Agile Mission Operations in the CubeSat Project MOVE-II

Conference Paper · May 2018

DOI: 10.2514/6.2018-2635

6 authors, including:

Alexander Lill
Technische Universität München
5 PUBLICATIONS **4** CITATIONS
SEE PROFILE

Thomas Zwickl
Technische Universität München
6 PUBLICATIONS **4** CITATIONS
SEE PROFILE

Constantin Costescu
Technische Universität München
3 PUBLICATIONS **0** CITATIONS
SEE PROFILE

Lucie Patzwahl
Technische Universität München
2 PUBLICATIONS **0** CITATIONS
SEE PROFILE

Agile Mission Operations in the CubeSat Project MOVE-II

A. Lill*, T. Zwickl, C. Costescu, L. Patzwahl, C. Soare, M. Langer
Chair of Astronautics, Technical University of Munich, Boltzmannstrasse 15, 85748 Garching, Germany

With the continuous advancements in commercial off-the-shelf small satellite technology, there has been a significant increase in proposed missions and an ongoing trend towards rapid development and launch. Traditionally, mission operations of larger spacecraft utilize established products, adapting the operational interfaces and software to the needs of the specific mission. This paper reports the creation of the mission operations interface for a CubeSat mission from scratch to a fully operational system in about seven months under the voluntary commitment of about six students. We will first give a short introduction into the traditional software development used extensively in the space sector before we briefly describe CubeSats and our own CubeSat project, called MOVE-II. The used software development process will be described using the agile methodology Scrum as a baseline. It will be explained how this process was implemented utilizing tools like Trello, GitLab and Slack. Afterward the software design, as well as used frameworks and software, are briefly described to demonstrate how these go hand in hand with our development process. Subsequently, our operations software, as well as satellite operations with our software, are described with a focus on advantages and disadvantages of our software development approach and general lessons learned that we gathered during our project.

I. Introduction

Traditional software for space applications focuses on achieving robustness and reliability and therefore follows classical and predictive development approaches such as the V-model or the waterfall model. These models aim at building the system sequentially by determining the requirements of the mission as a first step, then deriving the software architecture and finally breaking it down into modules that are designed in more detail [1]. This process can be formalized by modeling the system with the help of a System Modeling Language [2], which allows defining the subsystems and interfaces of the system in more detail before the start of the development phase. After the implementation of the software according to the requirements and specified architecture, all software is validated through unit and system testing. Final user acceptance testing verifies that all user requirements are met. Only after this phase is concluded real-time operations begin. Most satellite systems go through this mostly linear process, which demands special attention to the formulation of the requirements as they become more difficult to change later on in the project. Although these methodologies may imply longer implementation times and demand higher budgets when compared to more modern, adaptive development strategies, the space industry has been reluctant to adopt newer approaches. Furthermore, institutions in the space software industry may decide to use legacy software frameworks [3] which are hard to adapt to newer models (such as the spiral model or agile development).

CubeSats [4] are standardized satellites that come in sizes specified in so-called units (10 x 10 x 10 cm) and are usually developed for educational purposes or for technology demonstration. The low cost of CubeSat projects enables students and companies to experiment with different development processes and technologies while obtaining fast hands-on experience with spacecraft technologies without the complexity and cost of traditional space programs. Although it may vary for each project, the development of a CubeSat follows a linear pattern, as workforce and budget are usually limited. Due to these limitations the focus is usually placed on the development of hardware and on-board software, pushing the development of operations software to the very end, potentially causing a rushed implementation. This issue can be tackled by agile development techniques which allow to start the creation of the operations software early and to grow alongside the CubeSat, as it is dynamically adapted to new requirements.

*alexander.lill@tum.de, phone: +49 89 289 16017

The Munich Orbital Verification Experiment II (MOVE-II) project is the second satellite of the CubeSat program MOVE of the Technical University of Munich. This one-unit (1U) CubeSat is being developed since April 2015 [5] and will be launched into a 575 km sun-synchronous orbit in late 2018. The project is supervised by two Ph.D. students and over the course of three years, more than 150 students from five different faculties were involved in developing and building the MOVE-II CubeSat and its associated systems. During the development, most of the hardware and software were designed and implemented by students through voluntary work or theses and study projects at the associated Chair of Astronautics.

The MOVE-II working environment is quite different compared to the environments found in the commercial space industry. All members of the project are students who pursue their bachelor's or master's degree and are therefore not working full-time for the project. Their participation in the project is either connected to a thesis, part of their study curriculum as Interdisciplinary Project (IDP), or completely voluntary. This leads to less weekly working hours compared to full-time developers and makes daily meetings impossible. This is compensated by the high motivation and dedication of the students for the MOVE-II project. Furthermore most of the students participating in the MOVE-II project start with only minor practical experience in their respective fields or have never worked in such a big team. Students join the project either because they are interested in spaceflight and satellite technology or because they want to acquire practical experience in a fascinating and extraordinary field.

As can be seen in Figure 1 the MOVE-II project is composed of a space segment and a ground segment. The space segment is represented by the Satellite System and contains several different subsystems that make up the hardware and software of the satellite itself. The Ground Segment contains the Ground Station System and the Operations System. The Ground Station System consists of the hardware required for receiving and transmitting signals from and to the satellite, and all necessary hardware and software to decode and encode these signals. The Operations System includes the hardware and software necessary for analyzing the satellite's data and controlling it. Its development and use will be the focus of this paper.

