USE OF MICRO-SERVICES ARCHITECTURE AND CONTAINERIZATION FOR THE FAST DEVELOPMENT AND TESTING OF THE CUBESAT NANOSATELLITES SOFTWARE

O. Liubimov
National Aerospace University «Kharkiv Aviation Institute «KhAI», 17, Chkalova Str., Kharkiv, 61000, Ukraine

Abstract. CubeSats as a sub-class of nanosatellites have become a game-changer in the industry of scientific research and exploration of the new space technologies. Their cost effectiveness, relative ease of manufacturing, and predicted lifecycle are the main factors of success, leading to roughly 2280 nanosatellites being put on the Earth’s LEO. However, many issues still arise in regard to the effective development, testing, and successful mission control of the CubeSats. The authors of the article suggest the combination of well-known software development processes, design paradigms, patterns, and techniques, that are creating a new way of making CubeSat software development as flexible and as easy as possible. The very first part of the research suggests the outlook on modern software development processes and their evolution over the last years. The second part of the research looks into virtualization and containerization principles as the architectural response to complex software development. The third and fourth parts of the research concentrated on the selection and proper testing of the container execution engine and its performance with the most common algorithm used in embedded software development. Via the porting and proper performance test of the WASM3 WebAssembly interpreter, the authors provide valuable research in regard to the practical use of micro-services, and containerization in the CubeSats. For the sake of the usefulness and completeness of the porting and performance testing – the authors suggest the most-known STEM platforms as the test environments. Concluding the research by the practical porting and testing steps, and found limitations of the containerization, the authors come up with a newly defined concept of so-called Software Defined Satellites. Such a concept could help the industry to minimize risks in reusing the software, perform several missions on the same CubeSat spacecraft, and thus, drastically decrease the cost of CubeSats launches.

Keywords: CUBESAT, COTS, MOTS, CONTAINERIZATION, OPEN-SOURCE, MICRO-SERVICES ARCHITECTURE, SOFTWARE DEFINED SATELLITE, WASM3, SOFTWARE, WEB ASSEMBLY.
Introduction

CubeSat, a type of nanosatellites, started its commercial and research way in 1999 when the CubeSat Design Specification was started in California [1]. From the beginning, CubeSats were invented as technology for research and education. Since 1999 this form factor and the ideology of nanosatellites have gained high popularity in both commercial and military industries, as well as in academia, which gave an opportunity to student and science teams to gain access to research of space and the Earth from the Low Earth Orbit (hereinafter LEO).

A typical CubeSat is a nanosatellite ranging in size from 1U – 10×10×10 cm up to 12U when several 1U units can be assembled together (stacked up or placed next to each other). Usually, CubeSats are delivered to orbit as "parasitic load", i.e. – secondary load. This is what makes its delivery to LEO very cheap, in comparison to the dedicated satellite launch. A typical program for such parasitic launches of CubeSats is NASA’s CubeSat Launch Initiative (CLI). A typical method for the local “from the rocket” launch of satellites to orbit is a mechanical armed-spring-based ejection, using special load "dispensers", for example, P-POD.

The key reasons for the high popularity of the CubeSats standard and approach, namely: the low cost of such satellites, the relatively short time of their construction and testing (which made it possible to use master’s degree students during their studies), and standardization, which made possible the reuse of both individual parts of satellites, as well as ground stations for receiving telemetry and controlling satellites at LEO.

The typical tasks of CubeSats include three types: remote sensing of the Earth or other space objects, communication infrastructure (especially for CubeSats constellations, examples of OneWeb and StarLink), and research of problems and tasks of re-entry into the Earth’s atmosphere.

Looking at statistics of launches and orbital deliveries for the past few years [2, 3], we can see a rapid growth in popularity and further development of this area of space technology (Fig. 1):

![Figure 1 – Statistics of CubeSats launches from 2000 up to 2023](image)

You can see that almost 500 CubeSat satellites were launched in 2021 and 2022. Unfortunately, even with 20 years of launch experience, a crucial number of launches remain unsuccessful. Looking at Figure 1, we can see that approximately 18.5% of satellites in 2022 were completely lost, and only 2.9% of satellites fully completed their task (full launch mission).

