COMPARATIVE STUDY INTO ARCHAIC SOFTWARE INTEGRATION BASED ON NATO S4 SOFTWARE LIBRARY

Liisa Sakerman, Madis-Mikk Remmet, Rait Rotšan, Kaarel Allemann, Kristi Reispass

Abstract. This study aims to research different integration strategies for a legacy software suite, focusing on comparing Command Line Interface (CLI) and web API methods. The research evaluates these approaches using both objective metrics, such as CPU/RAM usage and response times, and subjective metrics, including client satisfaction. Results showed that CLI integration handles a higher volume of low-computation-time requests more efficiently, while RAM usage was similar across all requests and integrations. Response times were marginally longer for the web API integration, but the differences became less significant for more complex requests. Feedback from the client indicated a lack of strong preference, however the importance of efficient CPU and RAM usage was emphasised due to the crucial need for battery efficiency. Although the integrations in this study exhibited minimal differences, these findings will provide valuable insight for future integration projects and suggest possible avenues for further research in this domain[1].

Keywords: Legacy System, Software Integration, Command-Line Interface (CLI), Web Application Programming Interface (API)

Võtmesõnad: pärandkoodi süsteem, tarkvara integreerimine, käsurea liides, veebi rakendusliides

1. Introduction

Software development as an engineering sub-discipline has evolved over nearly a century, giving rise to a vast array of programming languages and software systems. Among these is the S4 program, a NATO software development project aimed at creating a shared software library implementing NATO standards for indirect fire. Despite being developed using a programming language and tools that are less commonly used today (as discussed further

[1] The study is supported by the research grant of the Estonian Ministry of Defence.

in section 2), the S4 program holds significant value particularly for nations seeking to rapidly standardise their artillery capabilities. This research project aims to address the avenues and challenges of integration by identifying and comparing the most effective integration methods. Objective criteria will be established to evaluate various integration methodologies, with a focus on facilitating seamless integration with different platforms. The selected methodologies will then be used to develop a software library tailored to meet the specific requirements outlined in collaboration with this research project's client-partner who intends to actively implement it in the future. The project will conclude with an assessment of the integration methods based on client feedback, resulting in the creation of two functionally identical yet differently integrated software libraries. The first section gives an introduction; the second section introduces the technical possibilities for interlanguage integration; the third section presents the methodology for measuring integration effectiveness; the fourth section outlines design sketches for the integration architecture; the fifth section describes the process from architecture to implementation; then a comparison of the integrations is made, limitations are considered and, finally, conclusions are made.

2. Technical Possibilities for Interlanguage Integrations

The SG2 Shareable (Fire Control) Software Suite (S4) is a project written in the Ada programming language, aimed at developing standardised and shareable fire control and ballistic computation software. It started with the NABK – NATO Armaments Ballistic Kernel and has since grown in functionality through other software products that increase output accuracy by considering meteorological, terrain elevation and other applicable data in system computations.[2]

Ada is not as widely used today compared to other modern programming languages. As a compiled language, Ada requires a compiler for execution, but free and open-source compiler options are limited compared to languages with broader support. For example, in Visual Studio Code, the AdaCore

[2] **Fonnor, J**. 2018. Development of firing tables for accuracy stretches back more than 100 years. https://www.army.mil/article/215516/development_of_firing_tables_for_accuracy_stretches_back_more_than_100_years (8 June 2024).

extension has about 53,000 downloads[3], whereas a comparable C/C++ extension has nearly 73 million downloads[4]. This difference highlights the slower development of tools for Ada, likely due to its narrower usage and developer base.

Although Ada remains significant in critical systems, particularly in defence, it is often seen as a legacy language because of its limited adoption in new projects and reduced community engagement. As of this writing, there are 1,900 Ada tag watchers on Stack Overflow[5] and 8,800 members in the Ada subreddit[6], compared to 1.1 million watchers[7] and 297,000 members[8] for C++. Regardless, the functionality that the S4 suite provides is valuable and remains the fastest way to implement NATO digital ballistics standards and gain co-operation capability with other allies. This is also the case currently in Ukraine that needs to standardise its indirect fire capabilities under difficult conditions. This research is the first step in the process of moving on from the Ada programming language, aiming to prove the sustainability of the S4 software suit.

2.1. Integration Techniques

This study aims to research the most effective ways to integrate S4 software into modern systems. There are multiple ways to integrate the legacy code into newer environments[9]. The first and most obvious answer is to redevelop the software completely. This, however, is technically and economically unfeasible in most cases. Old applications perform critical business functions in

[3] **AdaCore**. Ada & SPARK for Visual Studio Code. Visual Studio Marketplace. https://marketplace.visualstudio.com/items?itemName=AdaCore.ada (29 October 2024).

[4] **Microsoft**. C/C++ for Visual Studio Code. Visual Studio Marketplace. https://marketplace.visualstudio.com/items?itemName=ms-vscode.cpptools (29 October 2024).

[5] **Stack Overflow**. Ada Tag. Retrieved by hovering over the tag on the Stack Overflow website. https://stackoverflow.com/tags (29 October 2024).

[6] **Reddit**. Ada Programming Community. https://www.reddit.com/r/ada/ (29 October 2024).

[7] **Stack Overflow**. C++ Tag. Retrieved by hovering over the tag on the Stack Overflow website. https://stackoverflow.com/tags (29 October 2024).

[8] **Reddit**. C++ Programming Community. https://www.reddit.com/r/cpp/ (29 October 2024).

