Rail to Digital Automated up to Autonomous Train Operation

D31.3 – Demonstrator TCMS Data Service Formalisation

Due date of deliverable: 24/10/2025

Actual submission date: 24/10/2025

Leader/Responsible of this Deliverable: DB InfraGO AG

Reviewed: Y

Document status		
Revision	**Date**	**Description**
0.1	20/10/2025	Initial draft version of the document
0.2	24/10/2025	Finalized version. Chapter 2 completed. Title updated.

Project funded from the European Union's Horizon Europe research and innovation programme		
Dissemination Level		
PU	Public	x
SEN	Sensitive – limited under the conditions of the Grant Agreement	

Start date: 01/12/2022 Duration: 42 months

ACKNOWLEDGEMENTS

This project has received funding from the Europe's Rail Joint Undertaking (ERJU) under the Grant Agreement no. 101102001. The JU receives support from the European Union's Horizon Europe research and innovation programme and the Europe's Rail JU members other than the Union.

REPORT CONTRIBUTORS

Name	Company	Details of Contribution
Kai Schories	DB	Creator/Leader
Miguel Fernandez	CEDEX	Reviewer
Ignacio Alguacil	INECO	Reviewer
Jan Koning	NS	Reviewer
Thomas, Waschulzik	SMO	Reviewer

EXECUTIVE SUMMARY

This document provides the software delivery notes for the TCMS DataService (TCMS_DS) prototype, which is provided as deliverable D31.3 for integration and use in WP36 platform demonstrator [1].

The delivered software has been developed in-house by DB, mainly as a proof of concept for validating and demonstrating a formal TCMS data modelling and workflow automation approach.

The formal TCMS data modelling and workflow automation approach has been established in Task 31.2, and was jointly reviewed and documented in deliverable D31.2 [2], alongside with the software architecture of the TCMS_DS prototype.

The purpose of this document is to provide descriptions for delivered software components and guidelines for their integration and use in WP36 platform demonstrator.

The intended audience of this document are WP36 configuration managers, integrators and testers.

The targeted use case for the deployment of delivered software in WP36 platform demonstrator is DIAVEC/Brakes.

ABBREVIATIONS AND ACRONYMS

Abbreviation	Definition
ARCH	Architecture
API	Application Programming Interface
APP	Application
ATO	Automatic Train Operation
BCU	Brake Control Unit
CCS	Command, Control & Signaling
CCU	Command and Control Unit
CI/CD	Continuous Integration / Continuous Delivery
Dxx.x	Denotes an R2DATO deliverable, e.g. D31.2 or D36.1
DB	Deutsche Bahn
DEF	Definition
DEV	Device or Development, depend on context
DevOps	IT infrastructure and processes for automating and streamlining software development and deployment
DIA	Diagnostics
DiaLab	DB internal project and demonstrator environment
DIAVEC	Vehicle Diagnostics, DBS GoA4 function
DNS	Domain Name Service
DS	Data Service
EOL	End Of Lifecycle
FS	Functional Status or File System (depends on context)
GoA	Grade of Automation
GU(x)	Vendor specific term, synonym for BCU
HM	Health Monitor(ing)
HTTP	Hypertext Transfer Protocol
HW	Hardware
ID	Identifier
IP	Internet Protocol
JSON	JavaScript Object Notation
N/A	Not applicable
OB	Onboard

PC	Personal Computer
PID	Process identifier
POST	HTTP-POST request type
PubSub	Publish/Subscribe, communication pattern
R2DATO	Rail to Digital Automated up to Autonomous Train Operation
REPO	Repository
RootFS	Root File System
REST	Representational State Transfer
SRC	Source
SW	Software
SYS	System
TCMS	Train Control and Management System
TCMS_DS	TCMS Data Service
TCP	Transmission Control Protocol
TSI	Technical Specification for Interoperability
URL , URI	Uniform Resource Locator , Uniform Resource Identifier
UTC	Coordinated Universal Time
VLxx-yy	VLAN segment or node, e.g. VL100, or VL100-110
VLAN	Virtual Local Area Network
VM	Virtual Machine
WP	Workpackage
WS	Websocket protocol, when used in URLs like “ws:// …”
YANG	Yet Another Next Generation (a modular language representing data structures in an XML tree format)
YAML	YAML Ain't Markup Language

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 OVERVIEW

1.1 DOCUMENT PURPOSE

This document provides the software delivery notes for the TCMS DataService (TCMS_DS) prototype, which is provided as D31.3 deliverable for integration and use in WP36 platform demonstrator.

The delivered software has been developed in-house by DB, mainly as a proof of concept for validating the formal TCMS data modelling and workflow automation approach, that has been established in Task 31.2 and documented in deliverable D31.2 [2], alongside with the software architecture for the TCMS_DS prototype.

The purpose of this document is to provide descriptions for delivered software components and guidelines for their integration and use in WP36 platform demonstrator.

1.2 INTENDED AUDIENCE

The intended audience for this document are WP36 configuration managers, integrators and testers. Other potential readers are software engineers with background in TCMS, onboard platform architectures, and fault management.

1.3 SYSTEM CONTEXT

The target environment for the delivered TCMS_DS prototype software is the WP36 platform demonstrator [1], where the TCMS_DS prototype will be used in combination with other software for realizing the DIAVEC/Brakes use case. This use case has been derived from D36.1 user story “collect and aggregate health and performance information from vehicle”, defined in [3], ch. 2.5.10. Furthermore, this use case relates to D36.1 system capability “collection and usage of TCMS data for fault management use cases” from [4], ch. 3.2. The WP36 platform demonstrator setup for this use case is outlined [5] ch. 3.7.

1.4 PREREQUISITES

Before proceeding with this document, it is recommended to read the relevant chapters from prerequisite documents listed in Table 1 below.

