare to Figure 4). The template (see Figure 1) has a given username (*root*) and only allows to choose the password. The Wizard provides the possibility to generate a 16 digit password containing small and capital letters and numbers. If a different image version should be used, it can be defined in the "Image used" group. Next, container resources can be defined by checking the respective checkboxes. This will add a `Resources` object to the `Container`. CPU is measured in CPU units, given as the fragment of available processing time (0.2 = 20%). Memory is measured in bytes and is expressed as integer using one of these suffixes: T, G, M, K. It's possible - though not recommended in production - to publish a database container with a given "Published Port" in the "Ports" group. This will add a `Ports` object to the `Container`.

On the left side of Figure 5, the MongoDB TextWarningData can be configured. Here, both username and password can be chosen. Additionally, to the options above, it's allowed to replicate the MongoDB[20] if Docker Compose is used. If the Primary/Secondary option is chosen, a `Replication` object is added to the `Container`. The number of total Replicas denotes the number of additionally created containers (see 3.3).

On the left side of Figure 6, the database settings for VehicleDataDB, an `external` MongoDB (compare with the checkbox in Figure 4:left) are presented. Additionally, to setting the `Credentials`, the user has to give an `URI` pointing to the database in the "Database Address" group.

An example for using Helm charts in the `DB` AppData (compare with the checkbox in Figure 4:right) is given on the right side of Figure 6. The template for MariaDB Galera (see 2.1) already contains the repository settings. The user can specify the use of a custom *values* file. If the valuesFile field contains the repository name (here "bitnami"), the default values provided by the chart are taken[21].

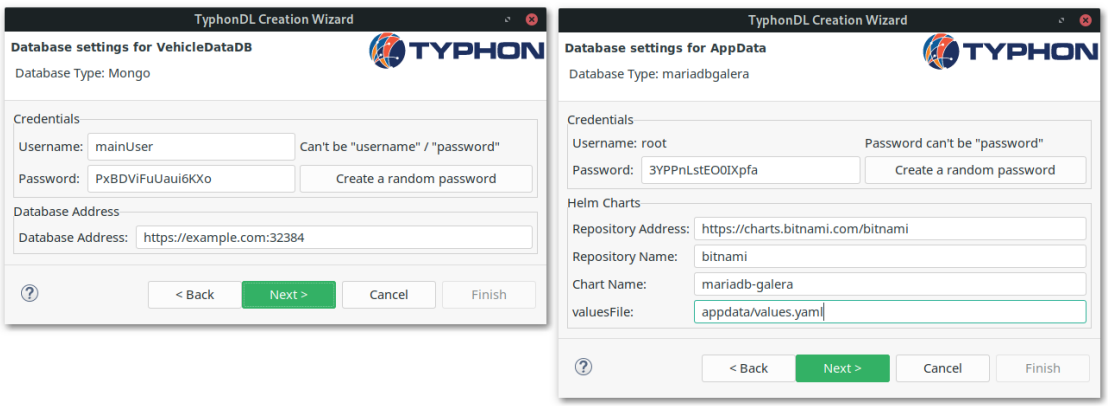

Figure 6: TyphonDL Creation Wizard: Further database configuration (MongoDB external database vs. MariaDB Galera Cluster).

When the wizard is finished, the following TyphonDL files get added to the project:

[20] https://docs.mongodb.com/manual/replication/
[21] E.g. https://github.com/bitnami/charts/blob/master/bitnami/mariadb-galera/values.yaml

- TyphonDL model file with the name that was given in the wizard (examples in Listing 1 using Docker Compose and Listing 2 using Kubernetes).
- Properties file needed to generate deployment scripts (see 3.3 for more details).
- One model file for each database (examples in Listing 3, Listing 4 and Listing 5).
- One model file containing the `DBTypes` (example in Listing 6).

```
import weatherModel.xmi
import VehicleMetadataDB.tdl
import AppData.tdl
import TextWarningData.tdl
import VehicleDataDB.tdl
import dbTypes.tdl
containertype Docker
clustertype DockerCompose
platformtype localhost
platform platformName : localhost {
	cluster clusterName : DockerCompose {
		application Polystore {
			container vehiclemetadatadb : Docker {
				deploys VehicleMetadataDB
				ports {
					target = 3306 ;
					published = 35201 ;
				}
				resources {
					limitCPU = 0.5 ;
					limitMemory = 512M ;
					reservationCPU = 0.25 ;
					reservationMemory = 256M ;
				}
				uri = vehiclemetadatadb:3306 ;
			}
			container appdata : Docker {
				deploys AppData
				uri = appdata:3306 ;
			}
			container textwarningdata : Docker {
				deploys TextWarningData
				uri = textwarningdata:27017 ;
				replication {
					replicas = 3 ;
					mode = replicaSet ;
				}
			}
		}
	}
}
```