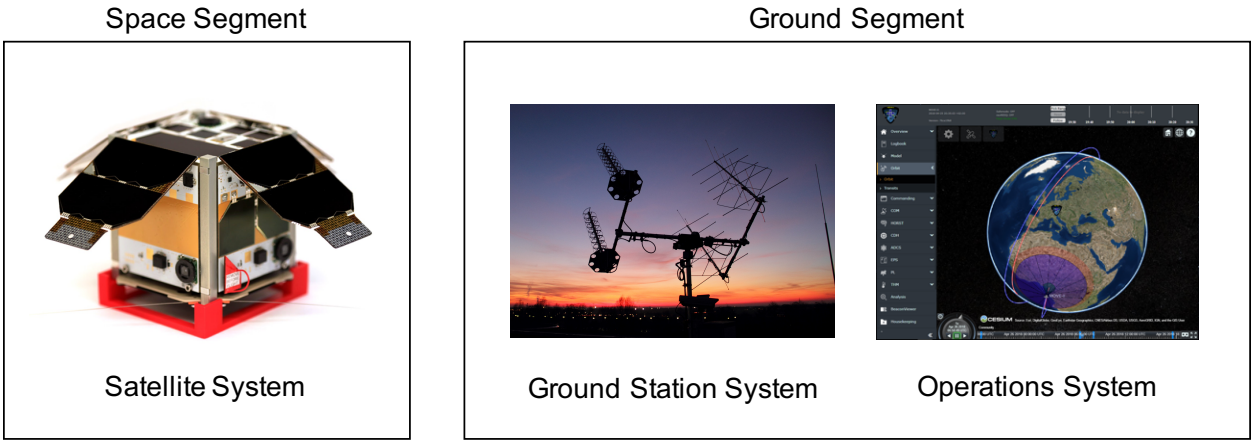

Fig. 1 System of Systems.

Statistics [6, 7] show that many CubeSats fail due to insufficient testing of the satellite in its flight configuration. Further, the lessons learned of First-MOVE [8] contain strong recommendations to start developing and testing the software as early as possible. As observed by other teams using agile methodologies helps testing software already during development and improves its maturity [9]. This is why agile development methodologies were used in MOVE-II [10].

II. Operations Software Development Process

This section introduces the used software development process and describes how this process was implemented.

A. Process Specification

The software development process used for the MOVE-II Operations System is based on agile software development methodologies. It is inspired by Scrum and was adjusted to our needs. This was necessary due to the reasons mentioned in section I. The specification of our software development process is described in more detail in an interdisciplinary project report [11] and briefly summarized in the subsequent paragraphs of this paper. The Scrum development process consists of artifacts created and used in the process, events that define the process, and roles that participants of the process have [12]. The following paragraphs will briefly describe the most important parts of Scrum and provide a summary of the adjustments we applied.

The Scrum artifacts **Product Backlog** and **Sprint Backlog**, which contain the total list of User Stories—things that one should be able to do with the product—and the list of User Stories that should be implemented in the current iteration respectively, were kept unchanged. This is also the case for the definition of the **Product Increment**. It describes the extended and improved version of the product after any given iteration.

The following events are defined in the Scrum process: The **Sprint**, describing the time frame of one iteration of the process, normally has a constant length of two to four weeks. In our environment, the duration was shortened to one week. Experience has shown that for longer *Sprints*, work was only done in the few days before the *Sprint* ends and that the amount of finished work can be increased by focusing more on short-term goals that can be reached during shorter sessions of a few hours, for example in the evenings or the weekends.

The **Sprint Planning** event usually happens between two *Sprints* and is the event where the whole team decides which User Stories will be worked on in the next *Sprint*. The effort for the different User Stories is estimated to get a better idea of how many User Stories should be included in the next *Sprint*. In our environment, the estimate for every task was given by the team leader and confirmed or discussed by the rest of the team, and then adjusted accordingly if necessary. Experience has shown that this approach saves time compared to other estimation techniques, for example **Planning Poker** [13].

Daily Scrum is the daily, very short meeting of the whole team where the finished tasks, ongoing tasks, and current impediments are discussed. Due to our environment (the voluntary nature of the project, this project not being a full-time endeavor, and the different schedules of the students) these daily meetings were completely dropped. Experience has shown that the voluntary team members rarely work on the project every day, but rather in specific time frames like for example weekends or on some free evenings every week. This was resolved by making the on-demand team communication as convenient as possible and will be explained in the following subsection II.B.

The **Sprint Review**, which is done at the end of every *Sprint* to inspect the latest *Product Increment* and update the *Product Backlog*, was kept unchanged. The same was done with the **Sprint Retrospective**, which is occasionally done to allow the team to reflect on the past *Sprint*, analyze their work and deduce what went well and where there might be potential for improvement. These two events are important for demonstrating the progress of the team and ensure continuous improvement.

The following roles are defined in the Scrum process: The **Product Owner** is responsible for the *Product Backlog*. User Stories need to be clearly defined and ordered by priority according to the *Product Owner*'s vision of the product. The *Product Owner* is accountable for the delivered product. The **Scrum Master**'s responsibility is to make sure that the Scrum process is followed and that all rules, practices, and values of the process are understood. The *Scrum Master* is a servant-leader to the *Product Owner* and the team of *developers* and also tries to ensure the team's progress by removing impediments. Due to the limited number of available persons in our environment the same person occupied the roles of *Product Owner* and *Scrum Master*. The **Developers** implement the User Stories contained in the *Sprint Backlog*. The *developers* are encouraged to organize and manage their own work. While the limitations of our environment (see section I) apply, this role was kept unchanged.

B. Process Implementation

Given the more formal specification of the software development process used for the MOVE-II Operations System in the last paragraphs, this subsection will describe how this process was implemented. First, our team communication tool and our project management tool are presented. Afterward, the implementation of our process is explained step by step.

The MOVE-II project makes great use of the team communication tool Slack [14]. It allows interactive communication with the rest of the MOVE-II team as well as within the team in real-time. It was mainly used to coordinate with other team members about subsystem related tasks and allowed the operators of the system to give us feedback about subsystem pages, bugs, and problems encountered while working with the interface as well as feature requests regarding missing functionality. As we did not know every requirement of each subsystem interface, Slack enabled us to continuously receive feedback after releasing and deploying a new *Product Increment*. This allowed us to refine the application towards the requirements of the operators with every iteration.