[9] **Sneed, H. M**. 2000. Encapsulation of legacy software: A technique for reusing legacy software components. – Annals of Software Engineering, Vol. 9, pp. 293–313 [**Sneed** 2000]; **Colosimo, M.; De Lucia, A.; Scanniello, G.; Tortora, G**. 2009. Evaluating legacy system migration technologies through empirical studies. – Information and Software Technology, Vol. 51(2), pp. 433–447.

an acceptable and tested manner and it would be irrational to discard them completely. The other solution would be to reuse the software. Sneed argues that the correct answer for whether to redevelop or reuse depends on the quantity, quality and complexity of the existing software, as well as the time and budget constraints of the customer. Considering the facts that the S4 software suite is a large project developed over many years carrying significant business logic, and the customer has a limited time and budget, the better approach in this case would be to reuse the existing software.

Now that the decision to reuse the software has been made, there is the question of how to reuse it. The two basic alternatives to consider are conversion and encapsulation[10]. Conversion means that the old code would have to be significantly reworked to be ported to the new environment; it might have to be translated into another programming language or be re-engineered architecturally to integrate the business logic statically with the new code. This is the most costly and risky approach as it mirrors a new development process, but with added constraints imposed by the legacy code.[11]

Using the encapsulation (or wrapping) approach leaves the code in its original environment and aims to connect it dynamically to the new code. In this way, changes made to the code are minimal and the main complexity comes from connecting the wrapped software to the new interface. The wrapper exposes only the required functionality of the wrapped software through the encapsulation layer, which provides further flexibility for the integration.[12]

It was for this reason that wrapping was chosen as the integration method to be used in this project. No suitable automatic wrapping tool was found for the Ada language, therefore the wrapping had to be done manually.

The wrapping method is essentially a message passing mechanism: the wrapper receives incoming requests, reformats them to be suitable for the wrapped software, loads the wrapped software, and then invokes it (or only some functions) with the reformatted input. Then it takes the results of the wrapped software, reformats them, and sends them back to the requester. Wrapping generally consists of three steps: constructing the wrapper, adapting the target program, and testing the interaction.[13]

---

[10] **Sneed** 2000.

[11] Ibid.

[12] Ibid.

[13] Ibid.

In this project, the adaptation of the target program and testing of the interaction lie beyond the scope of our research as it is unknown which systems will be using the wrapped code. The solution is intended to be universal, so any S4 working group member could implement the software into their systems with few changes. Therefore, this project only researches and provides wrapped software solutions.

2.2. Integration Methods Based on Wrapping

Different wrapping-based integration methods can be used, where the difference comes from the interface layer.

1. Papers like Larmuseau and Clarke[14] and Yallop et al.[15] have used foreign function interfaces (FFI) for cross-language interoperations. Similarly, one possibility would be to create a wrapper for the S4 Ada code, which converts function input types into C language types, allowing the function to be called from any language that supports the C language Application Binary Interface (ABI). The result is a dynamically linked software library (.so on Linux, .dll on Windows). For safety reasons, consideration should be given to using a protocol for C language ABI calls. FFIs are notoriously difficult to deal with safely and can introduce undefined behaviour[16]. Therefore, to minimise the potential attack surface[17] and make it easier to ensure the code is correct, the number of FFI functions should be limited to the minimum necessary to achieve the required functionality—ideally just one. No commercial software was used for this research; most of the tooling is available only by paying for the compiler license. As the C interface is built into the Ada language specification and is not an add-on with special tooling, it was chosen as the FFI method.

---

[14] **Larmuseau, A.; Clarke, D**. 2015. Formalizing a secure foreign function interface – extended version. – Calinescu, R.; Rumpe, B. (eds.). Software Engineering and Formal Methods. SEFM 2015. Cham: Springer International Publishing, pp. 215–230.

[15] **Yallop, J.; Sheets, D.; Madhavapeddy, A**. 2018. A modular foreign function interface. – Science of Computer Programming, Vol. 164, pp. 82–97.

[16] **Turcotte, A.; Arteca, E.; Richards, G**. 2019. Reasoning About Foreign Function Interfaces Without Modelling the Foreign Language. – 33rd European Conference on Object-Oriented Programming (ECOOP 2019), pp. 5, 7–8.

[17] **Howard, M**. 2019. Attack Surface. Mitigate Security Risks by Minimizing the Code You Expose to Untrusted Users. – Microsoft Learn. https://learn.microsoft.com/en-us/archive/msdn-magazine/2004/november/security-tips-minimizing-the-code-you-expose-to-untrusted-users (28 October 2024).

2. Glatard et al.[18] described task-based job submissions and service-based code execution paradigms on grid computing. The task-based job submission approach involves the description of tasks through command-line interfaces, followed by the remote execution of application code. Mirroring this method, this paper proposes a command-line interface (CLI) wrapper integration technique where legacy S4 Ada code is encapsulated and invoked through command-line commands. The CLI accepts input data based on a predefined protocol and returns responses in the same manner.
3. Similarly, relying on the service-based code execution method, a web server wrapper for the S4 Ada code is proposed which enables access to S4 functionality over HTTP requests using protocols such as RPC, RESTful APIs, SOAP, GraphQL or similar. The web service wrapping technique was used in two examples here[19] and here[20] where case studies of legacy code integration into a service-oriented architecture were presented. Sneed[21] claimed that the benefits of web integration were seamless access to software functionalities, simplified development efforts, and dynamic updates. However, there were two main problems with this method. First, performance was not the best as sending calls with long parameter lists over networks takes time, so transmission times can become long. Therefore, it is advisable to limit the number of calls and parameters. Second, web interface complexity can increase when functionality increases, which might need excessive testing.

The possible integration methods identified for this study were, therefore, the FFI, CLI and web-based integrations.

