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.

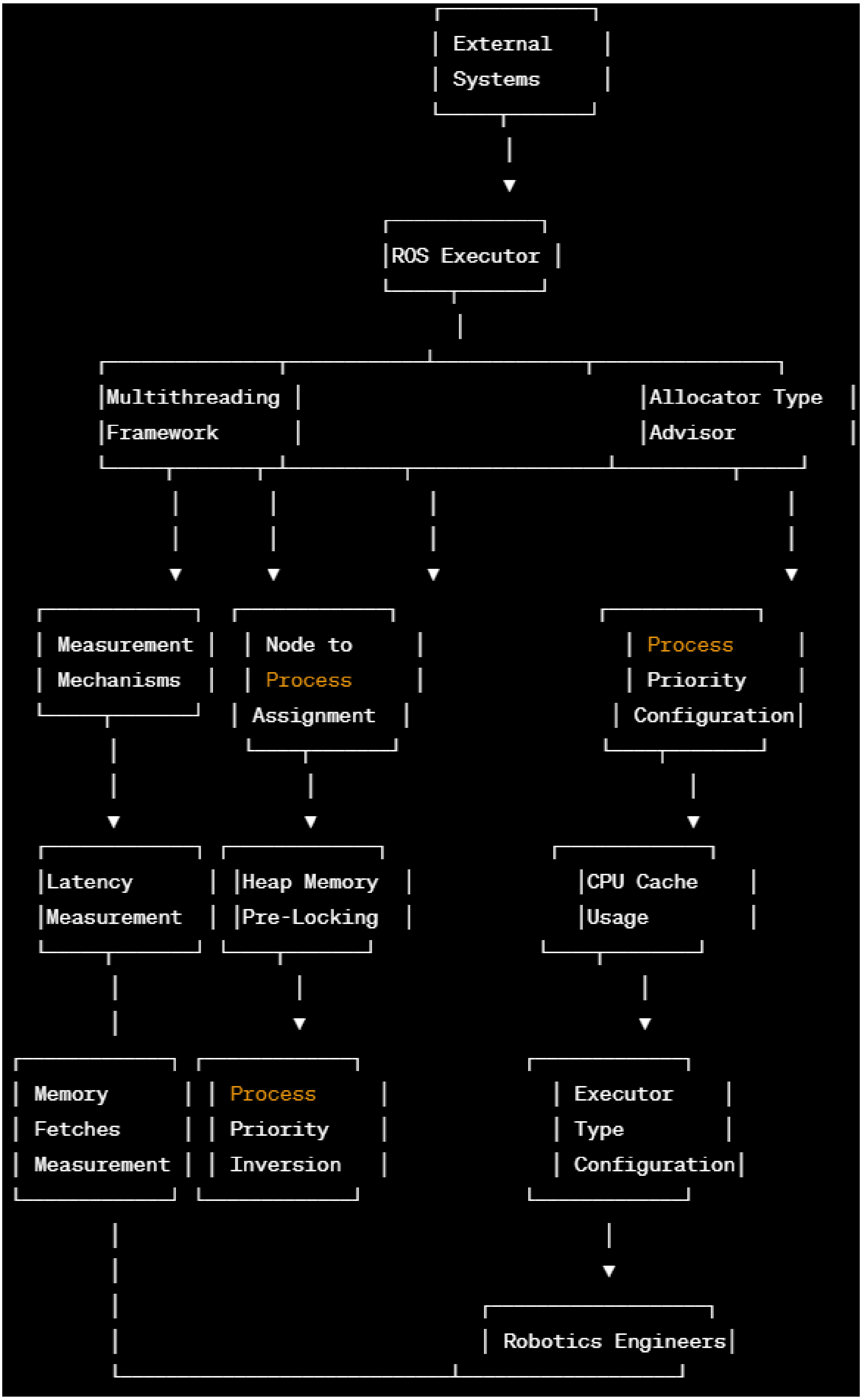
Problem: Robot Operating System 2 (ROS) is a complex infrastructure based on numerous distributed packages, which makes it hard for robotics engineers to utilize multi-cores properly to reduce execution time. Robotics engineers lack intuitive ways of parallelization, resulting in decreased efficiency and productivity.

Solution: A software system architecture solution is proposed to facilitate the development of multithreaded applications in ROS2, integrated with ROS executor. The solution includes pre-locking heap memory, process priority configuration, measurement of heap allocations, latency, jitter, CPU caches usage, and memory fetches. The framework will advise robotics engineers on the best performing executor type, node to process assignment, and allocator type. It will also output guidelines for robotics engineers and accept non-parallelized code and turn it into a parallel configuration. The solution aims to increase CPU utilization, decrease heap allocation, increase general performance, decrease average execution time, increase CPU cache utilization, decrease memory fetches, and increase productivity.

Use Cases: The proposed solution includes four use cases: Direct Inverse Kinematics, Detecting human faces and torso orientation, Kalman Filter, and Assessing execution time of real-time loop. These use cases aim to identify the way of doing image processing, sensor fusion, and prediction, and to measure real-time loop execution time for performance improvements.

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.