A large number of both scientific and commercial teams around the globe are trying to reduce the likelihood of failed or partly failed satellite and increase the percentage of satellites that fully complete their programmed mission.

Knowing the modern complexity of the CubeSat eco-system it is clear that the major part of the complexity and time spent is laying in the area of software definition, design, development, and testing (verification and validation, commissioning and proper support after the launch too). With the big portion of the software reuse and general use of open-source

Ключові слова: ВІДКРИТИ АПАРАТНІ ПЛАТФОРМИ, ВІДКРИТИ ПРОГРАМНІ ПЛАТФОРМИ, COTS, MOTS, КУБСАТ, МІКРО-СЕРВІСИ, КОНТЕЙНЕРІЗАЦІЯ, WASM3, ПРОГРАМНО-ВИЗНАЧЕНІ СУПУТНИКИ.
software, it is still the major headache of the development team to get it properly used.

**The main targets of this paper are** the following problems and/or the search for their solutions:

- Finding a reliable and predictable on-board software development process and techniques;
- Suggesting the use of the micro-services architecture for the development of the on-board software;
- Trying out the development of the on-board software with the containerization approach and finding its benefits;
- Practical results of porting WASM container machine to the RP2040 and ESP32 platforms and performance measurements;
- Further definition of Software Defined Satellite (SDS) ideology as the new class of the satellites.

**Research Problem Statement**

The main purpose of this work is the attempt to bring modern processes, architecture, and mindset to the software developers of the CubeSats. The essence of the attempt is to shift the importance of the total resource constraint designs to the modern concept of distributed micro-services-based development of software. Reinforcing this approach with the use of containerization for flight software is the key novelty that is addressed in the article.

The practical side of the work is related to the comparison of the performance of so-called “bare-metal” and container-based software modules, where several typical embedded software development algorithms were used as the performance reference model.

As the final word and emphasis in this work, the new thinking paradigm of Software Defined Satellite (SDS) is brought up by the research team.

**Industry state-of-the-art**

The usage of COTS components opens up a wide variety of both open-source and proprietary software to be used [4, 5]. The use of the COTS software creates a huge number of troubles in the way of developing and proper testing of the CubeSat on-board software. The majority of the research are concentrating on a very thorough Verification & Validation (V&V) approach [6], others are proposing Software In the Loop (SIL) or/and Hardware In the Loop (HIL) simulation [7]. Major efforts are also produced in the field of Failure Emulation Mechanisms (FEM) [8], as well as the introduction of different fault injections platforms (FIP) [9].

When we look into what actually makes CubeSats development a complex and challenging task, we can see a few major factors. The main cyclomatic complexity of making a CubeSat comes with the software development where development teams, especially the ones that consist mainly of undergraduate or/and postgraduate students, are facing the challenges of the low quality, low maturity of the source code, and lack of time to properly overview the entire code being developed. One of the main reasons for this is the widely-used approach called the “waterfall” development lifecycle. It is mainly driven by the fact of hardware and its integration greatly fitting into the paradigm of making an asset (hardware), leading the entire project to be within that development life-cycle (Fig. 2).
developers of the framework being decided to be used and knowing the time and skills constraints in the team – never challenged and changed.

If we take the more mature development teams (mainly those who are part of commercial companies), they are using the so-called V-model that comes from the functional safety world where every step of the SDLC shall be properly verified after being accomplished (Fig. 3):

![Diagram of a V-model verification approach](image)

**Figure 3** – Waterfall SDLC that is reinforced by the V-model verification approach

Using such an approach gives a closer look into the verification and later on validation of the software but doesn’t change either the design thinking or design approach of building the CubeSat software.

However, nowadays, the majority of the so-called “big IT” software development teams and specialists are eager to use iterative and/or incremental software development models, i.e., based on either SRUM, Agile, or Kanban (Fig. 4).

![Diagram of an Agile SCRUM-like SDLC](image)

**Figure 4** – Typical agile SCRUM-like SDLC

What such an approach means is that the whole CubeSat software is broken down into chunks of “features” (functions) that are ready to be implemented, tested, and demonstrated as a separate stand-alone function.

