Research and Development (Upbring and Optimization) of Programming Languages, Runtimes and Libraries on the EU Processor v1.0

Deliverable Number:	D.4.1
Due Date:	30.06.2024
Date Of Submission:	10.07.2024
Nature:	OTHER
Dissemination Level:	PU
Work Package:	WP4
Lead Beneficiary	UNIMAN

Grant Agreement No 101092850

Funded by
the European Union

Document Control Sheet

Deliverable Title:	Research and Development (Upbring and Optimization) of Programming Languages, Runtimes and Libraries on the EU Processor v1.0
Authors:	Christos Kotselidis, Juan Fumero, Athanasios Stratikopoulos (UNIMAN)
Contributors:	Foivos Zakkak (RHAT-IE), Uwe Dolinsky (CPLAY), Polyvios Pratikakis (FORTH), Iakovos Kolokasis (FORTH)
Reviewers:	Foivos Zakkak (RHAT-IE), Uwe Dolinsky (CPLAY)
Approved By:	Christos Kotselidis (UNIMAN), Dionisios Pnevmatikatos (ICCS)

Document History

Version	Date	Status	Description/Comments
0.1	30.05.2024	Draft	Initial version with Table of Contents
0.2	26.06.2024	Draft	Draft release for internal review
0.3	04.07.2024	Draft	Draft release with addressed comments from internal review
1.0	10.07.2024	Final	Final version submitted to EC

Disclaimer

Table Of Contents

1 Introduction 7
2 TornadoVM 8
2.1 Overview 8
2.2 Hardware & Software Specification of Tested Platforms 8
2.3 How to build the technology? 9
2.4 How to run? 10
3 TeraHeap 11
3.1 Overview 11
3.2 Hardware & Software Specification of Tested Platforms 12
3.3 How to build the technology? 13
3.4 How to run? 14
4 SYCL/oneAPI 15
4.1 Overview 15
4.2 Hardware & Software Specification of Tested Platforms 15
4.3 How to build the technology? 16
4.4 How to run? 17
5 Quarkus 18
5.1 Overview 18
5.2 Hardware & Software Specification of Tested Platforms 18
5.3 How to build the technology? 19
5.4 How to run? 20
6 Summary 21

Executive Summary

This document aims to report the progress of the AERO technologies that are being developed within the scope of WP4 till M18. To that end, we describe and provide guidelines on how to install and run all the WP4 software technologies on the evaluated tested platforms.

List of Abbreviations & Acronyms

Abbreviation/Acronym	Meaning
CPU	Central Processing Unit
DFT	Discrete Fourier Transform
FPGA	Field Programmable Gate Array
GC	Garbage Collection
GPU	Graphics Processing Unit
JIT	Just-In-Time
JMH	Java Microbenchmark Harness
JVM	Java Virtual Machine
OpenCL	Open Computing Language
OCK	oneAPI Construction Kit
OS	Operating System
SERDES	Serialization/Deserialization
SoC	System on Chip

1 Introduction

This deliverable presents the intermediate release of the AERO software technologies that have been developed and optimized under the scope of WP4. This report aims to outline the work that has been carried out till M18 and show how potential users can reproduce the submitted artifacts for the tested hardware platforms.

To that end, this report is structured in four main sections (Sections 2-5). Each section focuses on a specific technology, and aims to:

- Document the progress of the porting and optimization of the technology for a particular tested platform that is similar to SIPEARL Rhea.
- Outline the hardware and software specifications of the tested platform and how the technology is installed.
- Present a guideline for running specific tests and benchmarks on the tested platform.

2 TornadoVM

2.1 Overview

TornadoVM is a parallel programming framework for accelerating a subset of Java programs on hardware accelerators such as Graphics Processing Units (GPUs), Field Programmable Gate Arrays (FPGAs) and Central Processing Units (CPUs). The TornadoVM technology operates as a plugin to an existing Java distribution (e.g., OpenJDK, GraalVM, etc.) and it allows Java programmers to automatically run and accelerate programs that offer data parallelism on modern hardware. Currently, TornadoVM employs three technologies for acceleration, namely OpenCL, CUDA and SPIR-V, which are combined to increase the coverage of the supported target platforms and allow the deployment of applications on multiple and widely diverse compute environments.

TornadoVM is open-source, and it is available in AERO's GitHub repository space[1]. The TornadoVM API is licensed under Apache 2.0.

