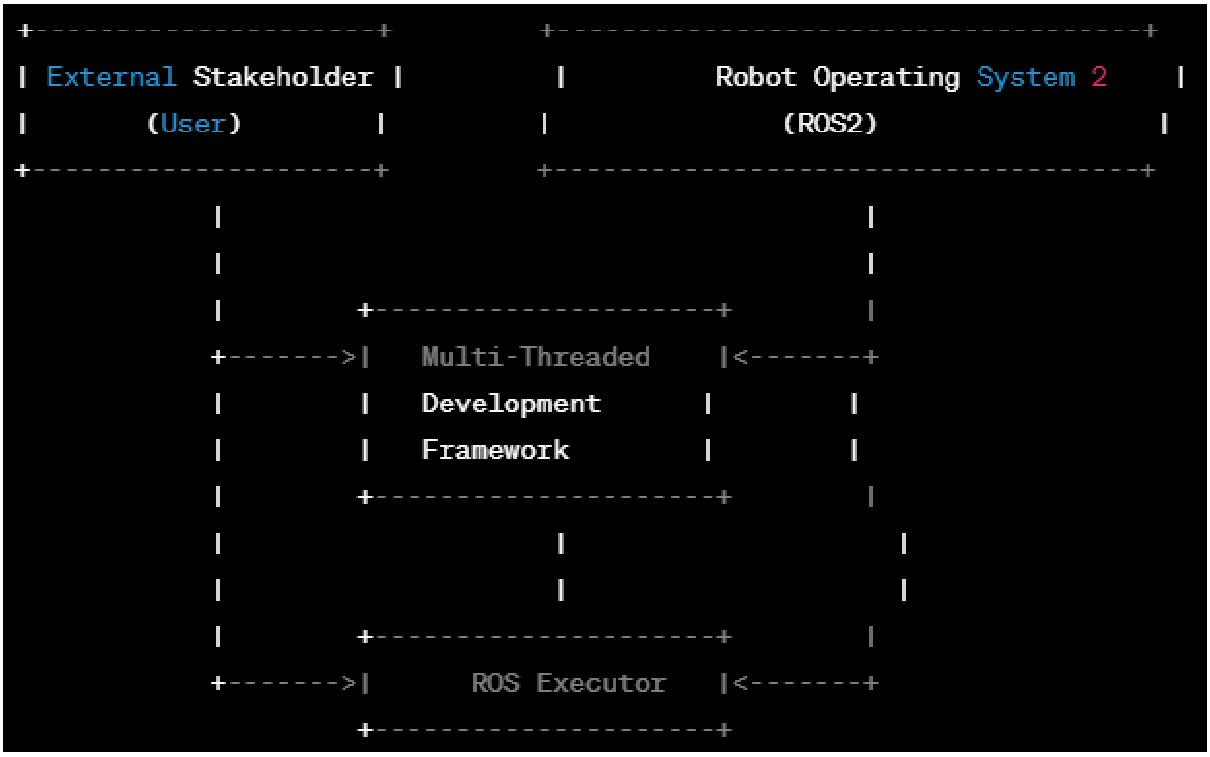
1. For the text below, summarise the problem and suggest a software system architecture solution. The main stakeholders in this case are: - Software system architects - DevOps engineers - Developers Problem, context, and related work Robot Operating System 2 (ROS) is a popular and widely adopted robotics engineering framework. Its main goal is to provide robotics engineers a platform that allows for the development of robotics applications. Framework's infrastructure, which is open source, is based on numerous distributed packages. ROS supports a wide range of devices, starting from small embedded devices up to autonomous industry robots. ROS was developed with real-time capabilities in mind. Since this technology is in its infancy, the ROS community encourages industry and academic research. Academic research in ROS has covered many aspects of robotics development and has achieved quantifiable results regarding soft and hard real-time constraints, lowering latency, and in general, improving performance. However, because of its complicated vertical stack of technologies, there is still room where significant performance benefits could be extracted. The default way to extract performance with aim to lower execution time is by utilizing multi-cores. However, ROS with its complex infrastructure and requirement from robotics engineers to learn parallelization from C/C++ perspective, make it hard to utilize multi-cores properly. Therefore, utilizing multi-cores to lower execution time is one of the main motivators that guides research in ROS. By lowering execution time we gain performance benefits and allow other parts of ROS infrastructure, which are influenced by latency and real-time constraints, to be positively impacted as well. Robots are very heterogeneous devices. Therefore, robots may come with a vast array of different system requirements and system constraints. Some of the system constraints are critical when it comes to the correct functionality of the robots. These critical constraints deal with the overall performance of the robotics application. In the scope of overall performance of the robotics application, the main goal of robotics engineers is to lower execution time or make a particular function execute faster. To facilitate this, we refer to utilization of multi-core processors. However, required specialized knowledge regarding concurrency and expertise, which robotics engineers do not have, slows them down in building their robotics application and decreases efficiency. Robotics engineers are required to understand the architecture of the CPU and how it interacts with main memory. On top of that robotics engineers are required to understand the intricacies regarding caches and data locality. Complemented with this is a software stack that is utilizing the CPU. They need to learn how to use multi-threading libraries, which is not simple. Functions for creating and managing threads require thinking about multiple data-flows in software stack. As a consequence, robotics engineers are required to understand in low-level detail how multi-threading works and how prioritization, scheduling, and affinity combine to solve the problem of lowering execution time. To conclude, the main gap is that robotics engineers do not have an intuitive way of utilizing multi-cores to be able to lower execution time. They are required to know low-level details regarding parallelization and have to develop multi-threading applications. Additionally, robotics engineers do not have accessible interfaces to control Operating System mechanisms regarding prioritization, scheduler type, and affinity Requirements for the solution Functional requirements • SR.01: Framework to facilitate development of multithreaded applications in ROS2 - Framework to assist robotics engineers in developing software that takes advantage of multi-cores in order to reduce average execution time, integrated in ROS2. • SR.02: Pre-locking heap memory - The framework enables pre-locking of heap memory during initialization of the application in order to avoid failed memory allocations, which can occur if application requires significant amount of memory. • SR.03: Process priority configuration - The framework provides interface for setting priorities of threads in order to avoid priority inversion. • SR.04: Measurement of heap allocations - The framework offers a visualization on how many memory allocations happened, and it should be easily accessible to the robotics engineers • SR.05: Measurement of latency. - The framework has a mechanism that