Table 1 Prerequisite documents and relevant chapters to read beforehand

Topic	Reference	Chapters
WP31, Task 31.2 – Project background and motivation	D31.2 [2]	1,2
WP31, Task 31.2 – TCMS data modeling workflow	D31.2 [2]	3
WP31, Task 31.2 – TCMS_DS prototype architecture	D31.2 [2]	4
WP36, Task 36.1 – Demonstrator Specification (4 parts)		
- D36.1 - Statement of Work	D36.1 – SOW [1]	1.3, 1.4, 3.4.3
- D36.1 - User Stories & Test Cases	D36.1 – US/TC [3]	2.5.10
- D36.3 – Architecture	D36.1 – ARCH [5]	3.7, 3.8
- D36.4 – System Definition	D36.1 – SYSDEF [4]	3.1, 3.2

2 DELIVERY SCOPE

2.1 SOFTWARE COMPONENTS

The D31.3 delivery for WP36 platform demonstrator comprises the following software components.

Table 2 D31.3 delivery scope: software components

Component	Repository items	Component description
tcmsds-server	`./src/server/`	Sources for *tcmsds-server* runtime, implemented in python3, according to architecture from D31.2 [2], ch. 4.
tcmsds-extensions	`./src/models/`	YANG models for Brakes and Doors use cases, as well as generated *tcmsds-extensions* (datastore *objects*, and service *apispecs*), obtained from TCMS data modeling and automation workflow described in D31.2 [2], ch. 3
docker-build-scripts	`./Makefile` `./Dockerfile`	GNU/Make and Docker/BuildKit scripts for building and packaging *tcmsds* configurations as docker images; for use in WP36 DevOps build pipelines
docker-run-scripts	`./startc` `./stopc`	GNU/Bash scripts for proper startup (and cleanup) of *tcmsds* configurations as docker containers; for use in WP36 DevOps integration pipelines
docker-test-scripts	`./runtest`	GNU/Bash scripts for module-testing of *tcmsds* build artifacts; for use in WP36 DevOps build- and integration pipelines

2.2 CODE REPOSITORY

Provisioning and versioning of software for WP36 platform demonstrator is accomplished through the D31.3 code repository, which is part of WP36 Azure DevOps environment; see Figure 1 below.

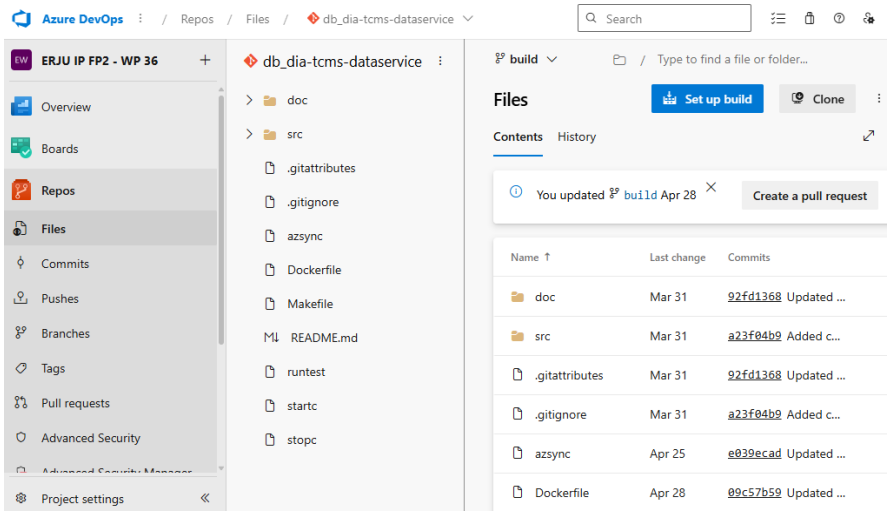

Figure 1 D31.3 delivery scope: code repository

2.3 .BUILD PIPELINES

D31.3 software delivery for WP36 is "source-only". Building and packaging *tcmsds* configurations as docker images for deployment in WP36 platform demonstrator is accomplished through *build pipelines* [6], defined in D31.3 code repository as *pipeline.yml* [7], [8].

D31.3 build pipelines use *docker-build-scripts* from D31.3 code repository in combination with build platform services from WP36 Azure DevOps environment. The main build platform service used by *docker-build-scripts* is the Linux-VM pool [9], where Ubuntu-22.04 LTS is chosen as build-VM. So, *docker-build-scripts,* invoked from D31.3 build pipelines, will run in a virtual Linux/Ubuntu-22.04 environment that provides the required build tools such as *bash, make, docker*, and *git.*

Figure 2 depicts an extract of the pipeline.yml used for building the *tcmsds-brakes* docker image.