---

[18] **Glatard, T.; Emsellem, D.; Montagnat, J**. 2006. Generic web service wrapper for efficient embedding of legacy codes in service-based workflows. – Grid-Enabling Legacy Applications and Supporting End Users Workshop, June. Paris, pp. 1–10. https://hal.science/hal-00683196/document (28 October 2024).

[19] **Canfora, G.; Fasolino, A. R.; Frattolillo, G.; Tramontana, P**. 2008. A wrapping approach for migrating legacy system interactive functionalities to service oriented architectures. – Journal of Systems and Software, Vol. 81(4), pp. 463–480.

[20] **Sneed, H. M**. 2006. Integrating legacy software into a service oriented architecture. – Conference on Software Maintenance and Reengineering (CSMR'06), pp. 11–14.

[21] Ibid.

3. Methodology for Measuring Integration Effectiveness

To facilitate a comparison between the chosen implementation methods, a structured methodology was devised. The selection of comparison metrics was tailored to the specific characteristics of the S4 software and its intended application domain. By employing a combination of objective and subjective metrics, this methodology aims to provide a comprehensive evaluation framework for comparing the performance, efficiency, and user experience of the chosen implementation methods.

3.1. Objective Metrics

Objective metrics provide measurable indicators of various aspects of the implementations. By analysing quantitative metrics, this paper aims to provide a clear comparison between the implementations identifying strengths, weaknesses, and areas for improvement.

1. **Central Processing Unit (CPU) utilisation during operation**: The efficiency of CPU resources utilised by each implementation during operation will be assessed. This metric, aligned with ISO/IEC 25002 standards[22], provides insights into the computational performance and resource management efficiency of the implementations.
2. **Random Access Memory (RAM) usage during operation**: The memory efficiency of the implementations will be evaluated through the monitoring of RAM usage. Adherence to ISO/IEC 25002 standards[23] enables a systematic assessment, aiding in the identification of potential memory leaks or excessive resource consumption affecting system stability and performance.
3. **Application speed in responding to queries**: The responsiveness of each implementation in handling queries will be quantified to gauge their processing efficiency. This industry-standard metric reflects the implementations' ability to efficiently process user requests, thus contributing to overall system performance. Responsiveness is measured using three fire mission use-cases, which are further described in section 6.1. Use-cases are defined as test code, thus ensuring the same input and starting parameters for all the implementations.

[22] **International Organization for Standardization**. ISO/IEC 25002:2024(E), 2014. Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Quality model overview and usage.

[23] Ibid.

3.1.1. Testing Environment

Objective metrics need to be tested in a reproducible environment. It must be noted that measurements are taken as relative to each other, not as absolute values. The underlying infrastructure does not reflect the real setup of the client. Hypervisor and VM specifications are recorded in order to ensure consistent and repeatable data.

The environment used to conduct tests:
- Hypervisor: VSphere 7.0.3.01700
- Guest VM OS: Debian 12 (Bookworm)
- CPU: 8 vCPU
- RAM: 16 GB
- Storage: 32GB virtual SSD

Debian is a popular freeware Linux distribution with a wide range of software packages. "Bookworm" is the codename for the fifth update released on 10 February 2024. Both implementations run inside Docker containers of the following specifications:
- Docker version: 26.1.3, build b72abbb
- Docker image: Ubuntu 20.04

Measurements of the implementations were taken by creating a simple framework in JavaScript. The tool set up the server using the child process module. For the web API the sending of requests and receiving of responses were handled using the node-fetch module. For the CLI integrations the requests were written directly into the standard input and read from the standard output streams. Request-response times were measured from the time the request was sent to the time the response was fully received. The act of measuring may have had an impact on results for the server's CPU usage by increasing the time spent between receiving a response and sending a new request, thus decreasing average CPU usage. However, this impact was considered negligible and the same for both implementations. Average CPU and current RAM usage of the implementations were monitored using the ps tool. Request-response time measurements were taken using the Node.js process.hrtime() function.
- Node.js version: v20.11.1
- ps version: 3.3.16

In both implementations, the server was initiated prior to executing a specific batch of requests. Once the batch was completed, the server was closed and re-opened for the subsequent batch. This guaranteed a consistent server state for each type of request and ensured that only the effects of that specific request were being measured.

The versions of the tested implementations and shared libraries are referenced by their latest commit hashes:

- CLI integration: 627bd2507a
- Web-API integration: 8948b7d2c3
- Json-parser: 3337076929
- Integration-utils: e093d1c9fc

3.2. Client Feedback

Client feedback as a subjective evaluation method captures the qualitative aspects of the implementations that may not be easily quantifiable but are nonetheless crucial in determining their success in a real-world mission-critical application. By examining these subjective metrics, this paper aims to uncover user perceptions, preferences, and pain points associated with each implementation.

1. **Time from concept to deliverable**. The speed and agility of the development process will be evaluated by measuring the time taken from conceptualisation to the delivery of a usable product. This metric, aligned with principles from the Agile Manifesto, assesses the efficiency and adaptability of the development approach.
2. **Developers' evaluation**. Subjective feedback from developers will be gathered to assess factors such as ease of development, code maintainability, and adherence to coding standards. This qualitative assessment, following industry best practices, offers insights into the challenges encountered and developer experiences with each implementation.
3. **User satisfaction**. User satisfaction information will be gathered to capture feedback on usability, reliability, and overall satisfaction with each implementation. This metric, derived from user satisfaction survey methodologies, provides valuable insights into the effectiveness of the implementations in meeting user needs and expectations.

4. Design Sketches for the Integration Architecture

Three different implementation methods were chosen for this study: command-line interface (CLI), dynamic link library (DLL), and web application programming interface (API). They all use the wrapping method but with different exposure layers.