Listing 1: Main model file deploymentModel.tdl generated by the TyphonDL Creation Wizard using Docker Compose

```
import weatherModel.xmi
import AppData.tdl
import TextWarningData.tdl
import VehicleMetadataDB.tdl
import VehicleDataDB.tdl
import dbTypes.tdl
containertype Docker
clustertype Kubernetes
platformtype minikube
platform platformName : minikube {
	cluster clusterName : Kubernetes {
		application Polystore {
			container appdata : Docker {
				deploys AppData
				uri = appdata:3306 ;
			}
			container textwarningdata : Docker {
				deploys TextWarningData
				uri = textwarningdata:27017 ;
			}
			container vehiclemetadatadb : Docker {
				deploys VehicleMetadataDB
				ports {
					target = 3306 ;
					published = 3306 ;
				}
				resources {
					limitCPU = 0.5 ;
					limitMemory = 512M ;
					reservationCPU = 0.25 ;
					reservationMemory = 256M ;
				}
				uri = vehiclemetadatadb:3306 ;
			}
		}
	}
}
```

Listing 2: Main model file deploymentModel.tdl generated by the TyphonDL Creation Wizard using Kubernetes

```
database AppData : MariaDB {
	credentials {
		username = root ;
		password = zRcUgpmgcBmZuSSI ;
	}
}
```

Listing 3: AppData.tdl containing the password created in the Wizard

```
external database VehicleDataDB : Mongo {
	uri = https://example.com:32384 ;
	credentials {
		username = mainUser ;
		password = yG7w4djhIg1F2ZI3 ;
	}
}
```

Listing 4: VehicleDataDB.tdl is an external database which is not deployed by a container in the main model file

```
database AppData : mariadbgalera {
	helm {
		repoName = bitnami ;
		repoAddress = https://charts.bitnami.com/bitnami ;
		chartName = mariadb-galera ;
		valuesFile = appdata/values.yaml ;
	}
	credentials {
		username = root ;
		password = ell8qy43MvnwxFEa ;
	}
}
```

Listing 5: AppData.tdl when using a Helm Chart and giving a custom values file

```
dbtype MariaDB {
	default image = mariadb:latest;
}
dbtype Mongo {
	default image = mongo:latest;
}
dbtype mariadbgalera {
	default image = bitnami/mariadb-galera;
}
```

Listing 6: dbtypes.tdl

2.3 TYPHONDL EDITOR

Xtext provides a textual editor with syntax highlighting, auto completion and an outline view. If the project that includes the models holds an Xtext nature, the TyphonDL Creation Wizard automatically adds it to the project, and linking between files shown in Figure 7 is also provided.

The TyphonDL Creation Wizard already creates a valid TyphonDL model, comprehensive enough to generate polystore deployment scripts, but the user can still add additional information. When Kubernetes is chosen, the `Platformtype` is automatically set to "minikube[22]", a testing environment. A different Platform Type can easily be used by changing the value of `Platformtype` and adding a "kubeconfig" Key_`Values` to the Cluster. The "kubeconfig" file can be downloaded from the cluster provider. An example for using AWS is shown in Listing 7.

[22] https://kubernetes.io/docs/setup/learning-environment/minikube/

```
platformtype AWS
platform platformName : AWS {
      cluster clusterName : Kubernetes {
          kubeconfig = /path/to/downloaded/kubeconfig.yaml;
```