To facilitate the project management the web-based project management application Trello [15] was used. Trello allows to create lists with an arbitrary number of cards. In our environment, each list represents the state of a card. Each card resembles a User Story and contains all the information required, such as a description, links to external resources and which *developers* are working on it. As can be seen in Figure 2 our User Stories can be in ten different states/lists, which will be explained subsequently.

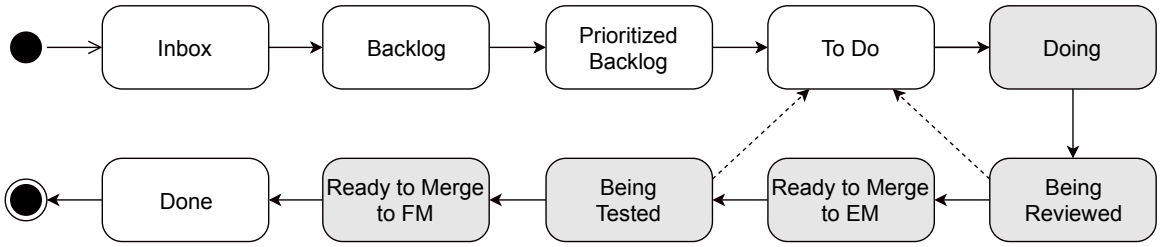

Fig. 2 User Story States.

The "Inbox" is the list where all new cards, which may be new features, observed bugs or possible improvements, are kept until they are sorted into one of the following lists. Our **Product Backlog** resides in the list "Backlog". From time to time, the most important tasks from this list are moved to the "Prioritized Backlog". This separation of backlogs was done to tackle the high number of items in the Backlog and to make the prioritization of tasks easier.

In our weekly team meeting the **Sprint Review** and **Sprint Planning** are conducted. All finished User Stories in the "Done" list are archived and therefore hidden from the board. Afterward, tasks that are started, but not yet finished and therefore still in any of the lists between "To Do" and "Done" (see gray boxes), are discussed to determine their status and their possible impediments. Then our "To Do" list is filled with tasks from the "Prioritized Backlog". After this weekly meeting our **Sprint** starts and the *developers* take cards from the "To Do" list and move it into the "Doing" list as soon as they start working on them.

As soon as a task is finished, the card is moved into the "Being Reviewed" list and assigned to another team member for review. We use the version control system git [16] in combination with the web-based git repository manager GitLab [17] and a Feature Branch Workflow [18]. In our project the `master` and `develop` branches represent the software version for our production and testing system respectively. Merge Requests (also known as *Pull Requests*) conveniently present all added, deleted and modified lines of code and are used to review the changes that are requested to be merged from a specific feature branch into the `develop` branch, the branch containing the most current version that is being tested. Reviewers can comment on every single line and point out problems, give feedback, and ask questions. The created Merge Requests are only accepted once all discussions are resolved by answering all questions and either implementing the feedback or agreeing that it is not necessary to do so.

Once the Merge Request for a User Story has been reviewed and approved, the card for this User Story is moved forward to the list "Ready to Merge to EM". If the review shows that changes to the code are necessary, the card is re-assigned to the initial developer and moved backward to the "To Do" list. The "Ready to Merge to EM" list represents changes that are ready to be deployed in our test environment and are just waiting for someone to oversee the deployment and check if the new version works, or for the test environment to be available. This is necessary as sometimes other members of the project are using our test environment for their own tests, as it is connected to the Engineering Model (EM) of our satellite and is therefore often used for operator trainings and to test new software for the on-board computer of the satellite.

To automatize the deployment process, GitLab offers a free to use Continuous Integration (CI) and Continuous Deployment (CD) pipeline called GitLab CI/CD which is used in the project to deploy the most up-to-date version of our system on the respective servers. It also allows to roll-back to a previous version whenever some problem with the new version is encountered to ensure the availability of the system for the ongoing testing of the satellite system. To facilitate the deployment process with the automatic build pipelines we used Docker [19] to containerize all our microservices (see subsection III.A). Docker provides the functionality to deploy applications without having to install and configure all their dependencies on the host operating system. In the project, it is used to run the microservices in so-called containers, which include the minimal runtime environment of the application, and thus are independent of the host system. These containers can run isolated without the need of virtual machines and have the advantage that the software will behave the same in every environment where it is deployed. Another advantage of Docker is that it offers reproducible builds because it always starts with a clean environment to avoid conflicts with other dependencies or leftovers from a previous installation or build. After the changes were successfully deployed in our test environment the corresponding User Story is moved to the "Being Tested" list. It stays here until all changes have been successfully tested and the team agrees that this User Story is ready to be deployed in our production environment. Every team member has the capability to make this decision, but every other team member can also raise his or her concerns about such a decision. This might be the case, if an issue concerning the implemented changes was found. In this case the card goes back to the "To Do" list.

User Stories that are in the "Ready to Merge to FM" list are normally deployed to the production environment once a week after the team meeting (see section about *Sprint Review* and *Sprint Planning* above). Our production environment is connected to the Flight Model (FM) of our satellite and should be working stable and reliable at all times. This will be of even more importance once our satellite has been launched. The successfully implemented User Stories end up in the "Done" column and will be archived at the beginning of the next team meeting (see section about *Sprint Review* above).

III. Operations Software Design

This section briefly describes the design of our Operations Software. First, the principle of **microservices** and their use in our architecture are explained. A **message broker** is used for the exchange of information between the microservices, while the connecting clients use a **REST API** and **WebSockets** for communication. Then open-source **frameworks** and **other software** used to build our system are described, such as databases, load balancing tools, and log management tools.

A. Microservices

In the project, we followed the approach of implementing a Microservice Architecture (MSA) which is a composition of several small and independent services that together provide the functionality of a complex software application [20]. Each of these microservices focuses on exactly one certain functionality. They are loosely coupled through lightweight protocols and provide high cohesion within themselves. By using technologies like a *message broker* (see subsection III.B) and a *REST API* (see subsection III.C) for communication each microservice can be written in a different programming language. Given this modular approach, it is quite easy to split the workload between multiple team members without too much overlap during implementation. A MSA, therefore, makes it easier to develop, understand and deploy the services by using, for example, an automatic build pipeline (see subsection II.B), that can be triggered as soon as there is a new version.