1. **CLI**. In this architecture the S4 software is wrapped by a CLI layer, allowing users to interact with the system via command-line inputs. The CLI wrapper communicates directly with the Ada components, executing commands and returning results to the user. It is platform-dependent and would require specific command-line support on the host system.
2. **DLL**. In this case, the S4 software is exposed as a DLL. This setup allows other applications that support C language ABI to load and call functions from the DLL. The DLL serves as an intermediary, exposing specific functionalities of the S4 software without revealing its internal implementation. This method is quite flexible and supports integration with various software applications running on the same platform.
3. **Web API**. The S4 software is exposed via a web API which can be accessed over a network. This method uses HTTP/HTTPS protocols for communication, making it suitable for integration with web applications, mobile apps, or other services that can consume web APIs. The web API layer is built using the Ada Web Server (AWS) framework which handles the requests and translates them into actions performed by the S4 software. This method offers the highest level of flexibility and accessibility, allowing users to interact with S4 from any device connected to the internet.

Due to project time constraints, it was only possible to implement two of the three methods. CLI and web API were chosen as the most interesting avenues of study. Firstly, the DLL method had already been implemented before this study, therefore it did not have any practical output on real life and would have needed to be reworked just to be used in the study's objective measurements. Secondly, the DLL method still requires writing both Ada and an external wrapper (to CLI or web) and is therefore not functionally equivalent to the CLI and web API methods. Furthermore, wrappers around DLLs are already integrator-specific and would break the universality principle of the work.

4.1. Non-Functional Requirements

One aim of this work is to hand over the implementations to the client to aid them in their activities. Therefore, to ensure practical utility and real-life use of the implementations, input was also gathered from clients. Client meetings were held to align expectations with technical possibilities. The main requirement specified was Android compatibility. According to the client, the ideal expected outcome would be an Android intent mechanism integrated within the Android system which would allow them to define the intents and commands. These intents could represent commands or requests to execute specific functions provided by the legacy CLI, for example.

4.2. Hypotheses for Integration Candidates

The integrations are expected to demonstrate distinct performance characteristics based on the selected metrics. The following hypotheses were set:

- **CPU usage**. Both integrations are expected to show similar CPU usage since the core functionality and underlying library operations remain consistent across both methods. Any differences in CPU consumption will likely be minimal and attributable to the processing overhead specific to each integration method.
- **RAM usage**. Both integrations are expected to show similar RAM usage since the core functionality and underlying library operations remain consistent across both methods. Any differences in RAM consumption will likely be minimal and attributable to the handling of input/output operations specific to each integration method.
- **Response time.** Response times for the CLI integration will be faster compared to those for the web API integration. The hypothesis claims that the CLI method, by bypassing the additional layers required for HTTP requests, will process and return results more swiftly.
- **Time from concept to deliverable**. The CLI integration will have a shorter time from concept to deliverable compared to the web API integration. This is because the CLI method involves fewer layers of abstraction and less complexity in setting up communication between components.
- **Developers' evaluation**. Developers will evaluate the CLI integration more favourably in terms of ease of development and code maintainability but will recognise the web API integration as beneficial for extensive interoperability.

- **User satisfaction**. Users will express higher satisfaction with the CLI integration due to its superior performance in terms of CPU efficiency and response times. The web API integration will be appreciated for its user-friendliness.

These hypotheses will be validated through testing and analysis, providing insights into the most effective integration method for the NATO S4 software library in an operational context.

5. From Architecture to Implementation

This section describes the process from architecture to implementation of the CLI and web API method. The aim is to provide an understanding of the decisions made during the development process and their impact on the final implementations.

5.1. Challenges of Creating the Development Environment

The first challenge encountered was setting up the development environment. The S4 library, being written in the Ada programming language, requires specific versions of the GNAT compiler and GNU Compiler Collection (GCC). Since the web API and CLI integrations need to run on different platforms—such as Linux, Windows and Android—and different architectures—such as ARM64 and ARM Hard Float (ARMHF)—additional compilers and libraries were required for cross-compilation.

After extensive research and testing, the required steps to set up the development environment were detailed in a Dockerfile. This Dockerfile could then be used to create a Docker container which is a lightweight package that includes everything needed to run a piece of software, including the system tools, libraries, and settings. This ensured that all developers were working in the same environment and that the integrations could be built and run on any platform.

5.2. Final Designs

Since both CLI and web API implementations would require a request-response translation protocol layer, the decision was made to use a JSON based protocol for both implementations. The JSON based protocol was

chosen for its simplicity, human readability and widespread support across various platforms. Since a JSON parser would be required to translate input from JSON to a language that is recognised by the S4 library and then back again, the GNATcoll framework was chosen.

The integrations do not aim to map the S4 library functions one to one, but rather to encapsulate features that the library is most used for, based on and prioritised by the client's requests. These features often require calling multiple S4 library functions in a specific order. By wrapping them into user-friendly functions, this complexity can be abstracted away. Initially, the intention was to keep these functions unique to each integration. However, it became evident that not only are some of these features long and complex, but also the logic for the integrations would be similar once the requests were translated and passed into either implementation. Therefore, after the start of the development process, the decision was made to create a second shared library that would contain the core logic. In hindsight, the two shared libraries could have been combined into one, but since the development phase was already well underway it was agreed that the two libraries would remain separate for ease of understanding and maintainability.

Unlike most command-line tools that accept input from the user in the form of flags or arguments, the CLI integration only accepts input in the form of JSON objects. This design choice was made to keep the input format consistent with the web API integration. However, since many functions may share the same input structure, the CLI integration requires an additional field 'command' which specifies which route to call.