Listing 7: Changing the Platformtype and providing a kubeconfig file

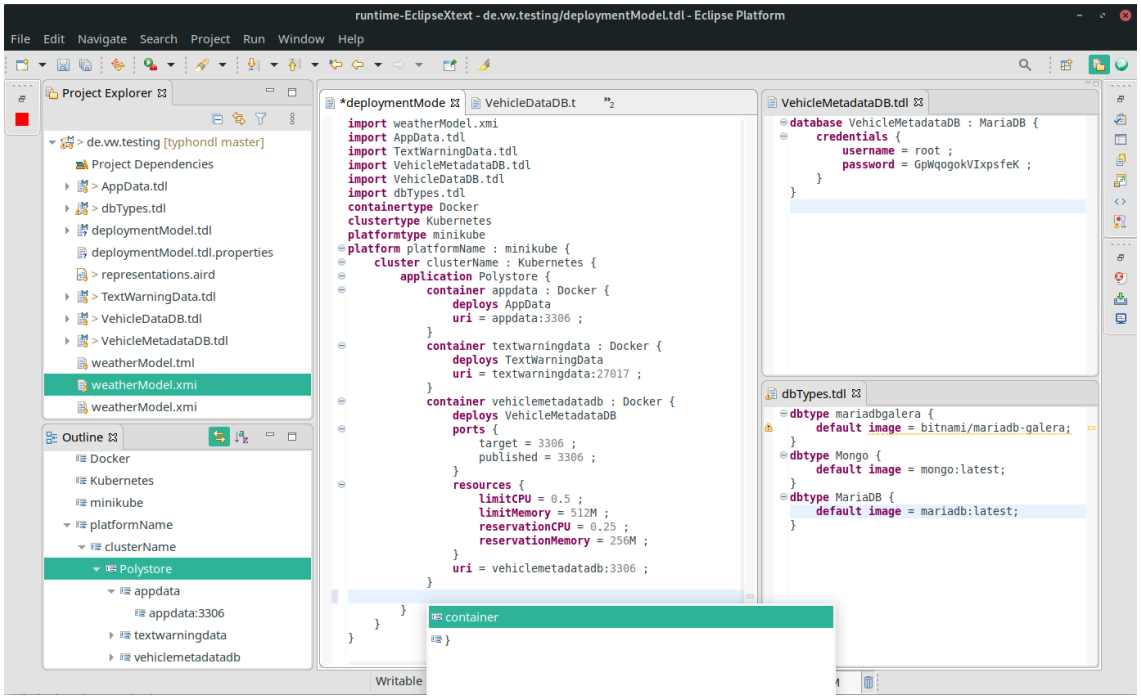

Figure 7: TyphonDL textual editor with syntax highlighting and auto completion

2.4 TyphonDL Script Generation and Running the Polystore

To create deployment scripts the TyphonDL Script Generator has to be started by selecting the created and completed DL model (main model file) and choosing *Generate Deployment Scripts* in the TyphonDL context menu. A folder with the name of the DL model is generated. It contains all files necessary to run the Polystore deployment.

1. If Docker Compose was chosen, a Service is created for every database and the Polystore can be started by running:

   ```
   $ docker-compose up -d
   ```

   If the DL model contains `Resources`, the Polystore has to be started by running

   ```
   $ docker stack deploy
   ```

   with Docker running in Swarm Mode. Otherwise the resource definition is ignored. The user can also setup Docker in Swarm Mode using multiple worker nodes and deploy the Polystore as a stack[23].

[23] https://docs.docker.com/engine/swarm/stack-deploy/

2. If Kubernetes was chosen, a Deployment and a Service to connect to the Pod(s) created by the Deployment is created for every database and the Polystore can be started by executing:

```
$ sh deploy.sh
```

The following sections contain example model objects and other properties set in the TyphonDL Creation Wizard and their impact on the generated Docker Compose and/or Kubernetes deployment scrips. The changes to the deployment script by adding or changing the model object are marked **bold**.

2.4.1 Container.name

The name of a `Container` is used for internal service discovery. It is part of the generated `Container.URI`, which the API uses to find all databases. The `URI` consists of the name of the container and the image's target port.

Model	*Docker Compose*	*Kubernetes*

container appdata : Docker
{…}
	services:
	  appdata:
	  …
		kind: Service
		apiVersion: v1
		metadata:
		  name: **appdata**

2.4.2 Container.Ports

By default the created TyphonDL models don't contain a `Ports` object. It can be added to publish a container/service:

Model	*Docker Compose*	*Kubernetes*

ports {
  target = 3306 ;
  published = 32123 ;
}
	ports:
	  - target: 3306
	  published: 32123
		kind: Service
		apiVersion: v1
		metadata:
		  name: appdata
		spec:
		  type: NodePort
		  ports:
		  - port: 3306
		  targetPort: 3306
		  nodePort: 32123
		  selector:
		  app: appdata-pod