measures latency • SR.06: Measurement of jitter - The framework has a mechanism that measures jitter • SR.07: Measurement of CPU caches usage & The framework has a mechanism that measures CPU cache usage • SR.08: Measurement of memory fetches. - The framework has a mechanism which measures memory fetches. • SR.09: Executor type - Robotics engineers write nodes' configuration as input to the framework. The framework outputs the best performing executor type configuration containing nodes. & ROS executor • SR.10: Node to process assignment - Robotics engineers write nodes' configuration as input to the framework. The framework outputs the best performing configuration. & node assignment, thread, process • SR.11: Allocator type - The framework accepts the message data types that are used throughout the application and it advises robotics engineers on which allocators to use. • SR.12: Easy to use user interface - All information and capabilities of the framework should be gathered in easy-to-use user interface. • SR.13: Guideline - Robotics engineers will be able to express their intentions of parallelizing the code in the framework. The framework will output guidelines for robotics engineers. • SR.14: Parallel Configuration - The framework will accept non parallelized code and will turn it into parallel configuration. & Parallel Configuration Quality requirements • SR.01: Increased CPU Utilization - The framework facilitates the development of multi-threaded applications, and as a result, increased CPU utilization. If the number of cores are n,(and n>1), then utilization is at least 0.7*n. • SR.02: Decreased Heap Allocation - The framework facilitates the mechanism with pre-locking memory during the start of the execution of the application. At least 50 percent decrease in heap allocations regarding memory is expected. • SR.03: Increased general performance of the application - The framework facilitates the mechanism for process priority configuration. We expect an increase in general performance in a region of 10 percent. • SR.04: Decreased average execution time - The framework facilitates the mechanism for decreasing average execution time by utilizing multi-threading. • SR.05: Increased CPU Cache utilization - We expect an increase in CPU Cache utilization by 30 percent. • SR.06: Decreased Memory Fetches - We expect to have a decreased memory fetches by 20 percent as a result of increased CPU Cache utilization. • SR.07: Increased productivity of robotics engineers - The facilities of the framework will output parallelized configurations. Compared with by-hand optimization, this process will be expected to take 30 percent less time. & increased productivity • SR.08: Usability - The framework will include an easy to use user interface. We expect robotics engineers to quickly understand the framework and use it efficiently. • SR.09: Modularity - The framework's design will closely follow ROS design philosophy. We expect the framework to be at least modular as ROS and to fit on top of ROS. & modularity Use cases • UC.01: Direct Inverse Kinematics - Sensors, Actuators, Moveable joints, Rotating Joints & Inverse kinematics is a field which concerns itself with correct positioning of the joints in order for end-actuator to reach desired position. As the name indicates, the positioning is worked out from the end-actuator backwards to the first-actuator. • UC.02: Detecting human faces and torso orientation - People in a scene, robot scanning the scene - Image processing is a very important field in robotics engineering. This use case tries to identify the way of doing image processing in robotics, so we can learn how to parallelize common functions. • UC.03: Kalman Filter & A specific robot does sensor fusion and uses kalman filter for prediction - Prediction of the measurements. Kalman filtering is a mathematical approach on how to improve our measures that come from the sensors. • UC.04: Assessing execution time of real-time loop - Real-Time Loops, Function, Time & Since ROS was built with real-time capabilities in mind, measuring real-time loop execution time can give us an insight on performance improvements.