Figure 3 shows the architecture of our Operations System. The light gray box in the center contains all our microservices, represented by red boxes. These microservices provide the means to store and retrieve data from the database (above the light gray box) and query this data using the provided *REST API* and the *WebSockets* for real-time data. Every subsystem has its own microservice and its own database schema, that the microservices connect to using the Transmission Control Protocol (TCP). In the following paragraphs the two most important microservices, Commanding (CMD) and Housekeeping (HK), will be explained in more detail.

On the left side of Figure 3 you can see all services that handle communication from and to the satellite by connecting to the Ground Station System. The Beacon Parser (BPA) is responsible for parsing the raw beacon received from the satellite as an array of bytes once a minute via a TCP connection to the Ground Station System. It is then interpreted and sent in an intermediary representation to the Beacon Poster (BPO) using *rabbitMQ*. The BPO transforms the received data into a JavaScript Object Notation (JSON) format and sends this to the different microservices using the *REST API*. The different "RESQ" services implement our application protocol for commanding the satellite and talk to their respective counterparts running on the satellite. These "RESQ" services provide functionality for sending commands to the satellite, receiving their results, and up- or downloading files. The "RESQ" services talk to the Ground Station System via TCP sockets on the one side, and via *rabbitMQ* to the CMD microservice on the other side. The CMD microservice is responsible for handling all commands and file transfers, which are received via the *REST API* and then forwarded via *rabbitMQ* to the "RESQ" services for execution.

The Housekeeping Processor (HPR) shown at the bottom of Figure 3 is responsible for parsing housekeeping files that are downloaded from the satellite and posting the parsed information to the *REST API*. The HK microservice is responsible for handling all housekeeping import requests, which are received via the *REST API* and then forwarded for execution via *rabbitMQ*. The Authentication (AUTH) server at the top of Figure 3 is responsible for user authentication and access control for the Operations System. It provides a user login and defines user permissions to restrict access to sensitive data. To implement the AUTH server the Open Authorization 2 (OAuth2) framework was used.

On the right side of Figure 3 you can see all the clients that use the provided *REST API* to interact with the microservices. The Notification Service (NOTS) receives beacons from the BPA via *rabbitMQ* and evaluates them for non-nominal values using meta-data from the *REST API*. In case there are non-nominal values the operators are notified, for example with a message on Slack. The Statistics Service (STATS) queries the different microservices once a day to provide the operators with statistics about the usage of the Operations System, like the number of executed commands, user logins or transferred files. The Frontend represents the Operations System Interface that the developers, as well as the operators, use to interact with the satellite system. It uses the *REST API* as well as *WebSockets* to retrieve the latest data from the microservices.

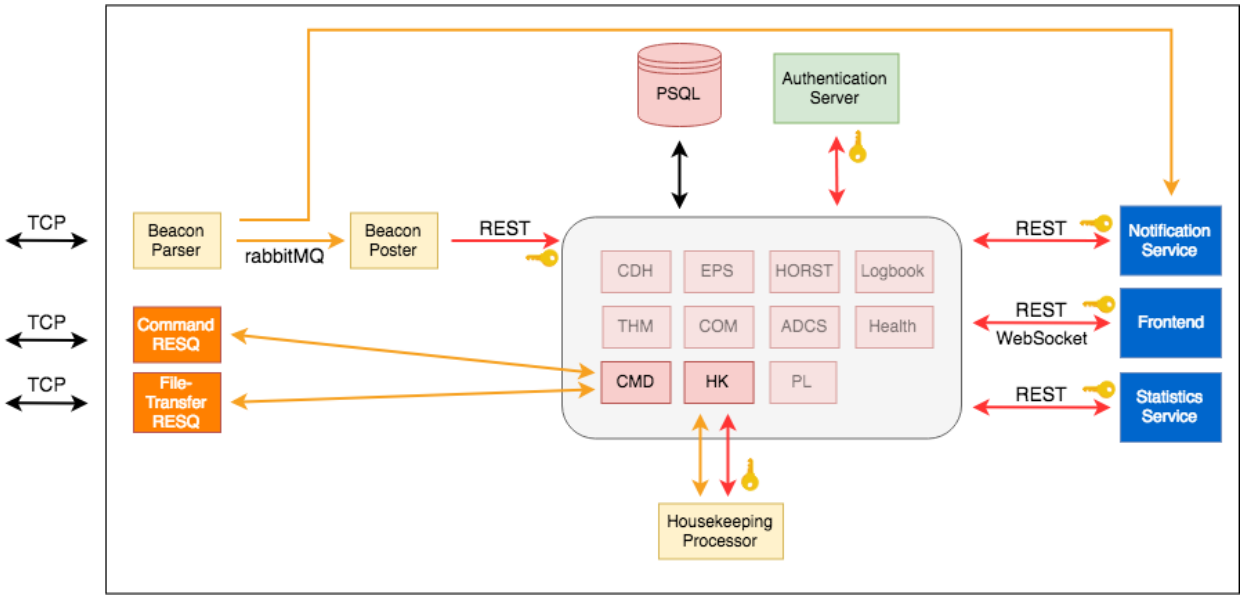

Fig. 3 Operations (OPS) System Architecture.

B. Message Broker

RabbitMQ [21] is an open-source message broker where applications can send and receive messages through defined queues. Using RabbitMQ as middle-ware between the different microservices allows to decouple them and define a platform and language independent interface. RabbitMQ also provides means for load-balancing and work distribution by its implementation of different queuing and producer-consumer messaging protocols. Here several applications, called the producers, send messages to the respective queue. Other applications, called the consumers, subscribe to a queue. When new messages arrive at the queue they are forwarded to the subscribed applications for further processing.