2.4.3 Container.Resources

Model	*Docker Compose*	*Kubernetes*

resources {
  limitCPU = 0.5;
  limitMemory = 512M;
  reservationCPU = 0.25;
  reservationMemory = 256M;
}
	deploy:
	  resources:
	  limits:
	  cpus: '0.5'
	  memory: 512M
	  reservations:
	  cpus: '0.25'
	  memory: 256M
		resources:
		  limits:
		  memory: "512M"
		  cpu: "0.5"
		  requests:
		  memory: "256M"
		  cpu: "0.25"

2.4.4 Container.Replication

Model	*Docker Compose*	*Kubernetes*
In a MongoDB container: `replication { replicas = 3 ; mode = replicaSet ; }`	`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'` `  ]` Also a Dockerfile called rsinit in the folder vehicledatadb: `FROM mongo` `ADD vehicledatadb/init_set.sh /usr/local/bin/` `RUN chmod +x /usr/local/bin/init_set.sh` And a file to initiate the MongoDB ReplicaSet: `echo "sleeping for 10 seconds"` `sleep 10` `echo init_set.sh time now: `date +"%T" ` `mongo --host vehicledatadb:27017 <<EOF` ` var cfg = {` ` "_id": "vehicledatadbReplset",` ` "version": 1,` ` "members": [` ` {` ` "_id": 0,` ` "host": "vehicledatadb:27017"` ` }` ` ,{` ` "_id": 1,` ` "host": "vehicledatadb-replica1:27017"` ` }` ` ,{` ` "_id": 2,` ` "host": "vehicledatadb-replica2:27017"` ` }` ` ,{` ` "_id": 3,` ` "host": "vehicledatadb-replica3:27017"` ` }` `]` ` };` ` rs.initiate(cfg);` `EOF`	`Not supported`

Model	*Docker Compose (only in Swarm Mode)*	*Kubernetes*

In a stateless container:
 	
`replication { replicas = 3 ;`
	`typhonql-server:`
	`  deploy:`
	`  mode: replicated`
	`  replicas: 6`
		`apiVersion: apps/v1`
		`kind: Deployment`
		`metadata:`
		`  name: typhonql-server-`

`mode = stateless ;`
`}`
		`deployment`
		`spec:`
		`  replicas: 3`
		`…`

2.4.5 Container.Networks

The `Networks` object is used to introduce inside a container to specify the network it is part of. Script generation of `Networks` generates the Docker Compose keyword `networks`. For Kubernetes a new Kubernetes script kind for namespaces are manually added with the matching network name `<networkName>`.

Model	*Docker Compose*	*Kubernetes*

`networks <networkName>`	`networks:`
	`  - <networkName>:`
		`kind: Namespace`
		`metadata:`
		`  name: <networkName>`
		`spec: {}`
		`status: {}`

2.4.6 Container.Volumes

The `Volumes` object allows the user to specify volumes parameters for the directories in a container such as a volume name, mount path, volume type. `Properties` is used to add other technology specific volume parameters.

Model	*Docker Compose*	*Kubernetes*

`volumes {`
` volumeName = <volumeName>;`
` mountPath = <volumePath>;`
` volumeType = <volumeType>;`
`  <Properties>`
`}`
	Outside of a Docker Compose service
	
	`volumes:`
	` <volumeName>:`
		Outside of a Kubernetes container
		
		`volumes:`
		`  - name: <volumeName>`
		`  <volumeType>:`
		`  <Properties>`

	Inside a Docker Compose service
	
	`volumes:`
	`  - type: <volumeType>`
	`  source: <volumeName>`
	`  target: <volumePath>`
	`  <Properties>`
		Inside a Kubernetes container
		
		`volumeMounts:`
		`  - name: <volumeName>`
		`  mountPath: <volumePath>`

2.4.7 Container.Properties and DB.Properties

The `Properties` object allows the user to add additional database and container specifications without the need to extend DL (e.g. adding a `restart = always;`[24] `Key_Values` object to a container).

Model	*Docker Compose*	*Kubernetes*
`key = value;`	`key: value`	`key: value`

`keyValueList {`
`  key = value;`
`}`
	`keyValueList:`
	`  key: value`
		`keyValueList:`
		`  key: value`

`keyValueArray [`
`  value1, value2`
`]`
	`keyValueArray:`
	`  - value1`
	`  - value2`
		`keyValueArray:`
		`  - value1`
		`  - value2`

2.4.8 DB.Credentials

The translation of the model object `Credentials` to the database credentials set in a container's environment is `DBType` dependent:

DBType	DBTypeKey Username	DBTypeKey Password
Mongo	MONGO_INITDB_ROOT_USERNAME	MONGO_INITDB_ROOT_PASSWORD
MariaDB	-	MYSQL_ROOT_PASSWORD
Neo4j	-	NEO4J_AUTH
Cassandra	-	-

In the following table, the `DBTypeKeys` are substituted by the `DBType` dependent keys given in the table above.