In the context of AERO, TornadoVM is being ported and tested to new platforms, such as the ARM GraceHopper which has similar processors with the AERO target platform (SIPEARL Rhea). Additionally, TornadoVM is being optimized with numerous new features in order to address the requirements of the AERO use cases (High-Performance Algorithms for Space Exploration - UNIGE, HPC/Cloud Database Acceleration for Scientific Computing - SED). Some optimizations that have already been upstreamed to the TornadoVM code base (current version is 1.0.5) are:

- Compatibility with the Java Vector API.
- Support for multi-threaded execution plans.
- Expansion of the list of supported math operations.
- Definition for copies of on-demand data ranges.

The following sections are organized as follows. Section 2.2 presents the hardware and software specifications of the tested platforms, whereas Section 2.3 describes how TornadoVM can be built. Finally, Section 2.4 outlines how to run the TornadoVM unit-tests and benchmarks.

2.2 Hardware & Software Specification of Tested Platforms

Since the Rhea platform is not ready for testing yet, a similar-alternative hardware platform has been used for porting and testing the TornadoVM software. The platform is hosted in premises of UNIPI and access to two compute-nodes for the ARM GraceHopper compute platform has been provided to UNIMAN.

2.2.1 Hardware Specifications

The hardware characteristics of the tested platform are described in Table 1.

[1] https://github.com/AERO-Project-EU/TornadoVM

Table 1. Hardware specifications of tested platform for TornadoVM.

Hardware Specifications	
System	ARM GraceHopper Architecture
GPU	GH200 with 97GB of global memory
CPU	ARMv64 Neoverse V2 with 72 cores
RAM	574 GB

2.2.2 Software Specifications

The software specification for installing TornadoVM and running its unit-tests and benchmarks on the tested platform are presented in Table 2.

Table 2. Software specifications of tested platform for TornadoVM.

Software Specifications	
OpenJDK	"21.0.3" 2024-04-16 LTS
TornadoVM	version=1.0.5-dev - commit=bb6205b
Cmake	3.25.2
Maven	3.9.3
Python	3.10.12
OpenCL	OpenCL 3.0
CUDA	CUDA 12.3.68
NVIDIA Driver	545.23.08
OS	Ubuntu 22.04.4 LTS, Linux GH200-1 6.2.0-1015-nvidia-64k

2.3 How to build the technology?

```
$ ./bin/tornadovm-installer

usage: tornadovm-installer [-h] [--version] [--jdk JDK] [--backend BACKEND] [--listJDKs]
[--javaHome JAVAHOME]


TornadoVM Installer Tool. It will install all software dependencies except the GPU/FPGA
drivers

optional arguments:

  -h, --help           show this help message and exit

  --version            Print version of TornadoVM

  --jdk JDK             Select one of the supported JDKs. Use --listJDKs option to see all
supported ones.

  --backend BACKEND    Select the backend to install: { opencl, ptx, spirv }

  --listJDKs           List all JDK supported versions

  --javaHome JAVAHOME  Use a JDK from a user directory
```

To install TornadoVM on a Linux ARM server with NVIDIA or Intel GPUs:

```
# Install the OpenCL backend with OpenJDK 21

$ ./bin/tornadovm-installer --jdk jdk21 --backend opencl
```

```
# Install the PTX backend with OpenJDK 21
$ ./bin/tornadovm-installer --jdk jdk21 --backend ptx

# It is also possible to combine different backends:
$ ./bin/tornadovm-installer --jdk jdk21 --backend opencl,ptx

# Source environment variables for the PATH
$ source setvars.sh
```

2.4 How to run?

Users can execute the TornadoVM unit-tests by running the following command:

```
$ tornado-test -V
```

Additionally, users can run the TornadoVM benchmark suite as follows:

```
$ tornado-benchmarks.py
```

To run specific benchmarks, such as the Discrete Fourier Transform (DFT) with the Java Microbenchmark Harness (JMH) toolkit:

```
$ tornado -m tornado.benchmarks/uk.ac.manchester.tornado.benchmarks.dft.JMHDFT
```

3 TeraHeap

3.1 Overview

Typically, big data analytics processing requires iterative computations over data until a convergence condition is satisfied. Each iteration produces new transformations over data, generating a massive volume of objects spanning long computations. Hosting a large volume of objects on the managed heap increases memory pressure, resulting in frequent garbage collection (GC) cycles with low yield. Each GC cycle reclaims little space because (1) the cumulative volume of allocated objects is several times larger than the size of available heap and (2) objects in big data frameworks exhibit long lifetimes. Although production garbage collectors efficiently manage short-lived objects, they do not perform well under high memory pressure introduced by long-lived objects.