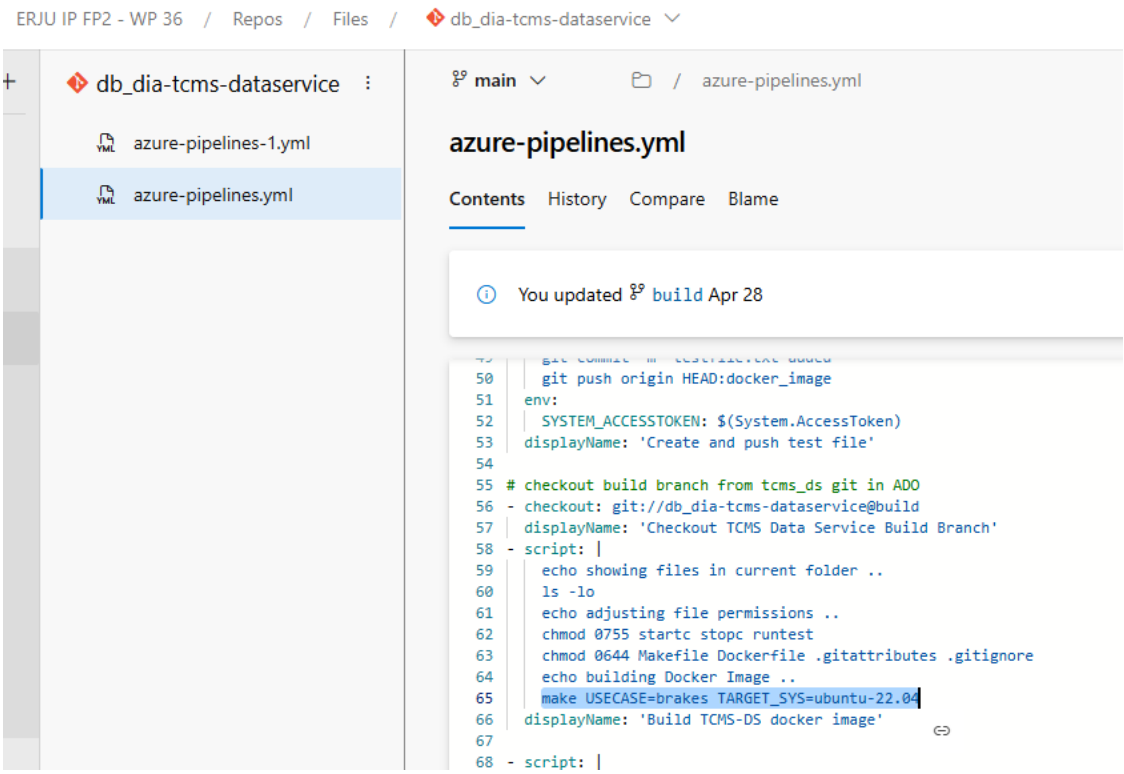

Figure 2 Extract of D31.3 pipeline.yml from *tcmsds-brakes* build pipeline

Figure 2 (pipeline.yml, L:55-L:65) shows how the build-branch from D31.3 code repo is fetched into the build-VM's workspace, and how *docker-build-scripts* are invoked to build the *tcmds-brakes* configuration as docker image; see next section 2.4 for further information.

2.4 DOCKER IMAGES

The deployment of *tcmsds* configurations for WP36 platform demonstrator shall be realized as docker images. For that purpose, the D31.3 code repository provides the *docker-build-scripts*; see section 2.2. These build scripts shall be used in D31.3 build pipelines as outlined in previous section 2.3.

The remaining paragraphs of this section describe the use of D31.3 *docker-build-scripts* in Linux/Ubuntu-22.04 (or similar) build-environments, and describe available build options and resulting build artifacts.

Docker-build-scripts are invoked from a shell prompt in D31.3 code repo root folder as follows.

```
$ make USECASE={brakes|doors} TARGET_SYS={ubuntu-18.04|ubuntu-22.04}
```

The above cmdline will use the *Makefile* script from D31.3 code repo and executes *make all* with build parameters USECASE=[…] and TARGET=[…]. *Make all* triggers the execution of *make docker-image*, which runs *docker-build* and *docker-export* recipes from *Makefile* script; see Figure 3 below.

```
75
76  # Default target
77  .PHONY: all
78  all :: docker-image
79
80  # Top-level build target
81  .PHONY: docker-image
82  # docker-image :: gitupdate configure check-rebuild docker-build docker-export
83  docker-image :: configure check-rebuild docker-build docker-export
84
```

Figure 3 D31.3 *Makefile*: *docker-image* rule, used as default

The *docker-build* recipe from D31.3 *Makefile* script executes the *docker build* command [10] with arguments derived from user-defined parameters and project-specific meta-information; see Figure 4 below.

```
168 # Build the Docker image
169 %.build.stamp: $(DOCKERFILE) $(APPCFG) $(APPREV)
170     @echo "Building Docker image $(APPNAME) ..."
171     @docker rmi -f $(APPNAME) > /dev/null 2>&1
172     docker buildx build --rm \
173         --build-arg maintainer_name=$(ORGNAME) \
174         --build-arg maintainer_email=$(ORGEMAIL) \
175         --build-arg description=$(DESCRIPTION) \
176         --build-arg appname=$(APPNAME) \
177         --build-arg version=$(APPVER)-$(TARGET_ARCH)-$(TARGET_SYS) \
178         --build-arg imagetag=$(APPIMAGE) \
179         --build-arg base_image=$(BASEIMAGE) \
180         --build-arg build_user=`id -un` \
181         --build-arg build_host=`hostname` \
182         --build-arg build_timestamp="`date`" \
183         --build-context appsrc=$(APPSRC) \
184         --tag $(APPIMAGE) \
185         -f $(DOCKERFILE) .
186     @echo "Building Docker image $(APPIMAGE) completed"
187     @docker image ls $(APPIMAGE)
188     @echo $(shell date +"%Y-%m-%dT%H:%M:%S%z") > $@
189
```

Figure 4 D31.3 Makefile: invocation of *docker build* command

The most important arguments used in *docker-build* recipe from Figure 4 are: `--tag $(APPIMAGE)`, `-f $(DOCKERFILE)`, `--build-arg baseimage=$(BASEIMAGE)`, and `--build-context appsrc=$(APPSRC)`. Argument `-f $(DOCKERFILE)` tells *docker build* to use instructions from D31.3 *Dockerfile* script to create the *tcmsds* docker image. *A*rgument `--build-arg`

`baseimage=$(BASEIMAGE)` defines the base image to be used as *rootfs*-layer for *tcmsds* docker image. Argument `--build-context appsrc=$(APPSRC)` is used to tell *docker build* command from where to copy the *tcmsds-server* and *tcmsds-extensions* components when creating the *app*-layer for *tcmsds* docker image. Argument : `--tag $(APPIMAGE)` holds the full image tag to be used for the resulting *tcmsds* docker image; see [11] for details regarding the format of docker image tags .