Model	*Docker Compose*	*Kubernetes*

`credentials {`
`  username = <username> ;`
`  password = <password> ;`
`}`
	`environment:`
	`  (<DBTypeKey>: <username>)`
	`  <DBTypeKey>: <password>`
		`environment:`
		`  (- name: <DBTypeKey>`
		`  value: <username>)`
		`  - name: <DBTypeKey>`
		`  value: <password>`

If a Helm Chart is used, the `Credentials` are used in the install command (see 2.4.13):

DBType	DBTypeKey Username	DBTypeKey Password
Mongo/ Mongo-Sharded	-	mongodbRootPassword
MariaDB/ MariaDB-Galera	-	rootUser.password
Neo4j	-	neo4jPassword
Cassandra	dbUser.user	dbUser.password

[24] https://docs.docker.com/compose/compose-file/#restart

With this TyphonDL feature, the API is able to read the database credentials from the DL model without having to know about DBMS dependent syntax.

2.4.9 DB.IMAGE

If the `DB` contains already an `IMAGE` object, then this object is used over the `DBType.IMAGE` object.

2.4.10 DB.Environment

Model	*Docker Compose*	*Kubernetes*

`environment {`
`  MYSQL_DATABASE = admin;`
`}`
	`environment:`
	`  MYSQL_ROOT_PASSWORD: nR6dupglQ4FROOGWQ`
	`  MYSQL_DATABASE: admin`
		`environment:`
		`  - name: MYSQL_ROOT_PASSWORD`
		`  value: ADpmYZCED5xiAFSZ`
		`  - name: MYSQL_DATABASE`
		`  value: admin`

If the `DB` has `Credentials`, the `Environment` gets added to the environment.

2.4.11 DB.external

If a `DB` is set `external`, no deployment scripts are generated.

2.4.12 DB.URI

The `URI` of a `DB` is only set if the `DB` is `external`, so that the API can find the database.

2.4.13 DB.HelmList

Helm charts can only be used with Kubernetes. The "helm install" command depends on the helm chart used. The `HelmList` contains a **`repoName`**, a **`repoAddress`** and a **`chartName`**. It gets translated to:

$ helm repo add **`repoName repoAddress`**

$ helm install **`Container.name`** --set fullnameOverride=**`Container.name`** <setAdditions> **`repoName`**/**`chartName`** -n typhon

The <setAdditions> are `DBType` dependent and mainly contain `Credentials`:

DBType	<setAdditions>
Mongo/ Mongo-Sharded	--set mongodbRootPassword=<DB.credentials.password>
MariaDB/ MariaDB-Galera	--set rootUser.password=<aDB.credentials.password>
Neo4j	--set acceptLicenseAgreement=yes --set neo4jPassword=<DB.credentials.password>
Cassandra	-set dbUser.user=<DB.credentials.username>,dbUser.password=<DB.credentials.password>

If a `Key_Values` *valuesFile=pathToValues.yaml* is given (see Figure 6), then it is added to the `helm install` command.

3. IMPLEMENTATION

The first version of the tools' implementation[25] as Eclipse plugin was described in D3.2 (TYPHON Consortium, 2018) and is continued and completed here as the full prototype of the TyphonDL tools' implementation.

3.1 TYPHONDL TEMPLATES

The TyphonDL Templates are implemented by creating `XtextTemplatePreferencePages` provided by the Xtext plugin[26]. Default templates are provided in a *templates.xml* file[27](see Annex I – template.xml).

3.2 TYPHONDL CREATION WIZARD

The TyphonDL Creation Wizard[28] is implemented as an `org.eclipse.jface.Wizard` (see section 2.2, Figure 2 to Figure 6).

3.3 TYPHONDL SCRIPT GENERATOR

Before Acceleo[29] is used to generate the deployment scripts (as described in D3.2), the Polystore components (see 1.2) need to be added to the model. If the Analytics component is to be used with Kubernetes, Flink and Kafka deployment files are downloaded[30] and included in the project.