5.3. Code Structure

The code structure for the CLI integration is straightforward, consisting only of the 'main' file and the 'basic' file, which contains the basic routes for the version and status of the server. The version route provides information about the current version the server is running. The status route provides basic information about the health and availability of the server, often with a simple 'OK' return message indicating that the server is capable of correctly receiving and responding to requests. The main file sets up the CLI logic and starts waiting for input; once received, it is passed to the shared JSON-parser library. Based on the 'command' field, the desired function is called from the shared library directly.

The code structure for the web API integration is organised around a central main file which sets up the web server and contains links to the available routes. The logic for each type of route is separated into its own file. For instance, all functions related to the FCI (fire control input) database routes are contained within a single file: 'route-fci.adb'. This simple approach helps to make the codebase easier to understand and maintain. The route functions in turn call the shared library functions to access the S4 library.

5.4. Testing

In both integrations, testing is done by using the Jest testing framework which is based on the JavaScript language. For web API, every route is tested with a pre-defined input and the output is compared to a pre-defined snapshot which has previously been approved by a developer. Both positive and negative outcome tests are conducted to ensure the correct behaviour of the requested functions.

Due to the nature and wide range of possibilities of input variables, testing every scenario is not feasible. Therefore, tests are focused on the most common and critical cases. The tests are run automatically on every push to the code repository to ensure that faulty code is not merged into the main branch and that the integrations remain deterministic.

The way the tests are run on both integrations is kept as similar as possible to speed up development. Before any testing is done, databases are initialised in order to get the right information. The web API integration is tested by setting up a single instance of the web server. All test requests are sent simultaneously. The CLI integration is tested by creating a global process and then sending commands in JSON form. Web API results and CLI outputs are then compared to snapshots to verify their validity.

5.5. Technical Debt

Regarding technical debt, there are some areas that could be improved. The CLI integration could use a cleaner code structure; this could be achieved by moving every route's logic to its own file, similarly to how it is done in the web API integration.

The early adoption of the shared libraries helped to minimise code duplication in the main integrations. It did, however, become an issue in the JSON-parser shared library where the steps needed for converting the JSON input into a format suitable for the S4 library are numerous and repetitive. Midway

through development a generic helper function was created to validate and convert any JSON input field into their desired format, but not all functions were updated to use this helper function. This helper function could be expanded to all field conversions, which would help to reduce code duplication and make the library easier to maintain.

6. Comparison of the Integrations

This chapter explains how the testing of the integrations was done and compares the two integrations based on the qualitative and quantitative metrics outlined in the methodology. The results of the comparison are presented and discussed.

6.1. Comparison of Objective Metrics

Two main fire mission routes and one utility location route were chosen to test the integrations: 'basic fire mission', 'fire mission with weather data', and 'shift to location'. The fire mission routes were chosen as they are the most critical functions of both integrations. The 'basic fire mission' is the simplest form of the fire mission request, while the 'fire mission with weather data' is more complex as it requires additional data from external files. The 'shift to location' route was chosen as a utility route to test and compare the integrations' ability to handle simpler requests with potentially much shorter response and calculation times. The first measurement for each type of request was taken before any requests were sent, to measure the initial idle state of the server. Subsequent measurements were taken after the request was sent, and the response was received.

The servers were set up, each request was sent to the server 10,001 times, and the server was restarted for the next set of requests. A sample size of 10,001 requests per route was chosen to ensure that the results would be statistically significant and to ensure that the server would be under a consistent load for each request type. We found that increasing the number of requests beyond this point did not significantly change results. The additional 1 request was used to measure the initial idle state of the server before requests were sent. CPU usage was measured as an average over the set lifetime of the server. RAM usage was measured as the current usage at the time of the measurement. Response times were calculated from the time the request was sent to the time the response was fully received.

6.1.1. CPU Usage

Since the underlying core logic was the same for both integrations, CPU usage was expected to be similar for both of them. The 'fire mission with weather data' request was the most complex in its input and output data, and therefore was expected to have the highest CPU usage. The 'shift to location' request was the simplest and was expected to have the lowest CPU usage. The results of the CPU usage measurements and comparison are presented in Table 1 and Figure 1. The percentage covers all available CPU cores.

Table 1. Performance comparison of different requests and integrations

Test Name	CLI CPU [%]	Web CPU [%]
Basic fire mission	69.70	54.24
Fire mission with weather data	68.63	55.32
Shift to location	5.36	4.13

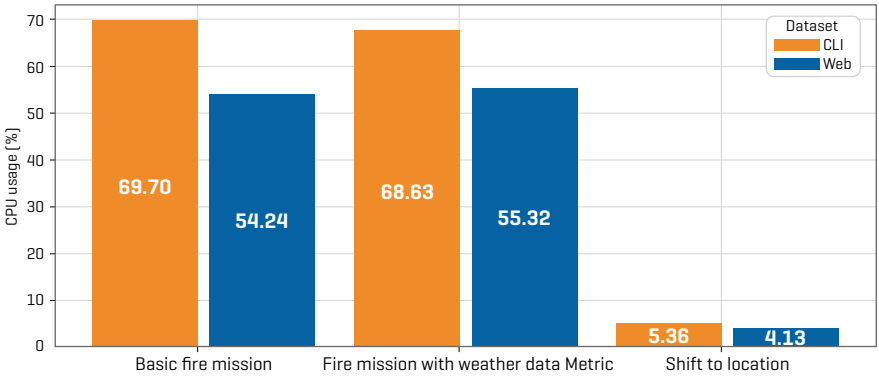

Figure 1. Average CPU usages for different requests and integrations

Within the same integration there were minor differences in CPU usage between the different fire mission requests. Even though the weather data was held in a separate file, it was handled efficiently enough to have no significant impact on CPU usage over the basic fire mission request. As expected, the 'shift to location' request had the lowest CPU usage, being the simplest request regarding input and output data as well as the number of S4 library functions called. Differences in CPU usage between the integrations were significant. This was likely due to the overhead of the HTTP protocol in the web API integration, where some time of the request was dedicated to setting up the