The common practice for coping with rapidly growing datasets and high GC cost is to move objects outside the managed heap (off-heap) over a fast storage device (e.g., NVMe SSD). However, frameworks cannot compute directly over off-heap objects, and thus, they (re)allocate these objects on the managed heap to process them. Although some systems support off-heap computation over byte arrays with primitive types, they do not offer support for computation over arbitrary objects, resulting in applications specific solutions, such as Spark SQL.

Moving managed objects off-heap has two main limitations. First, it introduces high serialization/deserialization (SERDES) overhead for applications that use complex data structures. Recent efforts reduce SERDES but demand custom hardware extensions and do not mitigate GC overhead. Second, moving a large volume of off-heap objects to the managed heap for processing increases the GC cost.

TeraHeap is a system that eliminates SERDES and GC overheads for a large portion of the data in managed big data analytics frameworks. TeraHeap extends the Java virtual machine (JVM) to use a second, high-capacity heap (H2) over a fast storage device that coexists alongside the regular heap (H1). It eliminates SERDES by providing direct access to objects in H2 and reduces GC by avoiding costly GC scans over objects in H2. Frameworks use TeraHeap through its hint-based interface without modifications to the applications that run on top of them. TeraHeap addresses three main challenges, as follows.

- **Identifying candidate objects for H2:** Big data frameworks move specific objects outside the managed heap on off-heap storage. For instance, Spark moves off-heap intermediate results; Giraph moves the vertices and edges of the graph and the messages sent between vertices. Frameworks organize such data (partitions) as groups of objects with a single-entry root reference. TeraHeap provides a hint-based interface that uses *key-object opportunism* and enables frameworks to mark objects and indicate when to move them to H2. During GC, TeraHeap starts from root key-objects and dynamically identifies the objects to move to H2.
- **Eliminating GC cost for H2:** TeraHeap presents a unified heap with the aggregate capacity of H1 and H2, where scans over H2 during GC are eliminated, to avoid expensive device I/O. To achieve this, TeraHeap organizes H2 into regions with similar-lifetime objects and deals

differently with liveness analysis and space reclamation. For liveness analysis, TeraHeap identifies live H2 regions by tracking forward (H1 to H2) and cross-region (into H2) references during GC. To identify live objects in H1, TeraHeap explicitly tracks backward references (H2 to H1) and fences GC scans in H2. TeraHeap tracks backward references using a card table optimized for storage-backed heaps, minimizing I/O traffic to the underlying device during GC. For space reclamation, the collector reclaims H1 objects as usual. For H2 regions, unlike existing region-based allocators. TeraHeap resolves the space-performance trade-off for reclaiming space differently. Existing allocators reclaim region space eagerly by moving live objects to another region, which would generate excessive I/O for storage-backed regions. Instead, TeraHeap uses the high capacity of NVMe SSDs to reclaim entire regions lazily, avoiding slow object compaction on the storage device.

- **Applying TeraHeap:** Managed big data analytics frameworks exhibit significant diversity concerning the objects they move off-heap. We investigate how Spark and Giraph, two widely-used frameworks, resolve the trade-off between GC cost due to large heaps and the overhead of off-heap accesses. Spark users explicitly store immutable cached data on the device, while Giraph transparently (without user hints) offloads mutable objects to the device. We modify the two frameworks to use TeraHeap. The use of TeraHeap is different in each framework: Spark uses TeraHeap to store immutable intermediate results, whereas Giraph uses TeraHeap to store mutable objects, such as edges and messages. Within AERO, FORTH will port TeraHeap to the ARMv64 architecture, deploy it on the Rhea platform, and explore its use on the AERO use cases. TeraHeap is public and can be found in the AERO project repositories[2]. TeraHeap is an extension and fork of OpenJDK and is distributed under the GPL.

3.2 Hardware & Software Specification of Tested Platforms

Since the Rhea platform is not available to the AERO project yet, all development and testing is currently performed on an Ampere server hosted by FORTH.

3.2.1 Hardware Specifications

The hardware characteristics of Ampere server are described in Table 3.

Table 3. Hardware specifications of tested platform for TeraHeap.