The problem is that robotics engineers lack an intuitive way to utilize multi-core processors to lower execution time, due to the complexity of ROS infrastructure and the required specialized knowledge in concurrency. The solution is to develop a framework that facilitates the development of multi-threaded applications in ROS2, integrated with ROS executor, and provides a user interface with features such as pre-locking heap memory, process priority configuration, measurement of heap allocations, latency, jitter, CPU caches usage, and memory fetches. The framework also advises on executor type, node to process assignment, and allocator type, and can output guidelines and parallel configurations. The quality requirements include increased CPU utilization, decreased heap allocation, increased general performance, decreased average execution time, increased CPU cache utilization and productivity of robotics engineers, and modularity. The use cases include direct inverse kinematics, detecting human faces and torso orientation, Kalman filter, and assessing execution time of real-time loop.

2. Suggest architecture of a conceptual solution for this problem. We need a C4 system context diagram (https://c4model.com/#SystemContextDiagram). Please use text (arrows, dots, different shapes, and different colours) to visualise the solution. Consider the following: - Identify relevant external systems (if any) and add them to the diagram. - Identify relevant external stakeholders (if any) and add them to the diagram. - Identify high level abstract data types transferred between components and stakeholders. - Summarise data types transmitted between components. Add this information to the connections and interfaces between components.

C4 System Context Diagram for a conceptual solution to the problem:



```
+---------------------+         +-------------------------------------+
| External Stakeholder |         |         Robot Operating System 2    |
|       (User)        |         |              (ROS2)                 |
+---------------------+         +-------------------------------------+
            |                                       |
            |                                       |
            |        +---------------------+        |
            +------->|   Multi-Threaded    |<-------+
            |        |   Development      |        |
            |        |   Framework        |        |
            |        +---------------------+        |
            |                    |                    |
            |                    |                    |
            |        +---------------------+        |
            +------->|      ROS Executor   |<-------+
                     +---------------------+
```

The diagram shows an external stakeholder, the user, who interacts with a multi-threaded development framework designed to help robotics engineers utilize multi-core processors in order to lower execution time. The framework is integrated with the Robot Operating System 2 (ROS2) and provides interfaces for setting process priorities, pre-locking heap memory, measuring heap allocations, latency, jitter, CPU caches usage, and memory fetches, among other things. The framework also advises on executor type, node to process assignment, and allocator type, and can output guidelines and parallel configurations.