connection and sending the response. The CLI integration communicated directly with the server using the standard input-output streams, which was more efficient in processing the requests. The reason the CLI integration used more CPU than the web API integration could be attributed to the fact that more requests could be sent to the server in the same period, meaning the server was under more load and therefore used more CPU. This is supported by Figure 2 which shows that the CLI integration could handle more requests per minute than the web API integration. The difference in results handled by CLI and web API was also proportional to the complexity and length of the requests. The CLI integration handled about 18 percent more fire mission requests per minute than the web integration, while the much simpler and shorter 'shift to location' request was handled 59 percent more efficiently by the CLI integration, indicating that the overhead of the HTTP protocol is more significant for shorter requests.

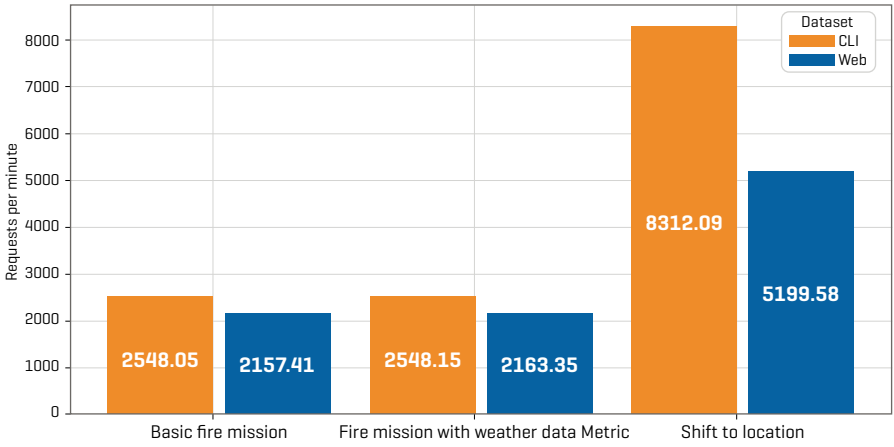

Figure 2. Average requests per minute for different requests and integrations

6.1.2. RAM Usage

RAM usage was expected to be similar across all requests. An initial increase in RAM usage was expected after the first request which may have initialised some variables. However, after the first request RAM usage should have remained constant. The results of the RAM usage measurements are presented in Table 2 and Figure 3; CLI request RAM usages are shown in Figure 4 and web API request RAM usages in Figure 5.

Table 2. Average RAM usage for different requests and integrations

Dataset	Test	Average RAM Usage (MB)
CLI	Basic fire mission	446.66
CLI	Fire mission with weather data	444.71
CLI	Shift to location	443.46
Web	Basic fire mission	448.07
Web	Fire mission with weather data	450.11
Web	Shift to location	445.00