Hardware Specifications	
Processor	Ampere Altra, 2 socket, 80 cores/socket @3 GHz (ARMv64 Neoverse N1 family)
RAM	256 GB
Disk	1 TB NVMe system, 256 GB NVMe (H2 Heap)

3.2.2 Software Specifications

The software specification for installing TeraHeap and running its unit-tests and benchmarks on the tested platform are presented in Table 4.

[2] https://github.com/AERO-Project-EU/teraheap

Table 4. Software specifications of tested platform for TeraHeap.

Software Specifications	
OpenJDK	OpenJDK, version jdk17u067
OS	CentOS Linux 7 (ARM)
Tools	gcc <= 8.5, python 3, scan-build, compdb, OpenJDK dependencies

3.3 How to build the technology?

Install the prerequisites packages:

```
$ sudo yum install python3-pip
$ pip3 install scan-build --user
$ pip3 install compdb --user
```

Build TeraHeap:

- **Build allocator**

```
$ cd allocator
$ ./build.sh
$ cd -
```

Read the README.md file in allocator directory to export the specific environment variables.

- **Build tera_malloc**

```
$ cd tera_malloc
$ ./build.sh
$ cd -
```

Read the `README.md` file in `tera_malloc` directory to export the specific environment variables.

- **Set your gcc/g++ path/alias**

```
$ cd ./jdk17u067 # for building java17
```

and set CC and CXX variables inside compile.sh to your gcc path/alias.

- **Build JVM (release mode) or Build JVM (fastdebug mode)**

For the release mode:

```
$ ./compile.sh -r
$ cd -
```

For the fastdebug mode:

```
$ ./compile.sh -d
$ cd -
```

3.4 How to run?

To run TeraHeap unit-tests, as well as examples of big data applications and benchmarks of TeraHeap, a user can use AERO's GitHub repository "tera_applications"[3] and follow the instructions listed for the following major frameworks, namely Spark, Giraph, and Neo4j. Furthermore, we are currently working on adding a Lucene deployment.

[3] https://github.com/AERO-Project-EU/tera_applications

4 SYCL/oneAPI

4.1 Overview

SYCL is an open royalty-free Khronos standard enabling portable software acceleration on a wide range of diverse platforms ranging from GPUs, CPUs, FPGAs, SoCs and other hardware. It's a standard C++ API for executing device code written in C++ on these platforms enabling many compiler and runtime optimisations. Under the hood it can also take advantage of existing acceleration APIs/backends such as OpenCL and CUDA. OneAPI is an open, cross-vendor/platform programming model that leverages SYCL and provides a large rapidly growing open-source software ecosystem covering algorithms from AI, Image processing, High Performance Computing and other applications.

AERO supports SYCL and thereby enables oneAPI through the open-source DPC++ toolchain on AArch64 and RISC-V hardware which is at the heart of the EU processor systems. AERO enables SYCL on these platforms in two ways:

1) via an OpenCL driver from the open-source oneAPI Construction Kit (WP3), and
2) via targeting the new DPC++ NativeCPU device to these platforms.

All work related to the NativeCPU device was directly contributed as open-source to the DPC++ repository[4] and the Unified Runtime repository[5]. All work on the OpenCL driver and NativeCPU vectorizer and built-in support was contributed to the oneAPI Construction Kit GitHub repository[6]. Fixes related to AArch64 and RISC-V support have been contributed to the LLVM repository[7]. All these repositories use Apache License v2.0 with LLVM Exceptions.

4.2 Hardware & Software Specification of Tested Platforms

Since the Rhea platform is not available yet and to ensure portability of our OCK and DPC++ work to other AArch64 systems, we have currently tested on alternative AArch64 platforms specified in the following section. We will add Rhea to our testing as soon as it becomes available. We have also started testing OCK and DPC++ on more recent RISC-V hardware, but this is still work in progress and will be reported in the next deliverable.

4.2.1 Hardware Specifications

The hardware characteristics of the tested platforms are described in the following tables.

[4] https://github.com/AERO-Project-EU/llvm
[5] https://github.com/AERO-Project-EU/unified-runtime
[6] https://github.com/AERO-Project-EU/oneapi-construction-kit
[7] https://github.com/llvm/llvm-project

Table 5. Hardware specifications of tested platform for SYCL/oneAPI Construction Kit (Cavium).

Hardware Specifications	
System	Cavium (Vendor)
CPU	ThunderX 88XX, Model 1, 48 cores per cluster, 64-bit
RAM	32GB RAM

Table 6. Hardware specifications of tested platform for SYCL/oneAPI Construction Kit (Graviton perf).