The main output of D31.3 *docker-build-scripts*, when invoked via *make* cmdline from p. 11, is the docker image, as compressed tar-archive, named: *tcmds-{brakes | doors}.tar.xz.* In addition to this, three identically named build stamps will be generated as small text files, containing useful build information; see Table 3 below.

Table 3 Build artefacts produced by D31.3 *docker-build-scripts*

A successful `make` will produce the following build artifacts in workspace folder.

filename	description
`tcmsds-<usecase>.*`	all build artifacts have this file prefix
`*.tar.xz`	docker image, saved as compressed tarball
`*.build.stamp`	contains UTC build time stamp in `ISO 8601` format
`*.build.scmref`	contains `tcmsds` appsrc version info
`*.image_name`	contains full docker image name and tag

Possible *tcmsds* configurations that can be built as docker image, using D31.3. *docker-build-scripts*, are listed in **Table 4** below.

Table 4 Possible *tcmsds* docker images supported by D31.3 *docker-build-scripts*

Valid combinations of `make` params `USECASE` and `TARGET_SYS` are listed in the below table, together with their resulting docker images.

USECASE	TARGET_SYS	pyenv	resulting docker image-name:tag
brakes	`ubuntu-22.04`	`py39`	`dialab/tcmsds-brakes:0.1-x86_64-linux_ubuntu-22.04`
	`ubuntu-18.04`	`py37`	`dialab/tcmsds-brakes:0.1-x86_64-linux_ubuntu-18.04`
doors	`ubuntu-22.04`	`py39`	`dialab/tcmsds-doors:0.1-x86_64-linux_ubuntu-22.04`
	`ubuntu-18.04`	`py37`	`dialab/tcmsds-doors:0.1-x86_64-linux_ubuntu-18.04`

The *tcmsds* docker images, produced by D31.3 *docker-build-scripts,* are using the minimized Ubuntu-22.04 rootfs from docker base image *phusion/baseimage:jammy-1.0.4*; see [12] for details. The python39 environment, which used to run the D31.3 *tcmsds-server* prototype on top of the

rootfs, is installed as *condaenv* [13], whose package configuration is defined by D31.3 config file *src/server/condaenv39.yml*.[1]

For historical reasons, an alternative *tcmsds* image variant, that uses a minimized Ubuntu-18.04 rootfs and python37 environment, can be built with *D31.3 docker-build-scripts*. This alternative image variant is, however, not applicable for WP36 platform demonstrator, and should not be used at all. Moreover, standard support for both Ubuntu-18.04 and Python37 has reached EOL.[2]

2.5 DOCKER CONTAINERS

D31.3 delivery provides *docker-run-scripts* as means to setup and run *tcmsds* configurations, that have been built and packaged as docker images (by using the D31.3 *docker-build-scripts*, as described in previous section 2.4).

The *startc* und *stopc* scripts from D31.3 code repo are essentially wrappers for *docker-cli* [14], and mainly implement the *docker run* and *docker stop* cmdlines for proper *tcmsds* container startup and cleanup. Furthermore, the *startc* script deals with *docker network* setup [15], which is needed for using *tcmsds* container services in combination with other docker containers; for example, the use in combination with the *nodered* container in WP36 DIAVEC/Brakes deployments.

The basic use of *startc* and *stopc* scripts is outlined in Figure 5 below for *tcmsds-brakes* configuration. In this example, it is assumed that the *tcmsds-brakes* image file and associated build stamps (see Table 3 from section 2.4) are located next to *startc* and *stopc* scripts (same folder).

```
user@BWP:~$ cd ~/workspace/tcmsds_docker
user@BWP:~$ ./startc brakes   # start tcmsds-brakes container in Docker Desktop
user@BWP:~$ docker cp tcmsds-brakes:/var/run/tcmsds.pid - # display tcmsds-brakes PID
user@BWP:~$ cat "/var/run/user/$(id -u)/TCMSDS_BRAKES_UPDATE_SERVICE" # get update service url
user@BWP:~$ cat "/var/run/user/$(id -u)/TCMSDS_BRAKES_PUBSUB_SERVICE" # get pubsub service url
user@BWP:~$ ./runtest brakes  # run smoketest, also calls startc/stopc as needed
user@BWP:~$ docker cp tcmsds-brakes:/var/log/tcmsds.log - # display tcmsds-brakes logfile
user@BWP:~$ ./stopc brakes    # stop tcmsds-brakes container in Docker Desktop
```

Figure 5 D31.3 *docker-run-scripts*, usage example, *tcmsds-brakes*

With respect to Figure 5, the cmdline `./startc brakes` executes *startc* script with args `$1: brakes`, which tells *startc* to setup and run *tcmsds-brakes* configuration as container in local docker backend, using image file `./tcmsds-brakes.tar.xz` and image tag according to contents from `./tcmsds-brakes.image_name` .

The image tag info is used by *startc* script to check whether the *tcmsds-brakes* image was already loaded into docker backend; see also the *docker load* and *docker ls* command reference from [14]. If so, the already loaded image will be re-used for *tcmds-brakes* container startup.[3]

[1] Note: required python39 conda packages are installed from conda-forge channel to avoid potential licensing issues; see https://conda-forge.org/blog/2020/11/20/anaconda-tos/ and https://www.anaconda.com/blog/is-conda-free for details.
[2] See https://devguide.python.org/versions/ and https://ubuntu.com/about/release-cycle
[3] Extracting and loading docker images from compressed archives takes some time, depending on the actual image size. Once loaded, images will be cached in docker backend and reused to speedup subsequent runs.