The high-level abstract data types transferred between components and stakeholders include:

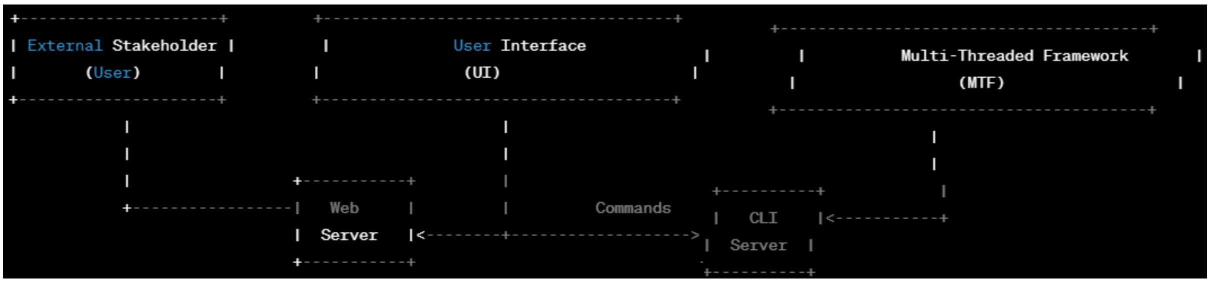
- User inputs: Commands and specifications from the user to the framework.
- ROS2 packages: Distributed packages of ROS2 infrastructure.
- Multi-threaded code: Code parallelized by the framework.
- Data measurements: Data measuring latency, jitter, CPU caches usage, and memory fetches.
- Configuration output: Optimal executor type, node to process assignment, and allocator type configurations.

The connections and interfaces between components transmit the following data types:

- User inputs to the Multi-Threaded Development Framework.
- ROS2 packages to the Multi-Threaded Development Framework.
- Multi-threaded code from the Multi-Threaded Development Framework to ROS Executor.
- Data measurements from the Multi-Threaded Development Framework to the user.
- Configuration output from the Multi-Threaded Development Framework to the user.

3. Suggest architectural components that we need to solve this problem. We need a C4 container solution diagram (https://c4model.com/#ContainerDiagram). Please use text (arrows, dots, different shapes, and different colours) to visualise the solution. Consider the following: - Define how the main stakeholder will use the solution. Add it to the diagram.. - Which quality properties are important? Suggest adequate architectural solutions to address them (architectural style, design patterns, design tactics, components and their relationships). Structure the components according to it. - Define data types transferred between components. - Summarise data types transmitted between components. Add this information to the connections and interfaces between components. - Define interfaces between components. Add information about required and provided interfaces to the components, along with the data types. - Document which anti-patterns we need to avoid when implementing this solution.

C4 Container Diagram for a conceptual solution to the problem:



```
+--------------------+      +-----------------------------------+
| External Stakeholder |    |            User Interface          |     +-----------------------------------+
|       (User)       |      |             (UI)                 |   |    |      Multi-Threaded Framework     |
+--------------------+      +-----------------------------------+       |              (MTF)               |
          |                                  |                         +-----------------------------------+
          |                                  |                                        |
          |             +----------+         |                                        |
          +-------------|   Web    |         |          Commands      +---------+      |
                        |  Server  |<--------+----------------------->|  CLI    |<----------+
                        +----------+                                  | Server  |
                                                                      +---------+
```

The diagram shows the external stakeholder, the user, interacting with the solution through a web or command line interface. The user can send commands and specifications to the multi-threaded framework, which helps robotics engineers utilize multi-core processors in order to lower execution time. The framework is integrated with the Robot Operating System 2 (ROS2) and provides interfaces for setting process priorities, pre-locking heap memory, measuring heap allocations, latency, jitter, CPU caches usage, and memory fetches, among other things. The framework also advises on executor type, node to process assignment, and allocator type, and can output guidelines and parallel configurations.