Hardware Specifications	
System	Graviton 3 (performance instance)
CPU	Vendor: ARM, AArch64, 64 Neoverse V1 cores per socket, 64-bit/32-bit, with ARMv8.4-A ISA including 4x128 bit Neon, 2×256 bit SVE, LSE, rng, bf16, int8, crypto
RAM	128GB RAM

Table 7. Hardware specifications of tested platform for SYCL/oneAPI Construction Kit (Graviton dev).

Hardware Specifications	
System	Graviton 3 (developer instance)
CPU	Vendor: ARM, AArch64, 64 Neoverse V1 cores per socket, 64-bit/32-bit, with ARMv8.4-A ISA including 4x128 bit Neon, 2×256 bit SVE, LSE, rng, bf16, int8, crypto
RAM	8GB RAM

4.2.2 Software Specifications

The software specification for installing OCK, DPC++ and running the unit-tests and benchmarks on the tested platform are presented in Table 8.

Table 8. Software specifications of tested platform for SYCL/oneAPI Construction Kit.

Software Specifications	
LLVM	LLVM tip (currently 19.0.0) - Also supported (17.0 and 18.0) for OCK backwards compatibility.
CMake	>= 3.20.0
Python	>= 3.8
GCC	>=7.4
Gnu Binutils	>=2.17
Zlib	>=1.2.3.4
DPC++	>= 19.0.0 (clang version)

4.3 How to build the technology?

Currently, the easiest way to build OCK and DPC++ is directly on the tested platforms (Graviton for AArch64 and Milk-V for RISC-V) using the default build instructions. Since some platforms may have limited or no adequate resources to do this, we are working on cross-compilation for DPC++ and OCK which we expect to be complete soon in the next quarter. Information about DPC++ with NativeCPU support can be found in the NativeCPU documentation[8].

The following steps can be used to build DPC++ with OpenCL and NativeCPU support:

[8] https://github.com/intel/llvm/blob/sycl/sycl/doc/design/SYCLNativeCPU.md

```
$ git clone https://github.com/AERO-Project-EU/llvm.git -b aero_m18
$ export DPCPP_HOME=$PWD
$ python $DPCPP_HOME/llvm/buildbot/configure.py --native_cpu
$ export CPLUS_INCLUDE_PATH=/usr/include/c++/13
$ cmake --build $DPCPP_HOME/llvm/build -- deploy-sycl-toolchain -j 8
$ export PATH=$PWD/llvm/build/bin:$PATH
```

Note that the above “git clone” command uses the “aero_m18” branch which may quickly be outdated. To build the most recent native_cpu implementation, ensure to update the sycl branch with the latest upstream updates and replace “-b aero_m18” with “-b sycl” in above “git clone”.

4.4 How to run?

The SYCL tests from DPC++ are run by using the following command:

```
$ cmake --build $DPCPP_HOME/llvm/build –target check-sycl -j 8
```

These tests include tests for NativeCPU. To build and run SYCL applications or tests separately, use:

```
$ clang++ -fsycl -fsycl-targets=spir64,native_cpu -o test test.cpp
```

This command builds the SYCL app/test simultaneously for OpenCL and NativeCPU. The ONEAPI_DEVICE_SELECTOR environment variable can be used then to select the SYCL target device, as follows:

To run on NativeCPU:

```
$ ONEAPI_DEVICE_SELECTOR=native_cpu:cpu ./test
```

To run on OpenCL (CPU):

```
$ ONEAPI_DEVICE_SELECTOR=opencl:cpu ./test
```

5 Quarkus

5.1 Overview

Quarkus is a Kubernetes-native framework for developing microservices using the Java programming language. Among its key features are:

- The ease of development through dev-mode, a very fast and interactive way to develop your applications without the need to manually recompile and reload the project. Quarkus also employs hot-reloading which allows it to update the running code without having to restart the whole application making changes appear instantly.
- The ability to perform a large part of initialization at build time and significantly reduce the startup time and the runtime dependencies of the project.
- The ability to native compile the applications which results in smaller self-contained containers with better startup performance.
- The integration with hundreds of very popular libraries through the Quarkiverse ecosystem where people can contribute their extensions enabling even more integrations or functionality.