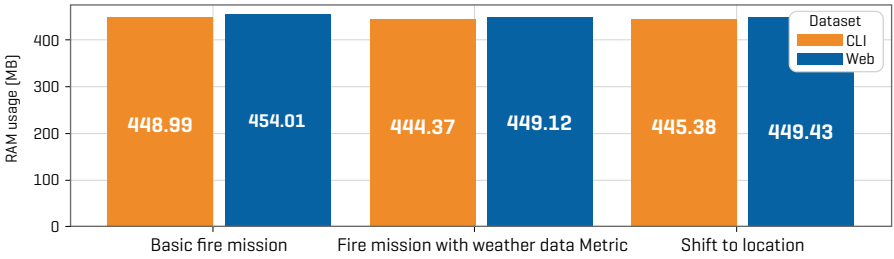

Figure 3. Average RAM usages for different requests and integrations

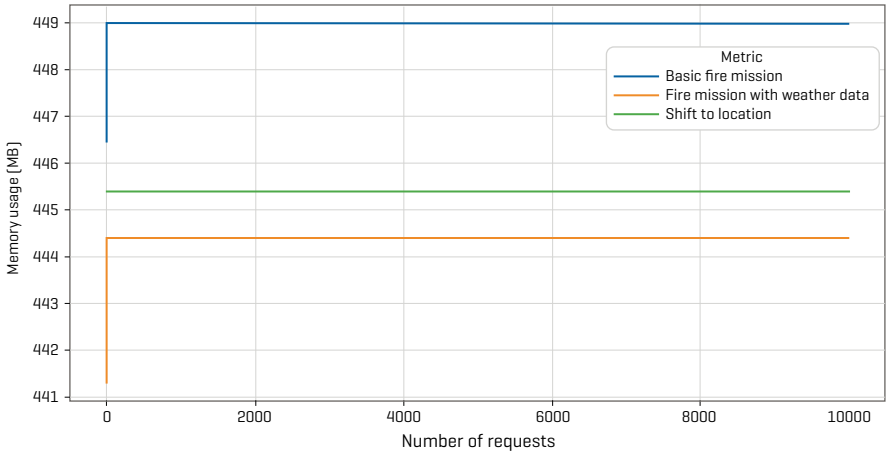

Figure 4. RAM usage over time for CLI

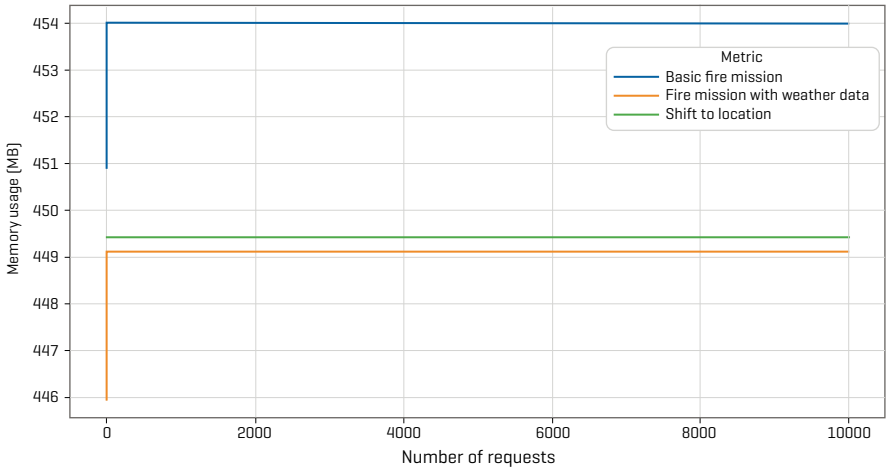

Figure 5. RAM usage over time for web

Negligible differences in RAM usage were observed between all requests and integrations. An initial increase in RAM usage was seen where request variables are initialised, after which RAM usage remained constant. This indicated that the servers were not affected by the number or the complexity of the request and were not leaking memory.

6.1.3. Response Times

Request-response times were expected to be slightly longer for the web API integration due to the overhead of the HTTP protocol. In comparison, the CLI integration communicated directly with the server using the standard input-output streams. The response times for the 'fire mission with weather data' request were expected to take the longest due to the complexity of the request and the need to access external weather data files. The 'shift to location' request was expected to have the shortest response time due to the simplicity of the request.

The results of the different response time measurements are presented in Table 3 and Figure 6.

Table 3. Duration metrics for CLI and web tests

Test Name	CLI durations [ms]			Web durations [ms]		
	Min	Avg	Max	Min	Avg	Max
Basic fire mission	15.97	16.16	16.57	16.60	16.81	17.41
Fire mission with weather data	15.94	16.12	16.57	16.63	16.80	17.25
Shift to location	0.15	0.17	0.21	0.76	0.88	1.12

The differences in response times between the integrations were as expected. The overhead for the HTTP protocol in the web API integration was especially visible for the 'shift to location' request. The CLI integration was able to handle the requests more efficiently, but the more complex the request, the less significant the differences in response times became.

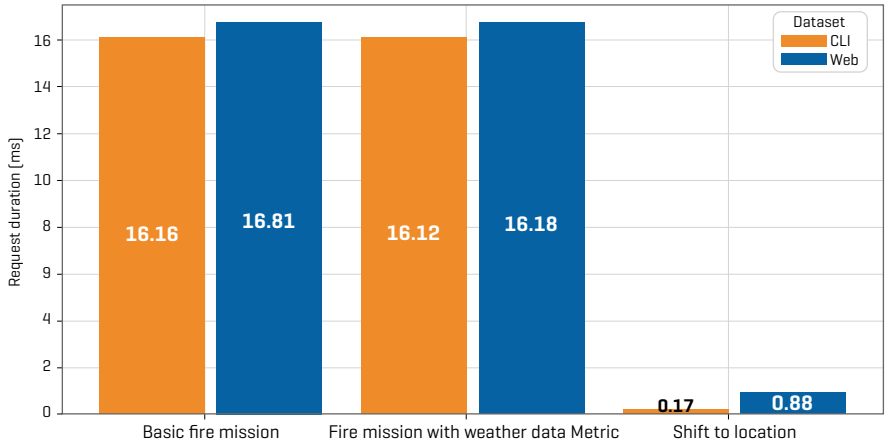

Figure 6. Request durations for different requests and integrations

6.1.4. Summary of Objective Metrics

Objective metrics provided a comprehensive comparison of the CLI and web API integrations. CPU usage, RAM usage, and response times were measured for three different types of requests: 'basic fire mission', 'fire mission with weather data', and 'shift to location'. Results indicated that the CLI integration was more efficient at handling a higher number of requests that required a low amount of calculation time, since the overhead of the HTTP protocol in the web API integration added a fixed amount of extra processing time for each request. This differed from the hypothesis set, as the CPU was expected to be the same for both implementations. RAM usage was similar across all requests and integrations, aligning with the hypothesis set. Response times were marginally longer for the web API integration due to the overhead of the HTTP protocol, as expected with the hypothesis set, but the differences became less significant for more complex requests. Due to the nature of the S4 library, the strengths of multi-threading and parallel processing could not be utilised for the web API integration but, even with the overhead of the HTTP protocol, the web API integration was still able to handle a significant number of requests in a short time frame. In practical applications, however,

the slightly lower benchmark results of the web API integration might not prove significant enough to outweigh the ease of adaptability and integration with various systems.

6.2. Comparison of Client Feedback

An interview was conducted with the client to gather feedback on the two integrations. The client, utilising an Android-based tablet setup and currently relying on manual work with ballistics tables, provided insight into their preferences and concerns. Three key metrics—time from concept to deliverable, developers' evaluation, and user satisfaction—were evaluated based on client feedback.

6.2.1. Interview Analysis

The client indicated a reasonable familiarity with both CLI and web-based applications. They employ Android tablets and currently use ballistics tables for operational purposes. This setup underscores the need for integrations that are compatible with Android systems and can operate efficiently within a mobile environment. Because of this, neither method was deemed entirely suitable due to stringent security requirements within their existing architecture. An internal API was suggested as a potential solution, highlighting the need for a flexible integration strategy. Security regulations and the need to comply with ARM64 architecture were identified as significant barriers to integration with either solution.