Important quality properties for this solution include:

- Performance: The framework should lower execution time and increase CPU utilization, while reducing heap allocations and memory fetches.
- Usability: The user interface should be easy to use and understand.
- Modularity: The solution should follow the ROS design philosophy and fit on top of ROS.

To address these properties, the solution can use the following components:

- Web Server: Handles incoming requests from the web interface.
- Command Line Interface (CLI) Server: Handles incoming requests from the command line interface.
- Multi-Threaded Framework: Implements the functionality of the multi-threaded development framework, utilizing ROS2 packages.
- ROS Executor: Implements the ROS2 executor for running multi-threaded code.

The data types transferred between components include:

- User inputs: Commands and specifications from the user to the framework.
- ROS2 packages: Distributed packages of ROS2 infrastructure.
- Multi-threaded code: Code parallelized by the framework.
- Data measurements: Data measuring latency, jitter, CPU caches usage, and memory fetches.
- Configuration output: Optimal executor type, node to process assignment, and allocator type configurations.

The connections and interfaces between components transmit the following data types:

- User inputs from the web server or CLI server to the Multi-Threaded Framework.
- ROS2 packages from the Multi-Threaded Framework to the ROS Executor.
- Multi-threaded code from the Multi-Threaded Framework to the ROS Executor.
- Data measurements from the Multi-Threaded Framework to the user.
- Configuration output from the Multi-Threaded Framework to the user.

The interfaces between components include:

- Web Server: Provides a RESTful API to receive user commands and specifications. Accepts and returns JSON data.
- Command Line Interface (CLI) Server: Provides a command line interface to receive user commands and specifications. Accepts and returns JSON data.
- Multi-Threaded Framework: Provides a library interface to be used by the web server and CLI server to execute multi-threaded code. Accepts and returns ROS2 packages and configuration data.
- ROS Executor: Provides an interface for executing multi-threaded code. Accepts and returns ROS2 packages.

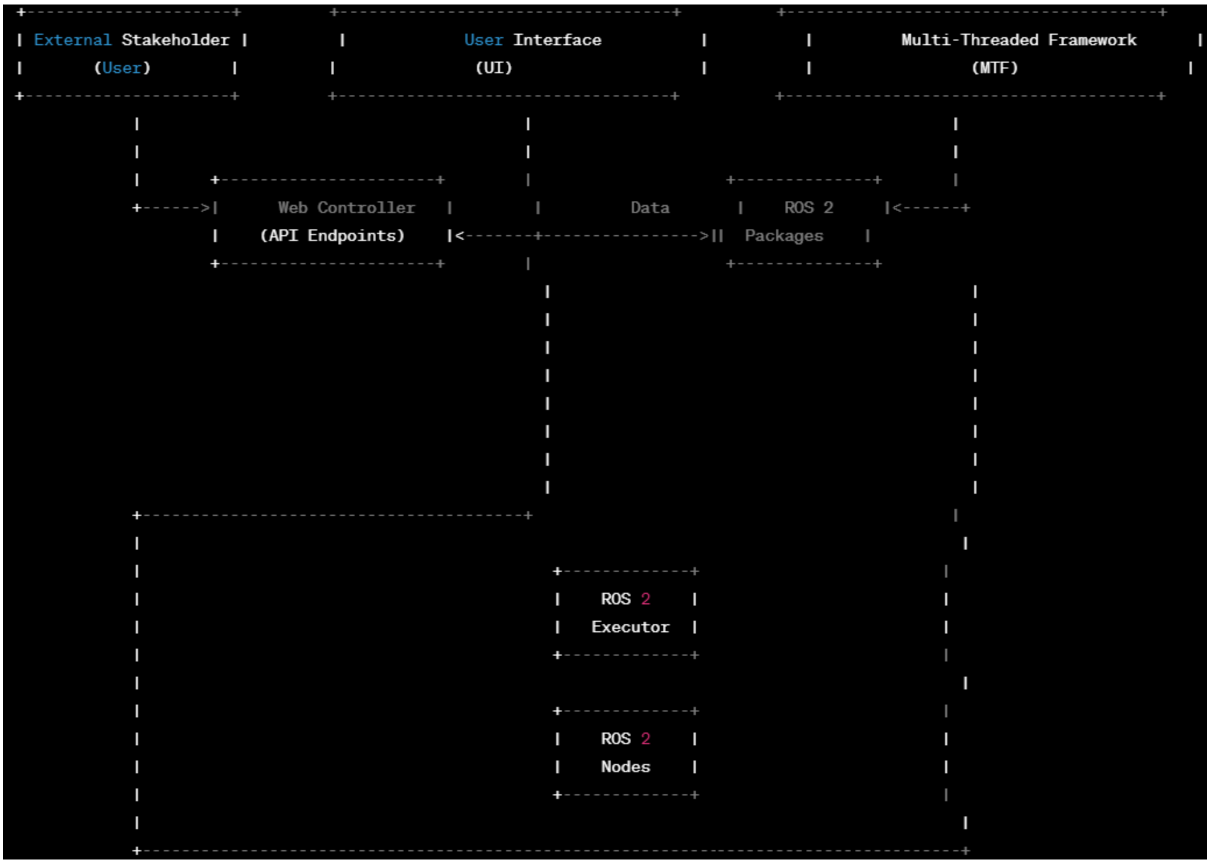
Anti-patterns to avoid when implementing this solution include:

- Monolithic architecture: The solution should be modular and follow the ROS design philosophy.

- Tight coupling: Components should be dec

4. Suggest architectural components that we need to solve this problem. We need a C4 component solution diagram (https://c4model.com/#ComponentDiagram). Please use text (arrows, dots, different shapes, and different colours) to visualise the solution. Consider following: - Define how the main stakeholder will use the solution. Add it to the diagram. - Which quality properties are important? Suggest adequate architectural solutions to address them (architectural style, design patterns, design tactics, components and their relationships). Structure the components according to it. - Define data types transferred between components. - Summarise data types transmitted between components. Add this information to the connections and interfaces between components. - Define interfaces between components. Add information about required and provided interfaces to the components, along with the data types. - Document which anti-patterns we need to avoid when implementing this solution.

C4 Component Diagram for a conceptual solution to the problem:



```
+---------------------+         +-----------------------------------+         +--------------------------------------+
| External Stakeholder |         |            User Interface          |          |        Multi-Threaded Framework       |
|       (User)        |         |              (UI)                   |          |                (MTF)                |
+---------------------+         +-----------------------------------+         +--------------------------------------+
            |                                       |                                           |
            |                                       |                                           |
            |       +----------------------+        |                    +--------------+       |
            +------>|      Web Controller   |        |         Data       |   ROS 2      |<------+
                    |    (API Endpoints)    |<-------+---------------->|| Packages    |
                    +----------------------+        |                    +--------------+
                                                      |                                           |
                                                      |                                           |
                                                      |                                           |
                                                      |                                           |
                                                      |                                           |
                                                      |                                           |
                                                      |                                           |
                                                      |                                           |
            +---------------------------------------+                                           |
            |                                                                                    |
            |                                          +-------------+                         |
            |                                          |    ROS 2    |                         |
            |                                          |   Executor  |                         |
            |                                          +-------------+                         |
            |                                                                                    |
            |                                          +-------------+                         |
            |                                          |    ROS 2    |                         |
            |                                          |    Nodes    |                         |
            |                                          +-------------+                         |
            |                                                                                    |
            +------------------------------------------------------------------------------------+
```

The diagram shows the external stakeholder, the user, interacting with the solution through a web or command line interface. The user can send commands and specifications to the multi-threaded framework, which helps robotics engineers utilize multi-core processors in order to lower execution time. The framework is integrated with the Robot Operating System 2 (ROS2) and provides interfaces for setting process priorities, pre-locking heap memory, measuring heap allocations, latency, jitter, CPU caches usage, and memory fetches, among other things. The framework also advises on executor type, node to process assignment, and allocator type, and can output guidelines and parallel configurations.