C. REST API

An Application Programming Interface (API) defines an interface that is used for exchanging data between different software components. REpresentational State Transfer (REST) is an architectural style for developing stateless services and is building upon the very widely used Hypertext Transfer Protocol (HTTP) standard. The REST architecture provides interoperability between services over the internet and usually defines the following CRUD* operations to access and manipulate textual representations of web resources: The GET request is used to retrieve an existing resource from a collection. The POST request is used to create a new resource and add it to a collection. The PUT request is used to update an existing resource. The DELETE request is used to delete an existing resource.

D. WebSockets

One drawback of using the REST communication protocol is that there is no direct connection from the server to the client, as the client only requests data from the server or sends data there. This means that the web application running in the browser is not notified about new data from the microservices. In order to guarantee that the visualized data is up-to-date, the client needs to poll the microservices every few seconds. This has proven to be very inefficient as it generates unnecessary requests when no new data is available and creates some unwanted delay which inhibits the operation of the satellite in real time. Thus, we decided to extend our microservices by WebSockets to enable the user's browser and the server to open an interactive communication session. This enables the client to send messages to the server and receive event-driven responses without having to poll the server for a reply. WebSocket is a protocol that operates as an upgraded HTTP connection, exchanging variable-length frames between the two parties, instead of a stream. For implementing the WebSockets we used the STOMP protocol which is commonly used inside a WebSocket when a web app needs to support bidirectional communication with a web server. It defines an interoperable wire format so that any of the available STOMP clients can communicate with any STOMP message broker. This provides easy and widespread messaging interoperability among languages and platforms.

E. Frameworks

This section describes the frameworks Angular 2 [22] and Spring [23] that were used within the project to simplify building and deploying our applications.

Angular 2 is an open-source web application framework that was used to develop the user interface for the operators that operate the satellite. Angular 2 provides some essential advantages compared to the traditional development of web-pages with libraries such as jQuery. For once, Angular 2 supports templating, that allows to reuse components on different pages without writing the same code over and over again. Second, it allows to write very modular code, so that several developers can work in parallel on the application, without creating too many conflicts when the same parts of the software are changed by two people at the same time.

The Spring Framework is an application framework for the Java Platform and was used in the project to implement all the microservices responsible for retrieving and persisting the data from the satellite. With Spring Boot one can create a stand-alone Spring application that is easy to deploy on basically any environment that supports Java. The Spring Framework comes with a lot of other functionality, for example, annotations that allow to develop a REST interface with minimal effort.

F. Other Software

During the project, a lot of other software was used to complement the functionality of our own applications. In the following, some of the most important used software is briefly introduced.

*Create, Read, Update, and Delete

PostgreSQL [24] is an open-source, general purpose and object-oriented database management system that was used to persist all the data exchanged with the satellite or generated by our Ground Station System software.

HAProxy [25] is an open source, high-performance TCP/Hypertext Transfer Protocol Secure (HTTPS) load balancer that acts as a single point of contact for all clients connecting to our microservices. This provides a simple interface to our clients and hides the complexity of different microservices reacting to different requests. With the HAProxy the requests can be forwarded to the respective microservice by mapping the Uniform Resource Locator (URL) to the matching microservice. This provides an abstraction to the clients as they do not need to know on which port and server the respective microservice is running. They just contact the HAProxy which takes care that the request will arrive at the correct microservice.

As the entire system only consists of many different microservices that can also be run distributed over different machines, it is a challenge to collect all the logs that may be involved in identifying and fixing a certain problem. To fix this issue, Graylog 2 [26] in combination with Logspout [27] was used to send and accumulate all logs from all microservices in chronological order in one central place. With those tools, it is now possible to search through all logs and pinpoint the cause of the problem.

IV. Results

This section describes the results of the software development process defined in the previous sections and summarizes its key advantages. Lessons learned from using this process can be found in section V.

The developed Operations System Software consists of a browser application that is used by the operators for analyzing the satellite's data and controlling it. This browser application connects to our servers using a REST API and WebSockets (see subsection III.C and subsection III.D). This allows to show historic telemetry as well as live telemetry in real time and access satellite status information from anywhere in the world. Figure 4 shows the page containing the commanding interface of the MOVE-II satellite, while Figure 5 shows an example of the graphs used for analyzing telemetry.

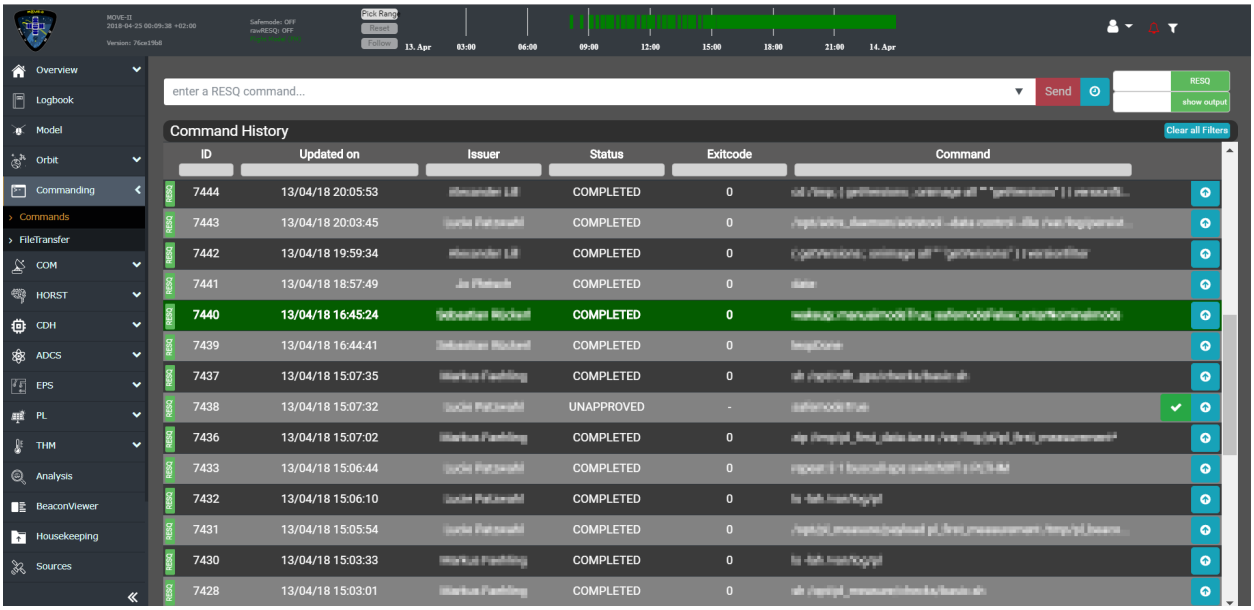

Fig. 4 Commanding Page in the OPS Interface.