The client prioritised metrics related to CPU and RAM usage, critical for maintaining battery efficiency. This consideration is particularly relevant in mobile systems where power management is a key concern. The interview responses highlight that the integration needs to be both lightweight and efficient.

The client acknowledged that long-term maintenance and scalability would depend on the frequency of updates to supporting libraries and FCI table changes.

DLL could be considered a third alternative way of solving the customer's problems. However, the DLL approach lies beyond the scope of this research as there was no possibility for the authors of this article to develop it further. In order to accurately compare the DLL objective metrics function-to-function with Web and CLI applications, the DLL would have needed modifications, and for this limiting reason it was not considered.

6.2.2. Evaluation Metrics

The speed and agility of the development process are crucial in evaluating the effectiveness of integration methods. The time taken from conceptualisation to the delivery of a usable product reflects the efficiency and adaptability of the development approach. The client's responses indicate that both integration methods—web and CLI—would involve significant development time due to security and compatibility challenges. They estimate that both methods would require 3–6 months to implement, test, and deploy the solution; the time from concept to deliverable is therefore expected to be similar for both methods and no strong preference was indicated here.

Subjective feedback from developers provides essential insights into ease of development, code maintainability, and adherence to coding standards. The client indicated satisfaction with the code quality, documentation and README text files of both codebases. They highlighted that they are familiar with both integration methods but strict security requirements as well as a significant rework of their current workflow would complicate any development process using either method. One interviewee expressed personal preference for the CLI version, however this was not for any specific technical reasons.

User satisfaction is a key metric that captures usability, reliability, and overall satisfaction with implementation. The client emphasised the importance of minimising CPU and RAM usage to enhance battery life, which directly impacts end user satisfaction. Ensuring that the integrated solution is intuitive and reliable is essential for achieving high user satisfaction levels. As RAM usage was similar across both integrations, and CPU usage was slightly better for the CLI integration, the client expressed a preference for the CLI method. The client, however, did not see any significant difference in end user satisfaction between the two methods regarding the user interface and user experience.

6.2.3. Summary of Client Feedback

Client feedback provided insight into the strengths and weaknesses of CLI and web API integrations. The client concluded that no significant differences were found between the integrations from their point of view, and therefore they do not have any clear preference regarding which method they prefer. The client emphasised the importance of efficient CPU and RAM usage due to the battery efficiency crucially needed in mobile systems. As shown by an objective comparison of the methods, the CLI method provided slightly better

CPU usage and was therefore somewhat more preferred by the client; additionally, one interviewee expressed a personal preference for the CLI method. Overall client feedback on the time from concept to deliverable, developers' evaluation, and end user satisfaction produced the result that neither method was significantly better than the other; both would need considerable development time to implement into their workflow, and both were seen as equally satisfactory to the end users. As the CLI method demonstrated marginally better CPU usage, it was speculated that it would provide better battery efficiency to the end user.

7. Limitations and Future Work

This study aimed to provide a comprehensive comparison of different software implementation methods; however, some limitations should be considered when interpreting the results.

The selection of comparison metrics was based on the software's purpose and available resources. While the chosen metrics encompassed a range of objective and subjective aspects, they may not have captured every relevant aspect of the implementations. External factors may have impacted the comparison process of quantitative metrics, even with the best efforts to keep the comparison environment the same for both implementations. The evaluation of subjective metrics such as developer and user satisfaction inherently introduce a degree of subjectivity and individual preferences, thus influencing the results.

Additionally, the sample size and diversity of the software implementation methods under comparison were limited by practical constraints. As only two different implementations were developed and other methods were discarded, the conclusions drawn from this study may not be fully representative of other software implementation scenarios and might influence the generalisability of the findings.

While this study provides valuable insight into the strengths and weaknesses of the software implementation methods examined, it is not exhaustive, and future research should consider these limitations. The authors acknowledge that a complete implementation of the two integrations into an existing workflow would provide practical insight into their operation and usability. Due to time constraints and the scope of this paper, this was not feasible. Future research endeavours should aim to address this gap. Additionally, comparable DLL functionality could be tested to provide more insight into how it compares with the proposed Web and CLI approaches. This would enable us to understand more clearly which method is more suitable for the client.

8. Conclusions

This comparative study into the integration of the NATO S4 software library highlighted the potential and the challenges of modernising legacy systems. Both CLI and web API methods were found to be feasible solutions, each with slightly different considerations. The CLI method showed marginally better performance in terms of CPU usage, which can be critical in resource-constrained environments such as mobile. However, the web API method offers greater flexibility and accessibility, making it suitable for diverse integration scenarios. Client feedback emphasised the importance of efficient resource usage, ease of maintenance, and adaptability to existing workflows; neither method was said to be better than the other. Considering both objective and subjective metrics, it cannot be conclusively determined which integration method is more suitable for the S4 software suite. As technical test results differed only marginally, the decision ultimately depends on the system integrating the software, and the integrator. Future research should focus on addressing the limitations identified in this study, such as the need for more comprehensive performance metrics and the exploration of additional integration methods. This work emphasises the importance of carefully selecting integration techniques based on specific application requirements and resource constraints.

LIISA SAKERMAN, MSc
System Analyst, Foundation CR14

MADIS-MIKK REMMET
Software Developer, Foundation CR14

RAIT ROTŠAN
Software Developer, Foundation CR14

KAAREL ALLEMANN, MSc
Software Architect, Foundation CR14

KRISTI REISPASS, BSc
Senior Software Developer, Foundation CR14