In the context of AERO, we focus on improving the stability of Quarkus on AArch64 architectures like the SIPEARL Rhea. We work on supporting and improving both the JVM mode (i.e. running on OpenJDK's HotSpot Virtual Machine) as well as the native mode (i.e. compiling the project to a native executable using the GraalVM project). We (RHAT) maintain our own distribution of GraalVM, named Mandrel, which enables us to focus solely on the parts of GraalVM we are interested in (i.e. native compilation) and build it based on the well-established and mature OpenJDK instead of the Oracle Labs JDK that upstream GraalVM is based on. In AERO we focus on productizing Quarkus (along with Mandrel) for architectures like SIPEARL Rhea. This includes building and testing both projects (as well as their dependencies) through RHAT-CZ's internal systems and providing them as supported products (on AArch64 based architectures) to our customers.

In the context of AERO, we maintain the following clones of the corresponding upstream repositories:

- https://github.com/AERO-Project-EU/quarkus
- https://github.com/AERO-Project-EU/mandrel

All of the Quarkus related work (including the projects it depends on) is done upstream in the open through the community.

5.2 Hardware & Software Specification of Tested Platforms

5.2.1 Hardware Specifications

For the needs of AERO, RHAT-CZ purchased two machines based on the ARM Neoverse N1 architecture which is compatible with that of SIPEARL Rhea. Both machines feature the same hardware specifications as listed in Table 9.

Table 9. Hardware specifications of tested platform for Red Hat Mandrel/Quarkus.

Hardware Specifications	
Processor	2x Ampere Altra Max 128 cores (ARM Neoverse N1 family)
RAM	32x 32GiB DDR4 3200 MHz
Disk	4x 960GB NVMe

5.2.2 Software Specifications

The software specification for installing Red Hat Mandrel and Quarkus and running the unit-tests on the tested platform are presented in Table 10.

Table 10. Software specifications of tested platform for Red Hat Mandrel/Quarkus.

Software Specifications	
Host OS	Red Hat Enterprise Linux release 9.3 (Plow)
Guest OS	Red Hat Enterprise Linux release 8.10 (Ootpa)
Podman	4.9.4
Quarkus	3.8
Mandrel	23.1
OpenJDK	21

5.3 How to build the technology?

5.3.1 Quarkus

To build Quarkus one needs to first clone the repository and checkout the 3.8 branch

```
$ git clone https://github.com/AERO-Project-EU/quarkus.git –branch 3.8
```

To build Quarkus, one will need an OpenJDK 21 installation. We suggest using the Temurin distribution. Please head to https://adoptium.net/temurin/releases/ and choose the Operating System and architecture of your machine to download a compatible archive with the JDK. Once downloaded, extract it and set JAVA_HOME to point to it, e.g. on unix:

```
$ export JAVA_HOME=/opt/jvms/jdk21
```

Then enter the project directory and build it with:

```
$ cd quarkus
$ ./mvnw -Dquickly
```

5.3.2 Mandrel

By default, Quarkus will use prebuilt Mandrel images that it automatically pulls and runs using a container runtime engine. But for development purposes or in cases where a container runtime engine is not available users can set GRAALVM_HOME to point to a local installation.

To build Mandrel from source we need JAVA_HOME set to point to a JDK 21 installation like in the Quarkus build process above. We will also need a tool called mx and some build scripts available in the repository mandrel-packaging.

To get mx use the following command:

```
$ git clone https://github.com/graalvm/mx.git -branch 6.46.1
```

To get mandrel-packaging use:

```
$ git clone https://github.com/graalvm/mandrel-packaging/ -branch 23.1
```

Finally get mandrel using:

```
$ git clone https://github.com/AERO-Project-EU/mandrel/ -branch mandrel/23.1
```

Then enter the mandrel-packaging project directory and build mandrel using:

```
$JAVA_HOME/bin/java -ea build.java \
  --mx-home ../mx \
  --mandrel-repo ../mandrel
```

5.4 How to run?

The Quarkus repository contains a number of integration tests that can be used to test Quarkus and Mandrel.

To run the tests on JVM mode one can choose the tests they are interested in and run:

```
$ ./mvnw verify -f integration-tests/pom.xml -pl test1,test2,test3
```

To run the tests on native mode one can choose the tests they are interested in and run:

```
$ ./mvnw verify -Dnative -f integration-tests/pom.xml -pl test1,test2,test3
```

6 Summary

To sum up, this deliverable has focused on reporting the status of the software technologies that are in the scope of WP4. For each technology, we provided a list of the software and hardware specifications of the tested platforms that have been used for the development and testing of our technologies till M18. Finally, we pointed to the code repositories where the open-source software technologies are hosted, and outlined the steps to build and test the technologies.