Understanding such a concept of the modern software development process and strengthening it with newly introduced embedded software principles such as microservices architecture and further – containerization is a good candidate for providing a counter-solution to the monolithic and waterfall-created software (Fig. 5).

![Diagram of Microservices vs Monolithic architecture](image)

**Figure 5** – Example of the micro-services vs monolithic architecture

What is the micro-services architecture? Well – it is a development concept when the entire software to be built is broken down into a number of small, independent, and loosely coupled services that communicate to each other in a HTTP, WebSockets, AMQP or even MQTT way. Obviously, connecting such an approach to the idea of SCRUM-like decomposed and developed at the same time, smaller fractions/modules of the CubeSat software make perfect sense. Practically this will allow to: develop different software modules simultaneously, isolate quality issue and errors in the particular module of the software, reuse someone’s else modules in much more simple manner, use bigger development team to shorten time for the development.

Looking into the industry practices, what are the key drivers of the micro-services architecture success? They are:

- Using containers to package and run microservices in an isolated and portable way.
- Using orchestration platforms such as Kubernetes or Cloud Run to manage the deployment, scaling, and networking of microservices.
- Using service meshes such as Istio or Linkerd to provide service discovery, load

131
balancing, routing, observability, and security for microservices.

- Using domain-driven design (DDD) to identify and design microservices based on bounded contexts and business domains.

- Using event-driven architecture (EDA) to enable asynchronous communication and integration between microservices using events and message brokers.

- Using continuous integration and continuous delivery (CI/CD) practices to automate the testing and deployment of microservices.

According to Google Cloud analytics, containerization is what really drives and pushes the use of the micro-services approach into the redesign of classical monolithic software to the services-based one.

Yet an important and major effect of another aspect of the micro-services architecture use, CI/CD development, is brought up by the so-called DevOps development approach. The area of DevOps emerged and revolutionized the processes used by developers to create and manage software and how the software is deployed to a staging or production environment. One of the key concepts of the DevOps approach is the ability to deliver the working software to the end client at any given point in time. This fits perfectly well with the idea of being able to do CubeSat stand-alone functions development, testing, and V&V process. Another major effect of the popularity of the micro-services design approach is the advancement in software virtualization and containerization, and the emergence of tools such as Docker [10] and Kubernetes [11] that became the industry standard.

What is containerization in a broad sense? According to IBM [12], one of the frontiers of developing the concept and bringing it to the industry, containerization is the packaging of software code with just the operating system (OS) libraries and dependencies required to run the code to create a single lightweight executable – called a container – that runs consistently on any infrastructure. More portable and resource-efficient than virtual machines (VMs), containers have become the de facto compute units of modern cloud-native applications.

Containers are often referred to as “lightweight,” meaning they share the machine’s operating system kernel and do not require the overhead of associating an operating system within each application. Containers are inherently smaller in capacity than a VM and require less start-up time, allowing far more containers to run on the same compute capacity as a single VM. This drives higher server efficiencies and, in turn, reduces server and licensing costs (Fig. 6).

![Diagram of containerization principles](image)

Figure 6 – Containerization principles

Perhaps the most essential thing is containerization allows applications to be “written once and run anywhere”. This portability speeds development, prevents cloud vendor lock-in, and offers other notable benefits such as fault isolation, ease of management, simplified security, and more.

As the importance of embedded applications is rising, hardware capacity is increasing - the development of the microservice architecture on embedded systems is booming. There are many examples of the home-baked frameworks that are more and more changing developer's mindsets into micro-services-based architectures [13].

During the analysis of the IEEE Xplore, Scope, and Google Scholar papers on the matter of embedded software containerization the following container engines were found:

- MicroPython;
- Jerry Script;
- Singh;
- Femto Containers;
- Bacelli;

132
• Velox VM;
• Toit;
• Golioth;
• Wasm3.

Making a top-level analysis and review it became clear that more or less each container engine is based on the WebAssembly principles of real-time code translation.