After loading the image file into docker backend, using *docker load* command, *startc* calls *docker run* according to Figure 6, with parameters according to Figure 7, in order to startup the *tcmsds* configuration (*tcmsds-brakes* in the above example) as docker container.

startc and *stopc* scripts are used from a shell prompt in D31.3 code repo root folder as follows.

```
$ ./startc {brakes|doors}  # env override: $APPNAME, $IMGFILE, $IMGNAME, $UPDATE_HOST, $PUBSUB_HOST
$ ./stopc {brakes|doors}
```

For certain *tcmsds* deployments, the shell variables, listed in the comment part from above *startc* cmdline, can be used to override the default settings applied by *startc* as *docker run* parameters (see Figure 7). One example, where this is actually needed for *tcmsds* deployment in WP36 platform demonstrator, is given at the end of this section.

```
115  # Test if app container is already running in docker engine
116  CONTID="$(docker ps -q -f "name=${APPNAME}")"
117  [[ -n "${CONTID}" ]] && {
118      echo "Already started, docker container ${APPNAME} (imgname: ${IMGNAME})"
119      exit 0
120  }
121
122  # Start app container in docker engine
123  echo -n "Starting docker container ${APPNAME} (imgname: ${IMGNAME}) ... "
124  docker run ${RUNARGS} "${IMGNAME}" > /dev/null
125  sleep 5
126  CONTID="$(docker ps -q -f "name=${APPNAME}")"
127  [[ -n "${CONTID}" ]] || {
128      echo "Error: failed starting docker container ${APPNAME}"
129      exit 1
130  }
131  echo "Done"
132
```

Figure 6 D31.3 *startc* script: *docker run* invocation

```
63  # TCMS DataService URLs
64  UPDATE_HOST=${UPDATE_HOST:=$(host "update")}
65  PUBSUB_HOST=${PUBSUB_HOST:=$(host "pubsub")}
66  UPDATE_PORT=${UPDATE_PORT:=$(port "update")}
67  PUBSUB_PORT=${PUBSUB_PORT:=$(port "pubsub")}
68  UPDATE_URL=${UPDATE_URL:="http://${UPDATE_HOST}:${UPDATE_PORT%%"/tcp"}"}
69  PUBSUB_URL=${PUBSUB_URL:="ws://${PUBSUB_HOST}:${PUBSUB_PORT%%"/tcp"}"}
70
71  # Docker run params
72  RUNARGS="-d --rm \
73      --name ${APPNAME,,} \
74      --hostname ${APPNAME,,} \
75      --network ${USECASE,,} \
76      --network-alias $(echo ${APPNAME,,} | cut -d- -f1) \
77      -p ${UPDATE_URL##*://}:8080 \
78      -p ${PUBSUB_URL##*://}:8181"
79
80  # See https://docs.docker.com/reference/cli/docker/container/run/#add-host
81  RUNARGS="${RUNARGS} --add-host host.docker.internal=host-gateway"
82
83  # See https://github.com/docker/for-linux/issues/1374#issuecomment-2122562014
84  RUNARGS="${RUNARGS} --sysctl=net.ipv6.conf.all.disable_ipv6=1"
85
86  # See https://stackoverflow.com/a/57607433
87  RUNARGS="${RUNARGS} -e TZ=$(cat /etc/timezone)"
```

Figure 7 D31.3 *startc* script: *docker run* parameter assignments

The interesting parts from *startc* code snippets in Figure 6 and Figure 7, are the *docker run* cmdline (Figure 6, L:124) and the `$RUNARGS` assignments (Figure 7, L:72 .. L:87).

Considering the example from Figure 5, the *docker run* cmdline from Figure 6 will create a new docker container named *tcmsds-brakes* from image tag *dialab/tcmsds-brakes:0.1-x86_64-linux_ubuntu-22.04* (which is the id-string used by docker backend to identify the loaded *tcmsds-brakes* image; see also Table 4 from section 2.4). Furthermore, in this example, the `$RUNARGS` assignments for *docker run* cmdline will use the default settings for *tcmsds* container startup.

Default settings for *tcmsds* container startup are defined in *startc* script according to Table 5.

Table 5 D31.3 *startc* script: *docker run* default settings

The following table lists the `docker run` options used as defaults by `startc` script.

docker run arg	setting	description
-d -rm	--	`run` container as daemon process, destroy container on `docker stop`
--name	*tcmsds-<usecase>*	container name, either *tcmsds-brakes* or *tcmsds-doors*
--hostname	*tcmsds-<usecase>*	docker-internal DNS name, used to resolve container IP address
--network	<usecase>	user-defined docker network (bridge) the container will be attached to
--network-alias	*tcmsds*	docker network local DNS alias, container node-name
-p *:8080	see published services	published port mapping for service `update.tcmsds-<usecase>`
-p *:8181	see published services	published port mapping for service `update.tcmsds-<usecase>`

The last two rows from Table 5 list the *docker run* arguments determining the published port mappings [16] of *tcmsds* container services, i.e. the TCP port mappings for the *update-service* (http://*:8080) from *tcmsds-backend* interface, and the *pubsub-service* (ws://*:8181) from *tcmsds-frontend* interface.

When using *tcmds* container services from other docker containers, attached to the same docker network, e.g. when subscribing to *tcmsds-brakes* container's *pubsub-service* from *nodered-brakes* container via the user-defined, docker-internal *brakes* network, then the TCP port numbers used in request-URLs for *tcmsds update-* and *pubsub-service* will be unmapped, i.e. 8080/tcp and 8181/tcp.

For deployments, where both *tcmsds-brakes* and *tcmsds-doors* containers will be used in parallel on the same machine, and container services shall be exposed to native clients, e.g. system simulations sending state updates to *tcmds* containers, or native dashboards applications subscribing to *tcmsds* event streams, then the published port mapping for *tcmsds* containers applies and must be used to avoid port allocation conflicts in localhost network.

The published port mapping, which is used by *startc* script as default settings for *tcmsds* container startup, is shown in Table 6 below. This mapping is applied by *startc* script in *docker run* cmdline via `$RUNARGS` variable; see also Figure 6, Figure 7 and Table 5.