To upload the ML and DL model to the Typhon Metadata Database (which is a MongoDB database) automatically when using Docker Compose, a JavaScript file containing a mongo.insert(MLModel, DLModel) statement is created. By mounting that file's directory to the container's *docker-entrypoint-initdb.d* it gets executed when the container is first started. To add the models when using Kubernetes, a Job[31] containing the mongo.insert(MLModel, DLModel) statement is created.

After every Polystore component is added to the model[32], the deployment scripts get generated by using Acceleo[33].

A full deployment example can be found in the Typhon github repository, both for Docker Compose/Swarm[34] and Kubernetes[35].

[25] https://github.com/typhon-project/typhondl
[26] https://www.eclipse.org/Xtext/
[27] https://github.com/typhon-project/typhondl/blob/master/de.atb.typhondl.xtext.ui/templates/templates.xml
[28] https://github.com/typhon-project/typhondl/tree/master/de.atb.typhondl.xtext.ui/src/de/atb/typhondl/xtext/ui/creationWizard
[29] https://www.eclipse.org/acceleo/
[30] http://typhon.clmsuk.com/static/analyticsKubernetes.zip
[31] https://kubernetes.io/docs/concepts/workloads/controllers/job/
[32] Happens here: https://github.com/typhon-project/typhondl/blob/master/de.atb.typhondl.acceleo/src/de/atb/typhondl/acceleo/services/Services.java
[33] https://github.com/typhon-project/typhondl/tree/master/de.atb.typhondl.acceleo/src/de/atb/typhondl/acceleo/files

4. CONCLUSION

This document presented the work done in the TYPHON project in WP3, in particular in *T3.4 Assembly of Optimised Hybrid Polystore VMs from Deployment Models.*

The following Table 1 presents an overview of the requirements defined for TyphonDL in D1.1 and their implementation status.

Table 1: Overview of technology requirements and their implementation status

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)
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	SHOULD	Implemented for some

[34] https://github.com/typhon-project/typhondl/tree/master/demo.compose
[35] https://github.com/typhon-project/typhondl/tree/master/demo.kubernetes

ID	Requirement	Priority	Status
	consistency checks on the data can be supported.		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: Overview of industrial use case requirements and their implementation status

ID	Requirement	Priority	Status
40	The polystore deployment shall work with at least two containerization solutions	SHALL	Implemented
41	The polystore deployment should generate containers using Docker	SHOULD	Implemented
42	The outcome of the polystore deployment shall be containers ready to run without further configuration needed	SHALL	Implemented
43	The generated virtualized containers shall be tested in environments of at least two major online cloud providers: AWS, Google Cloud, Microsoft Azure, etc.	SHALL	Implemented
44	The containers shall automatically start the database nodes instance without human	SHALL	Implemented

ID	Requirement	Priority	Status
	intervention		
45	It should be possible to duplicate containers and boot them as additional nodes of the database instance without the need to modify the configuration	SHOULD	Implemented using Kubernetes
46	The polystore deployment should work with existing relational databases	SHOULD	Implemented

This document serves as description of this implementation given that the result of this task is actually the developed software (uploaded in GitHub). The present report documents the implementation of the TyphonDL tools. In particular it highlights the usage and implementation of the TyphonDL tools: Templates, Wizard, Editor, and Script Generation.

The first usage and testing of the TyphonDL tools have shown strengths and limitations of the prototype developed up to now:

- Strengths:
  - The TyphonDL tools are easy to use. The user does not have to be an Eclipse expert, a Kubernetes expert or a Docker expert to create deployment scripts.
  - Company specific database settings can be easily given by editing the TyphonDL Templates.
  - Two possible container options (Doker and Kubernetes) that cover/represent the majority of the approaches currently used in industry and is ready to be extended to further needed operations.
- Limitations:
  - Scaling databases in containers to adapt to large amounts of data is not straight forward. Using Helm Charts is a good solution for this difficult task. The Helm Charts can be configured by giving a custom *values.yaml*, this is not included in the TyphonDL plugin.

The outlook for the TyphonDL tools has several objectives:

- In a near future, within the Typhon project lifetime, the TyphonDL tools will be further optimised based on the use case evaluation test.
- In the medium to long future, the TyphonDL follows the Typhon project Open Source Strategy and will be therefore published in a github repository to be available to the wider community. ATB plans to offer customisation services to

the DL toolset. Tied to this, ATB as Eclipse Associate member will use this connection within the Eclipse community to further exploit the DL toolset.