Important quality properties for this solution include:

- Performance: The framework should lower execution time and increase CPU utilization, while reducing heap allocations and memory fetches.
- Usability: The user interface should be easy to use and understand.
- Modularity: The solution should follow the ROS design philosophy and fit on top of ROS.

To address these properties, the solution can use the following components:

- Web Controller: Implements API endpoints for the web interface.
- ROS 2 Packages: Distributed packages of ROS 2 infrastructure.
- ROS 2 Executor: Implements the ROS 2 executor for running multi-threaded code.
- ROS 2 Nodes: Implements ROS 2 nodes for running multi-threaded code.
- Multi-Threaded Framework: Implements the functionality of the multi-threaded development framework, utilizing ROS 2 packages.

The data types transferred between components include:

- User inputs: Commands and specifications from the user to the framework.
- ROS 2 packages: Distributed packages of ROS 2 infrastructure.
- Multi-threaded code: Code parallelized by the framework.
- Data measurements: Data measuring latency, jitter, CPU caches usage, and memory fetches.
- Configuration output: Optimal executor type, node to process assignment, and allocator type configurations.

The connections and interfaces between components transmit the following data types:

- User inputs from the Web Controller to the Multi-Threaded Framework.
- ROS 2 packages from the Multi-Threaded Framework to the ROS 2 Executor and ROS 2 Nodes.
- Multi-threaded code from the Multi-Threaded Framework to the ROS 2 Nodes.
- Data measurements from the Multi-Threaded Framework to the user.
- Configuration

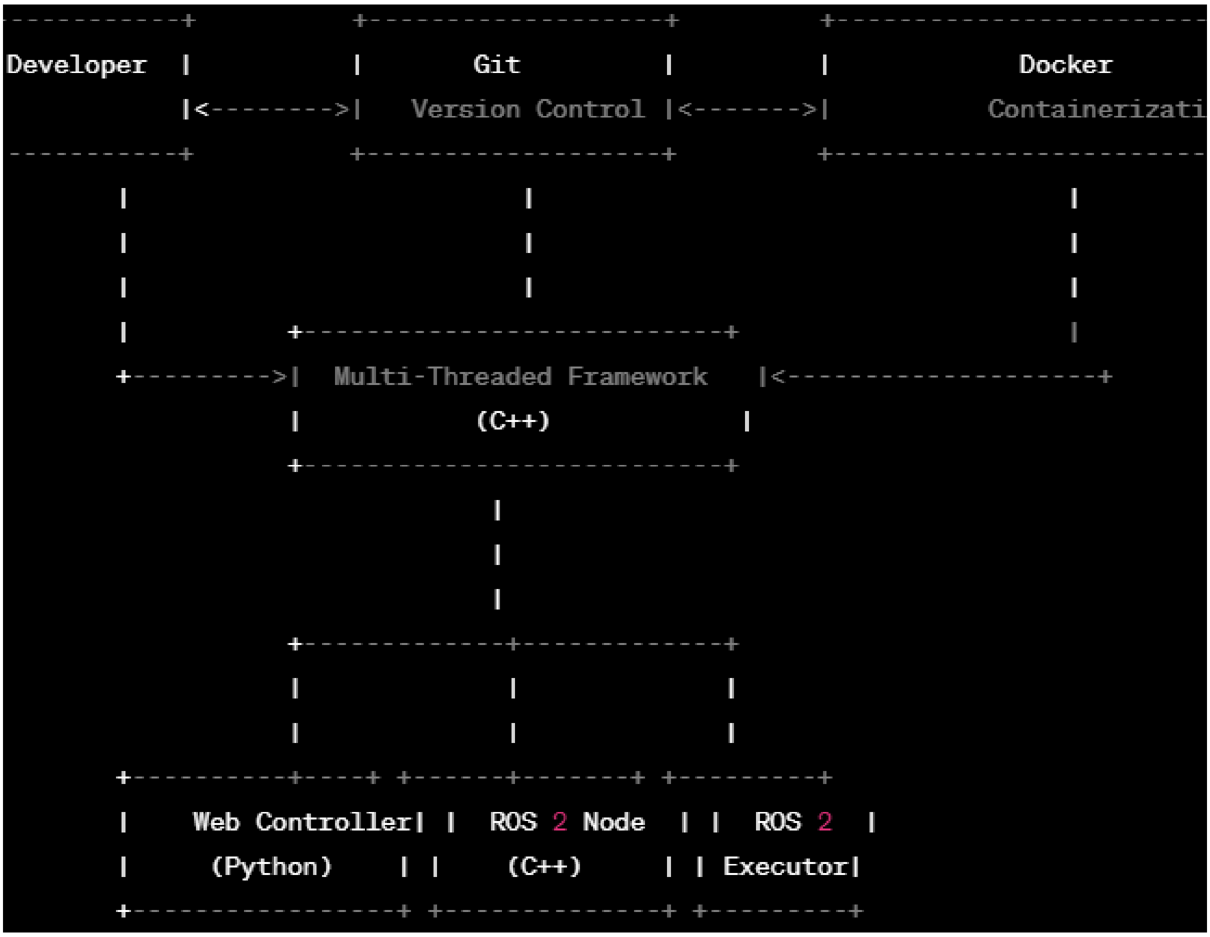
5. Please suggest a toolchain to implement the solution. Consider which tools we can reuse. For the components we need to develop, please suggest a programming language. Please use text (arrows and dots) to visualise the solution. Define data types transferred between components. Summarise data types transmitted between components. Add this information to the connections and interfaces between components.