Table 6 D31.3 *startc* script: published port default settings

The following table lists the published ports mapping used as defaults in `startc` script.

usecase	service	host-url	port mapping	comments
brakes	UPDATE	http://localhost:8780	8780:8080/tcp	`REST` service for posting `brakes` updates
brakes	PUBSUB	ws://localhost:8881	8881:8181/tcp	`websocket` for `brakes` event subscriptions
doors	UPDATE	http://localhost:9380	9380:8080/tcp	`REST` service for posting `doors` updates
doors	PUBSUB	ws://localhost:9481	9481:8181/tcp	`websocket` for `doors` event subscriptions

By default, the scope (i.e. the visibility) of *tcmsds* container published ports is restricted to *localhost* network respectively the *loopback* interface (127.0.0.1). So, by default, published *tcmsds* container services (e.g. *update.tcmsds-brakes* at port 8780/tcp) can be only used from host-local native clients, but not from remote clients running on other network nodes. For WP36 DIAVEC/Brakes use case, *tcmds-brakes* container services shall be accessible from remote clients as well.

Changing the default scope for *tcmsds* published port mapping can be accomplished by means of custom assignments for shell variables `$UPDATE_HOST` and `$PUBSUB_HOST`, that will be processed by *startc* script, as shown above in Figure 7.

By using the *startc* cmdline from Figure 8 below, one can dynamically override the default scope for *tcmsds* container published port mapping to meet the designated deployment for WP36 DIAVEC/Brakes use case, which is shown in Figure 9 below.

Note: the `$HOSTALIASES` variable in *startc* cmdline from Figure 8 is used to augment the hostname resolution of the system with user defined values; see [17] for details.

```
echo -en "vl100-110 10.36.100.110\n" > "${netcfg:=$(mktemp)}"
UPDATE_HOST="localhost" PUBSUB_HOST="vl100-110" HOSTALIASES="${netcfg}" ./startc brakes
```

Figure 8 D31.3 *startc* script: published port settings for WP36 DIAVEC/Brakes use case

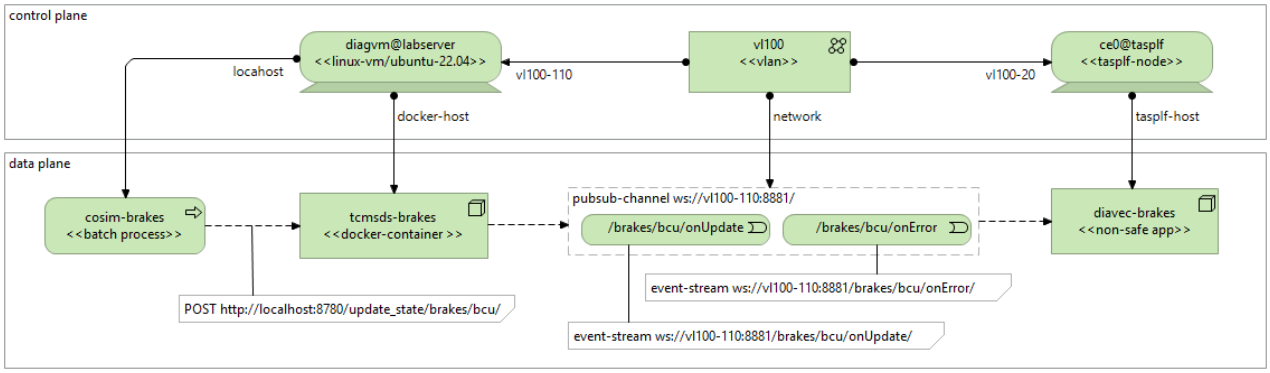

Figure 9 *tcmsds-brakes* deployment for WP36 DIAVEC/Brakes use case

2.6 MODULE TESTS

The D31.3 software delivery includes the *docker-test-scripts* component, comprising a single shell script, named *runtest*, which can be found in the root folder of D31.3 code repository.

The main purpose and scope of the provided *runtest* script is to perform basic module testing for *tcmsds* configurations, built and packaged as docker images in WP36 build pipelines, before deploying build artefacts in WP36 platform demonstrator.

In Figure 5 from previous section 2.5, the use of *runtest* is shown for the example of *tcmsds-brakes* configuration. The general use of *runtest* script from a shell prompt in D31.3 code repo root folder is shown below. It is assumed here that the *tcmsds* docker image of the configuration to be tested, as well as the associated build stamps, are located next to the *runtest* script (same folder), which is the case in D31.3 build pipelines (after successful *make* step); see also section 2.3.

```
$ ./runtest {brakes|doors}      # env override: $APPNAME, $IMGFILE, $IMAGENAME
```

When *runtest* is executed, it first checks whether the *tcmsds* configuration to be tested is already running as container in local docker backend. If not, then *runtest* calls the *startc* script to setup and start the *tcmsds* configuration as new container in local docker backend; see also section 2.5. Otherwise, *runtest* will use the existing *tcmsds* container as test target. The latter case applies for the usage example in Figure 5 from section 2.5.

Technical remarks: The *tcmsds-server* component will be automatically started as container-internal daemon process during container startup, and will be automatically restarted after internal errors; see [12] for details about how this can be accomplished; see also the *Dockerfile* from D31.3 code repo, where the container init functions are defined, using the method described in [12]. So, after successful *tcmsds* container startup, the *tcmsds-server* component should be up and running as container-internal daemon process. In the usage the example from Figure 5, one can see how to test the run state of the container-internal *tcmsds-server* daemon by means of checking the state of

the *server-pid* file from container filesystem path, i.e. `tcmsds-{usecase}:/var/run/tcmsds.pid`. If the container-internal *tcmsds-server* daemon is running, then the *server-pid* file exists and contains a valid *pid* (process-id; valid wrt. the container's process namespace). Otherwise, if the container-internal *tcmsds-server* daemon stops running and exits for some reason, then the *server-pid* file will be immediately removed, and a subsequent restart of the *tcmsds-server* will be triggered after a delay of 1sec. Such exceptional restarts might be monitored and reported as *server-pid* changes during expected uptime-periods.