The implemented web application can be accessed from anywhere and brings some major advantages for mission operations. In our environment (see section I) the working hours of the team members depend on their schedules as well as their commitment. Being able to access the satellite status and command the satellite this conveniently simplifies development as well as later operations of MOVE-II, enabling testing whenever desired and avoiding the necessity to come to university on weekends or late at night.

Using our agile software development process a very first version of the Operations System was available for the first thermal vacuum chamber tests in May 2017. This minimum viable product allowed to see the temperatures of

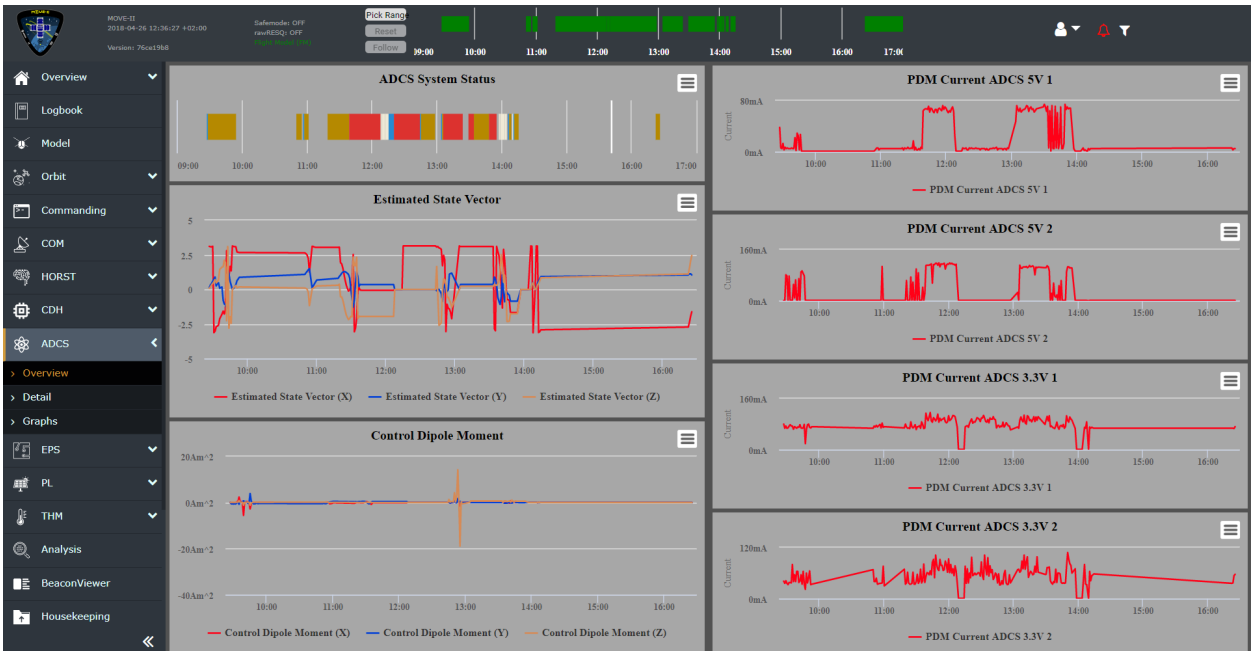

Fig. 5 Attitude Determination and Control System (ADCS) Page in the OPS Interface.

different satellite parts in convenient graphs, showing both historic data of the last hours as well as live data. For this the data was already transmitted via radio signals, processed by the Ground Station System and forwarded to our Operations System the same way it will happen during real operations. This allowed us to develop as well as debug the satellite using the complete chain of communication following a "test as you fly, and fly as you test" [28] approach. This also had a positive effect on the maturity of our Ground Station System.

Using our Operations System in parallel to the development of the satellite also allowed us to receive feedback from the satellite developers and operators right from the beginning. Using the system as if the satellite was already in orbit, for example by simulating the length of typical overpasses, additionally generated a constant flow of valuable feedback, either in person or via our team communication tool Slack (see subsection II.B). The feedback concerned the user interface, the behavior of the system and possible issues during the future operations of MOVE-II. This allowed us to detect and resolve many bugs during the development of our systems. It helped us to tailor our software to the needs of the future operators and thus avoid over-engineering and implementing superfluous functionality.

Due to the availability of the Operations System during the development of MOVE-II it was additionally possible to start the training of satellite operators even though the development of neither the Satellite System nor the Operations System was concluded. This enabled us to recruit future operators while our systems were still being developed and increased the possible time for trainings. This furthermore allowed us to transfer knowledge directly from the developers to the operators and therefore improve the operators' understanding of the systems.

V. Lessons Learned

During the development and use of the MOVE-II Operations System, a variety of observations was made and challenges had to be faced. In this section, our lessons learned in the areas of the development process and the developed system, communication, project management, and general software development will be briefly summarized.

A. Development Process

The MOVE-II project followed the approach of building an Engineering Model (EM) before starting to build the Flight Model (FM). This approach resulted in having one and shortly later two satellites that could be used for developing and testing the Ground Station and Operations System. As described in subsection II.B this was used for creating two different environments: The EM environment, used for testing new configurations and satellite software, and the FM environment, sometimes used for operator trainings and tests that have to be conducted with the flight hardware. At the

same time, the EM environment is used by the developers of the Operations System to test new features, improvements, and bug fixes. This sometimes resulted in an EM environment that was not stable or did not work as expected, which in turn caused delays for the developers of the satellite software. A possible solution for future projects would be having another environment only for the Operations System developers to test their new software without influencing other team members.