WebAssembly was developed by the W3C organization initially as the translator for the web-technologies and applications. Later on, because of its popularity, it became a good choice for the cross-platform engine for non-web applications too. Partially its popularity is driven by the support of the native to the typically embedded software developer programming languages such as C, C++, and Rust. As of today, there are more than 35 implementations of high-performance WebAssembly machines, where roughly 50% of those are actively supported and continue their lifecycle.

Having analyzed the modern software development lifecycles, mindset and culture of the developers, and available tools used by the containerization – the authors would like to propose a blend of the all stated above as a new and game-changing method of CubeSat software development and verification.

**Solution of the problem**

So, what are the proposed new methods of developing CubeSat software with the intent of using SCRUM-like development life-cycle and containerization. They are:

- Each container represents one either functional block of the CubeSat (ADCS, EPS, etc.) or/and the separate either OBC or Payload function;
- During development and V&V process of each separate container – the separate team or the team-member develops each separate container;
- Due to the fact of the containers being cross-platform – verification and development of each separate container is done at the PC and not at the CubeSat hardware.

As the basis of the solution of the container’s engine implementation on the typical CubeSat hardware, WASM3 interpreter engine was chosen. WASM3 was initially created to deliver an outstanding performance for the low performance targets and thus fully supports the energy efficient and low-performance hardware of CubeSats.

To be able to run WASM3 on the embedded target of any type, the following minimal infrastructure is to be ported:

- Filesystem – required to store containers and be able to upload those for the execution;
- CLI (Command Line Interface) – that allows to manipulate with the containers (Run, Stop, Load, etc.) and also allow the user to see the system parameters in the real-time;
- HTTP server that will allow a simple and easy to implement OTA (Over-The-Air) transfer of the container images from a PC to the embedded target. (For the communication with the ground stations – AX.25 over HTTP can be used).

As the typical RTOS for the CubeSats is the FreeRTOS by Amazon, the overall architecture of the solution looks like the following (Fig. 7).

![Figure 7 – CubeSat specific WASM3 engine porting architecture](image)

During the porting the following assumptions were done:

1. Each container is running as a separate FreeRTOS task;
2. Containers are running with the priority and scheduling by Round Robin principles as for the FreeRTOS;
3. File System is used for the containers storage and each container is uploaded to RAM before it’s use;
4. The overall porting is carried out by the use of MCU-specific API and FreeRTOS-specific API.
For the file-system implementation the SPIFF filesystem was selected as it already had a port for the ESP32.

All container images are permanently stored on the file-system and thus are ready for the operation right after the system boot.

Schematically the task upload sequence can be shown as the following (Fig. 8):

![Figure 8 – WASM3 container upload sequence diagram](image)

The sequence diagram above assumes that only one container is running at one single moment of time.

The decision about running an uploaded container is designed in an asynchronous manner – so it doesn’t require the upload request to be carried out.

If the container already exists on the file system the only “START” command is required to be altered (Fig. 9).

![Figure 9 – WASM3 container control sequence diagram (start of the container)](image)

In case of a CubeSat implementation – such a command can arrive via any other telemetry communication channel than the containers uploading interface.

For the performance testing the most popular in the open STEM-like hardware platforms were chosen, namely RP2040 by Raspberry (Fig. 10) and ESP32-WROOM32 (Fig. 11) by Espressif.

![Figure 10 – Raspberry Pi W and it’s processor – RP2040 (133Mhz)](image)

![Figure 11 – ESP32 development kit and it’s module – ESP32-WROOM32 (80-240Mhz)](image)

At the same time – typical algorithms for the embedded software were used for the benchmarking, namely: Fast Fourier Transform (FFT), Bubble Sort, and CRC-16 Checksum calculation algorithm.

The main reason for measuring performance here is to see how big overhead by the two layers of abstraction (WASM3 and its middleware) is done. Do determine it – same functional algorithms were run on a bare metal implementation, i.e., what vendors offer as low-level API and code written on C/C++ and WASM3 compiled code.

For the determination of the performance measurements deviation and dispersion – 1000 cycles of each algorithm were run (detailed results are available upon request).