As initial test step, *runtest* triggers a *tcmsds* datastore reset by sending a POST *load_state* request to the *tcmsds update-service* (container port 8080/tcp), which initializes/resets the hosted datasets (e.g. *brakes/bcu* states) with their default values; see also D31.2 [2], ch. 3.2.1, p. 23 f.

Figure 10 depicts the implementation of *runtest* script functions `http-post` and `tcmdsReset`, used for sending HTTP POST requests to *tcmsds update-service*, and for triggering *tcmsds* datastore resets.

```
36  # Lookup host urls of tcmsds services
37  TCMSDS_UPDATE_SERVICE=$(cat /var/run/user/$(id -u)/TCMSDS_${USECASE^^}_UPDATE_SERVICE)
38  TCMSDS_PUBSUB_SERVICE=$(cat /var/run/user/$(id -u)/TCMSDS_${USECASE^^}_PUBSUB_SERVICE)
39
40  # Send HTTP POST request to app service
41  # $1: http://host-ip/service-endpoint/
42  # $2: json-encoded data string, might be empty
43  function http-post () {
44      local endpointUrl="$1"
45      local jsonData="$2"
46      local httpResponse
47      echo ">> ${endpointUrl//\\/} >> ${jsonData}"
48      httpResponse=$(curl "${endpointUrl}" -H 'Content-Type: application/json' -d "${jsonData}" -s -w "%{response_code}")
49      echo "<< ${httpResponse}"
50      return $?
51  }
52
53  # Send tcmsds/load_state request
54  function tcmsdsReset () {
55      http-post "${TCMSDS_UPDATE_SERVICE}/load_state/" ""
56      return $?
57  }
58
```

Figure 10 D31.3 *runtest* script: *http-post* and *tcmsdsReset* implementations

If the status code returned by POST *load_state* request is `HTTP_OK` (200), then the initial test step is passed, which constitutes a successful "smoke test" for *tcmsds-server* startup (within the docker container) and the communication with *tcmsds update-service* (through its published container port) on localhost network.

Subsequent test steps, performed by *runtest* script, will test the processing of consecutive state updates in *tcmsds-server* backend. State updates are sent by *runtest* script as individual POST *update_state* requests to *tcmsds update-service*, using the `http-post` function from Figure 10 (in this example, via `brakesUpdateState` wrapper, which is implemented similar to `tcmsdsReset`). The sequence of state updates, which is used by *runtest* script for *tcmsds-brakes* configuration, is shown in Figure 11 below. For details about the JSON format and contents of payload objects used in *update_state* requests, refer to D31.2 [2], ch.3.

```
$ runtest
if [ "${USECASE}" == "brakes" ] ; then
    # BrakeState definitions
    S01='{"base":{"fs":"inop","hm":"noerr"},"stat":{"sb":"unknown","pb":"released"},"perfm":{"brkp":"0","perf":"0"}}'
    S02='{"base":{"fs":"op","hm":"noerr"},"stat":{"sb":"selfTest","pb":"engaged"},"perfm":{"brkp":"0","perf":"0"}}'
    S03='{"base":{"fs":"op","hm":"noerr"},"stat":{"sb":"holdingBrake","pb":"engaged"},"perfm":{"brkp":"140","perf":"100"}}'
    S04='{"base":{"fs":"op","hm":"noerr"},"stat":{"sb":"holdingBrake","pb":"released"},"perfm":{"brkp":"140","perf":"100"}}'
    S05='{"base":{"fs":"op","hm":"noerr"},"stat":{"sb":"noServiceBrake","pb":"released"},"perfm":{"brkp":"140","perf":"100"}}'
    S06='{"base":{"fs":"op","hm":"noerr"},"stat":{"sb":"serviceBrake","pb":"released"},"perfm":{"brkp":"140","perf":"100"}}'
    S07='{"base":{"fs":"op","hm":"noerr"},"stat":{"sb":"noServiceBrake","pb":"released"},"perfm":{"brkp":"140","perf":"100"}}'
    S08='{"base":{"fs":"op","hm":"noerr"},"stat":{"sb":"serviceBrake","pb":"released"},"perfm":{"brkp":"140","perf":"100"}}'
    S09='{"base":{"fs":"op","hm":"noerr"},"stat":{"sb":"noServiceBrake","pb":"released"},"perfm":{"brkp":"140","perf":"100"}}'
    S10='{"base":{"fs":"op","hm":"noerr"},"stat":{"sb":"slowingBrake","pb":"released"},"perfm":{"brkp":"140","perf":"100"}}'
    S11='{"base":{"fs":"op","hm":"noerr"},"stat":{"sb":"stoppingBrake","pb":"released"},"perfm":{"brkp":"140","perf":"100"}}'
    S12='{"base":{"fs":"op","hm":"noerr"},"stat":{"sb":"holdingBrake","pb":"released"},"perfm":{"brkp":"140","perf":"100"}}'
    S13='{"base":{"fs":"op","hm":"noerr"},"stat":{"sb":"holdingBrake","pb":"engaged"},"perfm":{"brkp":"140","perf":"100"}}'
    S14='{"base":{"fs":"inop","hm":"noerr"},"stat":{"sb":"unknown","pb":"engaged"},"perfm":{"brkp":"0","perf":"0"}}'

    # Test sequence: /update_state/brakes/bcu[id=GU1]/
    echo "Running test sequence ... "
    tcmsdsReset ; sleep 1
    brakesUpdateState "${GU1}" "${S01}" ; sleep 1
    brakesUpdateState "${GU1}" "${S02}" ; sleep 1
    brakesUpdateState "${GU1}" "${S03}" ; sleep 1
    brakesUpdateState "${GU1}" "${S04}" ; sleep 1
    brakesUpdateState "${GU1}" "${S05}" ; sleep 1
    brakesUpdateState "${GU1}" "${S06}" ; sleep 1
    brakesUpdateState "${GU1}" "${S07}" ; sleep 1
    brakesUpdateState "${GU1}" "${S08}" ; sleep 1
    brakesUpdateState "${GU1}" "${S09}" ; sleep 1
    brakesUpdateState "${GU1}" "${S10}" ; sleep 1
    brakesUpdateState "${GU1}" "${S11}" ; sleep 1
    brakesUpdateState "${GU1}" "${S12}" ; sleep 1
    brakesUpdateState "${GU1}" "${S13}" ; sleep 1
    brakesUpdateState "${GU1}" "${S14}" ; sleep 1
    echo "Done."
fi
```