Developing the Operations System in parallel to training the operators comes with constant changes in the functionality and appearance of the interface, a challenge that the operators have to deal with. New features are added or improvements and bug fixes are implemented. Sometimes the introduction of new bugs could not be avoided. In our experience, this especially affects training material for new operators that quickly becomes outdated, whereas trained operators normally catch up quickly or just ask their team for help.

B. Communication

Communication is one of the key success factors of projects with a lot of team members. Our experience shows that clearly defined communication channels are crucial for a good flow of information. In MOVE-II every subsystem had its own Slack channel for communication between all other subsystems and this particular subsystem, as well as a channel for team-internal communication to allow efficient exchange of information and to avoid non-transparent communication in direct messages, which are only visible to the sender and one receiver. This supports open and clear communication between the team members in different subsystems and their own subsystem.

The developers of the Operations System used the team-internal channel to quickly and transparently discuss problems and implementation details, while the public channel was used to get feedback from the operators using our software and to improve the understanding of their needs and requirements. Troubleshooting and creating bug reports were supported by this via the direct and responsive communication between the users and the developers, and automated workflows that allow the creation of bug reports directly from these messages, including the full conversation protocols [11].

Making the exchange of information as convenient as possible allowed us to be aware of the needs of our operators as well as the developers of the satellite and to incorporate this knowledge into our software.

C. Project Management

It is important to have a good overview in projects with multiple team members and a high number of work packages and tasks. Project management software is crucial to be aware of the status of each task, which tasks are being worked on and who is working on which task. For this purpose different project management software should be evaluated to guarantee that it fits the used development process. Our experience shows that it is especially important to have exactly one place for all tasks instead of using a variety of tools and places. This avoids confusion and ensures a good overview of all relevant tasks and visualization of progress or lack thereof.

During the project, the regular use of retrospectives ensures reflection of the used processes, tools and problems and allows to deduce improvements. Our experience shows that these improvements and lessons learned are important for the success of the current and future projects. While writing these lessons learned it was further realized, that continuously collecting lessons learned throughout the project simplifies writing a summary of lessons learned.

D. Software Development

The long-term success of any software project can be achieved by quality code supported by documentation. Maintaining a good code quality is hard, especially in a student team, but can be achieved with code reviews. In our experience, this can be implemented by having at least two developers who are responsible for each component or microservice. This way, there is always another team member that knows how a component or microservice works and can provide feedback and a valuable sanity check for every change. Furthermore, this approach ensures that the project's progress is not impacted by the fluctuation of team members. In our project peer code reviews were conducted for all changes using Merge Requests (see subsection II.B), which were on occasion additionally discussed in team meetings in form of group code reviews.

Another aspect of quality code is that there are automatic tests, which enable better reusability and maintainability. This is crucial for the detection of changes that introduce new bugs. This aspect is most often skipped in favor of faster development, as writing the tests can take more time than implementing the feature or fix itself. There are many reasons

why a team can end up not writing tests. From our experience, this might happen in exam periods, where team members have less time for the project and tasks are postponed until the exam period is over. In this case, seemingly less important tasks—for example writing tests—are neglected bringing instant short-term benefits, for instance less time spent or more visible progress. The negative consequences of this only become apparent later in the project, when the component becomes more complex or when new developers join the project and have no possibility to safely modify existing code.

The capabilities of the tools used for the development and testing of the Satellite System provide important insights into the necessary capabilities of the future Operations Software. In our experience, the software used for satellite operations should provide the same functionality as the software used during development and testing of the satellite. This avoids doing work twice, as the same code can be used for development, debugging and operations, and allows faster debugging of non-nominal behavior in orbit.

Given the used microservice architecture (see subsection III.A) the number of components that needed to be deployed every week was quite high. In our experience automation for all repeatedly executed steps always pays off. Therefore GitLab CI/CD was implemented (see subsection II.B) and used extensively in every part of our project, for example for automatically compiling the latest version of our LaTeX documentation and providing a Portable Document Format (PDF) document for download. This allowed us to improve team productivity and reduce human errors that often happen during routine tasks.

Due to the nature of student projects, it can never be predicted how many team members can actively collaborate, or how long they will participate in the project (see section I). Students may have to focus on other areas of their studies and reduce or even cease their efforts for the project. This can quickly become a critical issue if there is no documentation about their work or no other developer with knowledge about the internals of the affected components. In our experience, it is very important to never stop recruiting for the project, even if systems and components are finished. While documentation provides an overview of the systems and steps to troubleshoot and solve common issues, experts with knowledge of the system often reduce troubleshooting time and are able to provide more efficient help. This is especially helpful for satellite operations, where response times might be critical for the mission's success.

VI. Conclusion & Outlook

This paper presented the agile development process used by about six voluntary students and applied to the Operations System of the MOVE-II CubeSat project. This approach allowed to create software based on the users' changing requirements and resulted in a minimum viable product in time for our thermal vacuum tests only five months after development started. Continuous feedback from the satellite developers and operators was directly integrated into the development process, and the full chain of communication was tested as soon as possible. Since then this first version was extended and improved iteratively following the described process. Currently, the MOVE-II CubeSat and its related systems are being finalized, while the future operators are being trained. The launch is scheduled for October 2018.

It is planned to open-source the created software and further experiment with modern technologies and development processes. Many students from the MOVE-II project are currently transitioning to other projects, for example "MOVE-ON". The "MOVE-ON" project aims to develop and start a high-altitude balloon every half year. This serves as a platform for short-lived and cost-efficient experiments to verify and test spacecraft technologies and development processes. Furthermore, it is planned to investigate the use of Test-Driven Development (TDD) to implement our lessons learned and further improve our processes.