5. BIBLIOGRAPHY

Budinsky, Steinberg, Merks, Ellersick, & Grose. (2003). *Eclipse Modelling Framework: a developer's guide.* . Boston M.A.: Addison-Wesley.

TYPHON Consortium. (2018). *D3.2 TyphonDL Tools.*

TYPHON Consortium. (2019). *D2.5 TyphonML Model Analysis and Reasoning Tools.*

TYPHON Consortium. (2019). *D3.2 TyphonDL Modeling Tools.*

TYPHON Consortium. (2019). *D5.5 Event Publishing and Monitoring Architecture (Final Version)* .

TYPHON Consortium. (2019). *D6.3 Hybrid Polystore Data Migration Tools.*

TYPHON Consortium. (2019). *D7.2 Integrated Platform - Interim Version.*

TYPHON Consortium. (2020). *D3.4 Hybrid Polystore Deployment Language (Final Version).*

6. ANNEX I – TEMPLATE.XML

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<templates>
<template id="dbType_mariadb" autoinsert="true" con-
text="de.atb.typhondl.xtext.TyphonDL.DBType" deleted="false" description="Default template
for dbtype MariaDB using the latest image from Docker Hub" enabled="true"
name="MariaDBType">dbtype MariaDB {
    default image = mariadb:latest;
}</template>
<template id="dbType_mongo" autoinsert="true" con-
text="de.atb.typhondl.xtext.TyphonDL.DBType" deleted="false" description="Default template
for dbtype Mongo using the latest image from Docker Hub" enabled="true"
name="MongoType">dbtype Mongo {
    default image = mongo:latest;
}</template>
<template id="dbType_mysql" autoinsert="true" con-
text="de.atb.typhondl.xtext.TyphonDL.DBType" deleted="false" description="Default template
for dbtype MySQL using the latest image from Docker Hub" enabled="true"
name="MySQLType">dbtype MySQL {
    default image = mysql:latest;
}</template>
<template id="dbType_cassandra" autoinsert="true" con-
text="de.atb.typhondl.xtext.TyphonDL.DBType" deleted="false" description="Default template
for dbtype Cassandra using the latest image from Docker Hub" enabled="true"
name="Cassandra">dbtype Cassandra {
    default image = cassandra:latest;
}</template>
<template id="dbType_neo4j" autoinsert="true" con-
text="de.atb.typhondl.xtext.TyphonDL.DBType" deleted="false" description="Default template
for dbtype Neo4j using the latest image from Docker Hub" enabled="true"
name="Neo4j">dbtype Neo4j {
    default image = neo4j:latest;
}</template>
<template id="db_mariadb" autoinsert="true" context="de.atb.typhondl.xtext.TyphonDL.DB"
deleted="false" description="default minimal template for MariaDB" enabled="true"
name="MariaDB">database ${databaseName} : MariaDB {
    environment {
        MYSQL_ROOT_PASSWORD = ${password} ;
    }
}
</template>
<template id="db_mongo" autoinsert="true" context="de.atb.typhondl.xtext.TyphonDL.DB"
deleted="false" description="default minimal template for Mongo" enabled="true"
name="Mongo">database ${databaseName} : Mongo {
```

```
    environment {
        MONGO_INITDB_ROOT_USERNAME = ${username} ;
        MONGO_INITDB_ROOT_PASSWORD = ${password} ;
    }
}</template>
<template id="db_mysql" autoinsert="true" context="de.atb.typhondl.xtext.TyphonDL.DB"
deleted="false" description="default minimal template for MySQL" enabled="true"
name="MySQL">database ${databaseName} : MySQL {
    environment {
        MYSQL_ROOT_PASSWORD = ${password} ;
    }
}</template>
<template id="db_cassandra" autoinsert="true" context="de.atb.typhondl.xtext.TyphonDL.DB"
deleted="false" description="default minimal template for Cassandra" enabled="true"
name="Cassandra">database ${name} : Cassandra {
}</template>
<template id="db_neo4j" autoinsert="true" context="de.atb.typhondl.xtext.TyphonDL.DB"
deleted="false" description="default minimal template for Neo4j" enabled="true"
name="Neo4j">database ${name} : Neo4j {
    environment {
        NEO4J_AUTH = neo4j/${password};
    }
}</template>
</templates>
```