Figure 11 D31.3 *runtest* script: sending consecutive brake-state updates

The current implementation of *runtest* script, as delivered in D31.3 for WP36 platform demonstrator, does not perform exhaustive testing of all possible state updates and error reports in scope of *tcmsds-brakes* and *tcmsds-doors* configurations. Only some state updates for *tcmsds-brakes* and *tcmsds-doors* configurations are tested here; the processing of error reports is not tested at all.

When it comes to testing the proper functioning of *tcmsds pubsub-service*, i.e. testing the event-based notification of *tcmsds* subscribers (agents for operator-specific fault management applications, etc.) about changes in hosted datasets (after state updates or error reports triggered by system resources from TCMS domain), there is currently no such test implemented in *runtest* script. These kind of tests will be performed as part of WP36 integration tests, planned for *tcmsds* deployments for WP36 DIAVEC/Brakes use case, according to Figure 9 from previous section 2.5.

For the set of integration tests, planned for WP36 DIAVEC/Brakes use case (involving deployments of the TCMS_DS software prototype from D31.3 delivery in WP36 platform demonstrator), see user story from [3], ch. 2.5.10, and derived test cases from report D36.4 [18]. The latter document is currently work in progress, and planned to be released in 04/2026.

REFERENCES

[1] R2DATO, "D36.1 – Demonstrator Specification, Statement of Work," R2DATO project, 2023.

[2] R2DATO, "D31.2 – Validation of TCMS Data Service concept and formalisation," R2DATO project, 2025.

[3] R2DATO, "D36.1 – Demonstrator Specification, User Stories & Test Cases," R2DATO project, 2023.

[4] R2DATO, "D36.1 – Demonstrator specification, System Definition," R2DATO project, 2025.

[5] R2DATO, "D36.1 – Demonstrator Specification, Architecture," R2DATO project, 2023.

[6] Microsoft, "What is Azure Pipelines?," 2025. [Online]. Available: https://learn.microsoft.com/en-us/azure/devops/pipelines/get-started/what-is-azure-pipelines?view=azure-devops. [Accessed 19 10 2025].

[7] Microsoft, "Key Azure Pipeline concepts," 2025. [Online]. Available: https://learn.microsoft.com/en-us/azure/devops/pipelines/get-started/key-pipelines-concepts?view=azure-devops. [Accessed 19 10 2025].

[8] Micosoft, "YAML schema reference for Azure Pipelines," 2025. [Online]. Available: https://learn.microsoft.com/en-us/azure/devops/pipelines/yaml-schema/?view=azure-pipelines&viewFallbackFrom=azure-devops. [Accessed 19 10 2025].

[9] Microsoft, "Microsoft-hosted agents," 2025. [Online]. Available: https://learn.microsoft.com/en-us/azure/devops/pipelines/agents/hosted?view=azure-devops&tabs=linux-images%2Cyaml#software. [Accessed 19 10 2025].

[10] Docker Inc., "Docker Build Overview," 2025. [Online]. Available: https://docs.docker.com/build/concepts/overview/. [Accessed 20 10 2025].

[11] Docker Inc., "Build, tag and publish an image," 2025. [Online]. Available: https://docs.docker.com/get-started/docker-concepts/building-images/build-tag-and-publish-an-image/#tagging-images. [Accessed 20 10 2025].

[12] Phusion B.V, "github.com/phusion/baseimage-docker," 2025. [Online]. Available: https://github.com/phusion/baseimage-docker?tab=readme-ov-file#adding-additional-daemons. [Accessed 23 10 2025].

[13] Anaconda Inc., "Managing environments," 2017. [Online]. Available: https://docs.conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html. [Accessed 23 10 2025].

[14] Docker Inc., "Dockerdocs: docker-cli reference," 2025. [Online]. Available: https://docs.docker.com/reference/cli/docker/. [Accessed 20 10 2025].

[15] Docker Inc., “Dockerdocs: Networking overview,” 2025. [Online]. Available: https://docs.docker.com/engine/network/. [Accessed 20 10 2025].

[16] Docker Inc., “Dockerdocs: Publishing and exposing ports,” 2025. [Online]. Available: https://docs.docker.com/get-started/docker-concepts/running-containers/publishing-ports/. [Accessed 21 10 2025].

[17] The Linux man-pages project, “hostname (7) - Linux manual page,” 2025. [Online]. Available: https://man7.org/linux/man-pages/man7/hostname.7.html. [Accessed 23 10 2025].

[18] R2DATO, “D36.4 – Demonstrator implementation phase 3 concluded and test results consolidated (tba, planned for 31/03/2026),” R2DATO project, 2026.

[19] Eurospec, “Specification TCMS Data Service, 1st edition,” June 2019. [Online]. Available: https://eurospec.eu/tcms-data-service/.