VII. Acknowledgments

The authors acknowledge the funding of MOVE-II by the Federal Ministry of Economics and Energy (BMWi), following a decision of the German Bundestag, via the German Aerospace Center (DLR) with funding grant number 50 RM 1509.

References

[1] Ley, W., Wittmann, K., and Hallmann, W., *Handbook of Space Technology*, Vol. 22, John Wiley & Sons, 2009.

[2] Spangelo, S. C., Kaslow, D., Delp, C., Cole, B., Anderson, L., Fosse, E., Gilbert, B. S., Hartman, L., Kahn, T., and Cutler, J.,

"Applying Model Based Systems Engineering (MBSE) to a standard CubeSat," *2012 IEEE Aerospace Conference*, IEEE, 2012, pp. 1–20. doi:10.1109/AERO.2012.6187339.

[3] Evans, D. J., "OPS-SAT: Preparing for the Operations of ESA's First NanoSat," *14th International Conference on Space Operations*, 2016, pp. 2490–2499. doi:10.2514/6.2016-2490.

[4] Heidt, H., Puig-Suari, J., Moore, A., Nakasuka, S., and Twiggs, R., "CubeSat: A new Generation of Picosatellite for Education and Industry Low-Cost Space Experimentation," *14th Annual/USU Conference on Small Satellites*, 2000.

[5] Langer, M., Schummer, F., Appel, N., Gruebler, T., Janzer, K., Kiesbye, J., Krempel, L., Lill, A., Messmann, D., Rueckerl, S., and Weisgerber, M., "MOVE-II - The Munich Orbital Verification Experiment II," *Proceedings of the 4th IAA Conference on University Satellite Missions & CubeSat Workshop, IAA-AAS-CU-17-06-05*, Rome, Italy, 2017.

[6] Langer, M., and Bouwmeester, J., "Reliability of CubeSats – Statistical Data, Developers' Beliefs and the Way Forward," *Proceedings of the 30th Annual AIAA/USU Conference on Small Satellites,Paper SSC16-X-2*, Logan, UT, 2016.

[7] Swartwout, M., "The First One Hundred CubeSats: A Statistical Look," *Journal of Small Satellites*, Vol. 2, 2013, pp. 213–233.

[8] Langer, M., Olthoff, C., Harder, J., Fuchs, C., Dziura, M., Hoehn, A., and Walter, U., "Results and lessons learned from the CubeSat mission First-MOVE," *Small Satellite Missions for Earth Observation, 10th International Symposium*, IAA, Berlin, 2015.

[9] Wortman, K., Duncan, B., and Melin, E., "Agile methodology for spacecraft ground software development: A cultural shift," *2017 IEEE Aerospace Conference*, IEEE, 2017, pp. 1–8. doi:10.1109/AERO.2017.7943886.

[10] Lill, A., Messmann, D., and Langer, M., "Agile Software Development for Space Applications," *Deutscher Luft- und Raumfahrtkongress 2017*, Deutsche Gesellschaft für Luft- und Raumfahrt - Lilienthal-Oberth e.V., Munich, Germany, 2017.

[11] Lill, A., "Organization and Development of the Mission Operations System for the MOVE-II CubeSat," Interdisciplinary Project, Technical University of Munich, RT-IDP 2016/05, 2018. doi:10.13140/RG.2.2.35355.36646.

[12] Schwaber, K., and Beedle, M., *Agile Software Development with Scrum*, 1st ed., Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[13] Javdani Gandomani, T., Koh, T. W., and Binhamid, A., "A Case Study Research on Software Cost Estimation Using Experts'. Estimates, Wideband Delphi, and Planning Poker Technique," *International Journal of Software Engineering and its Applications*, Vol. 8, 2014, pp. 173–182. doi:10.14257/ijseia.2014.8.11.16.

[14] Slack Technologies, Inc., "Slack," , 2018. URL `https://slack.com`, v3.1.1, visited on 2018-04-20.

[15] Atlassian Corporation Plc, "Trello," , 2018. URL `https://trello.com/`, v2.10.3, visited on 2018-04-20.

[16] Linus Torvalds, "Git," , 2018. URL `https://git-scm.com/`, v2.16.3, visited on 2018-04-20.

[17] GitLab Inc., "GitLab Community Edition," , 2018. URL `https://gitlab.com/`, v10.5.6, visited on 2018-04-20.

[18] Atlassian Corporation Plc, "Tutorial: Git Feature Branch Workflow," , 2018. URL `https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow`, visited on 2018-04-20.

[19] Docker, Inc., "Docker," , 2018. URL `https://www.docker.com/`, v17.06, visited on 2018-04-20.

[20] Bakshi, K., "Microservices-based software architecture and approaches," *2017 IEEE Aerospace Conference*, 2017, pp. 1–8. doi:10.1109/AERO.2017.7943959.

[21] Pivotal Software, Inc., "RabbitMQ," , 2017. URL `http://www.rabbitmq.com/`, v3.6.11, visited on 2018-04-20.

[22] Google LLC, "Angular," , 2018. URL `https://angular.io/`, v5.2.0, visited on 2018-04-20.

[23] Pivotal Software, Inc., "Spring Framework," , 2017. URL `https://spring.io/`, v4.3.6, visited on 2018-04-20.

[24] PostgreSQL Global Development Group, "PostgreSQL," , 2017. URL `https://www.postgresql.org/`, v9.6.4, visited on 2018-04-20.

[25] Willy Tarreau, "HAProxy," , 2018. URL `https://www.haproxy.org/`, v1.7.11, visited on 2018-04-20.

[26] Graylog, Inc., "Graylog," , 2018. URL `https://www.graylog.org/`, v2.4.3, visited on 2018-04-20.

[27] Glider Labs, "Logspout," , 2018. URL `https://github.com/gliderlabs/logspout`, v3.3, visited on 2018-04-20.

[28] Leveson, N. G., "Role of Software in Spacecraft Accidents," *Journal of Spacecraft and Rockets*, Vol. 41, No. 4, 2004, pp. 564–575. doi:10.2514/1.11950.