The following results were obtained:

- CRC-16 algorithm (100/1000/1000 elements), (Fig. 12).

![Figure 12 – CRC16 calculation comparative results](image)
Fast Fourier Transform (128/256/512 samples), Fig. 13:

![FFT Comparison](image1)

**Figure 13 – FFT comparative results**

Bubble sort of 32-bit signed integers (100/500/1000 elements), (Fig. 14):

![Bubble Sort Comparison](image2)

**Figure 14 – Bubble Sort (32-bit signed integers) comparative results**

We can conclude that, regarding the execution of a single task, WASM3 implementation done for the purpose of this research, had respectable results. It can be seen as the middle ground between solutions like Arduino or ESP-IDF, which are very efficient, low-level and close to native performance, however not offering many functionalities and restricting the user, and solutions like Toit and MicroPython, that despite supporting many operations that ease the development of applications, lack in performance.

**Conclusions**

The proposed move from waterfall-based SDLC to the more agile and flexible SCRUM-like SDLC is definitely a very reasonable step forward. The combination of it with the microservices and containerization-based implementations of the CubeSat onboard software, provides a big novelty and ease of implementation to the developers and testers of CubeSats.

Comparing the performance of the WASM-based containers and pure “bare metal” implementations of the proposed algorithms has shown a 10x difference in performance. However, knowing the nature of the calculations and importance of the real-time data processing for the typical space exploration CubeSat satellites – the performance is acceptable and feasible. Surely, some found limitations of the current state of the WASM container engine and its porting by the research team, are to be addressed. Further development of the inter-container data exchange is yet to be designed and properly developed, as well as the concurrent container’s execution under the hood of the same processor.

Using the containerization approach and the ability to load the satellite mission after its launch and dispensing from the P-POD or similar dispenser, opens up a new design thinking paradigm of so-called Software Defined Satellites (hereinafter SDS). The paradigm allows multiple uses of the same CubeSat being launched only once until it gets through the decaying orbit. In this way, the CAPEX part of the launch and the overall cost of the launch get to an even smaller amount and potentially can leverage down the overall mission monetary costs to the bare minimum, i.e., a range of 5000 USD or so.

Surely, the biggest limitation of the multi-purpose CubeSat will be its size and the payload’s unique mission. However, having a big amount of sensors on board (several IMUs, temperature, accelerometers, etc.) – the use of those is definitely the mission by itself. Prolonging the lifecycle of the CubeSat via the ability to alternate the mission software via containers, drastically expands the lifetime of the onboard electronics and software that, in its turn, allows us to find the errors and anomalies that are important for the further maturity of the platform.

**Next steps and further research directions**

Step 1 – Further development of the ported WASM3 engine so it can be further optimized and support the concurrent container’s execution. This work shall allow simple yet powerful orchestration, for instance on the basis of Event Driven Architecture (EDA) or/and use of the Saga design pattern.
Step 2 – Porting of the implementation of the WASM3 engine to the Microchip SAM V71[14] processor as it is a decided platform for the author’s CubeSat OBC initiative.

Step 3 – Researching and implementation of the PC-based development and testing environment that could help developers to faster engage with the platform.

Step 4 – Research and practical implementation of the low-power modes for both FreeRTOS, the new platform, and WASM3 ported software – to be able to reach the lowest possible energy consumption.

Acknowledgments

The article was compiled with the support of Mr. Ihor Turkin as the scientific advisor - Professor, Software engineering department lead, dept. 603, National Aerospace University “KhAI”. As well as Mrs. Iryna Shulga as the language advisor.

A separate acknowledgement is to LLC “EKTEOS-UKRAINE” who supported with the prototypes manufacturing of the suggested OBC PCB.

References


Received by Editorial Board on 09.06.2023.
Information about the author

Liubimov Oleksandr.
Ukraine. National Aerospace University “KhAI”, dept. 603, PhD student. Head of the electronics development department at the Danish-Ukrainian design company EKTOS-Ukraine LLC. Sphere of interests – spacecraft software, embedded software, complex software-hardware solutions, embedded systems, functional safety, electronics. ORCID: 0000-0003-3636-6939.