To implement the solution, we can use a variety of tools, including:

- Robot Operating System 2 (ROS2) as the base infrastructure.
- C++ for programming the multi-threaded framework, ROS 2 Executor, and ROS 2 Nodes.
- Python for programming the web controller and user interface.
- Docker for containerizing the solution and ensuring reproducibility.
- Git for version control and collaboration.

We can reuse the ROS 2 infrastructure and packages for the ROS 2 Executor and ROS 2 Nodes components.

C4 Component Diagram for the toolchain to implement the solution:



```
-----------+          +-------------------+         +------------------------
Developer  |          |        Git        |         |           Docker
           |<-------->|   Version Control |<------->|        Containerizati
-----------+          +-------------------+         +------------------------
       |                        |                                  |
       |                        |                                  |
       |                        |                                  |
       |         +--------------------------+                      |
       +--------->|  Multi-Threaded Framework   |<---------------------+
                 |           (C++)            |
                 +--------------------------+
                            |
                            |
                            |
                 +------------+------------+
                 |            |            |
                 |            |            |
       +---------+----+ +-----+------+ +---------+
       |    Web Controller| |  ROS 2 Node  | |  ROS 2  |
       |     (Python)    | |   (C++)     | | Executor|
       +----------------+ +--------------+ +---------+
```

The diagram shows that the developer uses Git for version control and Docker for containerizing the solution. The multi-threaded framework component is programmed in C++. The web controller component is programmed in Python. The ROS 2 Node and ROS 2 Executor components are also programmed in C++. The data types transferred between components include:

- User inputs: Commands and specifications from the user to the framework.
- ROS 2 packages: Distributed packages of ROS 2 infrastructure.
- Multi-threaded code: Code parallelized by the framework.
- Data measurements: Data measuring latency, jitter, CPU caches usage, and memory fetches.
- Configuration output: Optimal executor type, node to process assignment, and allocator type configurations.

The connections and interfaces between components transmit the following data types:

- User inputs from the Web Controller to the Multi-Threaded Framework.
- ROS 2 packages from the Multi-Threaded Framework to the ROS 2 Executor and ROS 2 Nodes.
- Multi-threaded code from the Multi-Threaded Framework to the ROS 2 Nodes.
- Data measurements from the Multi-Threaded Framework to the user.

- Configuration output from the Multi-Threaded Framework to the user.