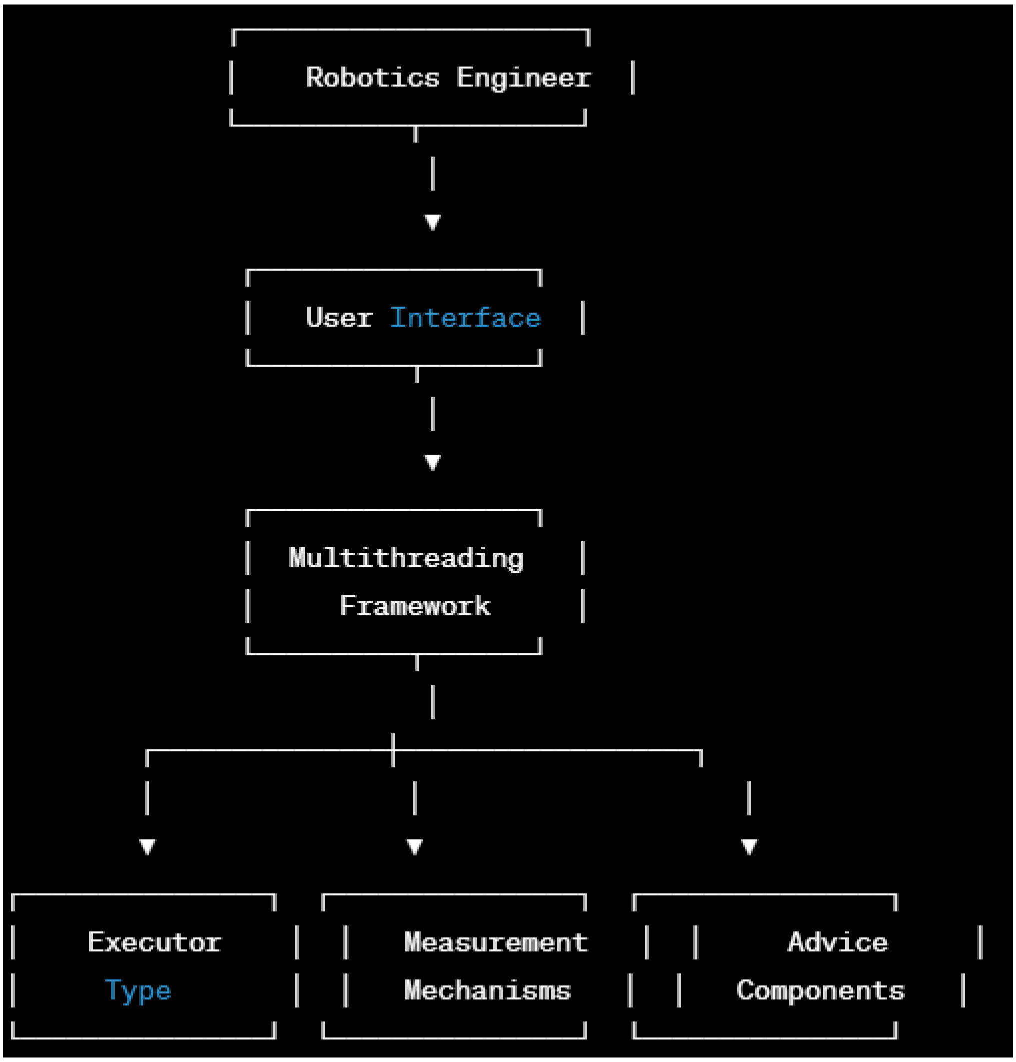
Please note that I cannot add colours and shapes as I am a text-based AI language model. Here's my attempt at creating a C4 system context diagram for the proposed solution:


External Systems
ROS Executor
Multithreading Framework
Allocator Type Advisor
Measurement Mechanisms
Node to Process Assignment
Process Priority Configuration
Latency Measurement
Heap Memory Pre-Locking
CPU Cache Usage
Memory Fetches Measurement
Process Priority Inversion
Executor Type Configuration
Robotics Engineers

The diagram shows the Multithreading Framework integrated with the ROS Executor and includes measurement mechanisms for latency, heap memory pre-locking, memory fetches, CPU cache usage, and process priority configuration. The framework includes the Allocator Type Advisor and the Node to Process Assignment components. The Executor Type Configuration component helps robotics engineers select the best-performing executor type. The diagram also includes external systems and stakeholders relevant to the system. The abstract data types transferred between components and stakeholders are not shown in the diagram.

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.

Here's my attempt at creating a C4 container solution diagram for the proposed solution:



The main stakeholder, Robotics Engineers, will use the solution via the User Interface component. They will input their ROS code to the Multithreading Framework, which will utilize the Executor Type Configuration and Node to Process Assignment components to determine the best-performing executor type and process assignment. The framework will then utilize the Measurement Mechanisms components to analyze the performance of the application and provide advice to the Robotics Engineers via the Advice Components.

Important quality properties include increased CPU utilization, decreased heap allocation, increased general performance, decreased average execution time, increased CPU cache utilization, decreased memory fetches, and increased productivity. To address these properties, the architectural style used will be a layered architecture, with the User Interface component at the top, the Multithreading Framework in the middle, and the Executor Type Configuration, Node to Process Assignment, Measurement Mechanisms, and Advice Components in the bottom layer.

The Multithreading Framework will utilize design patterns such as the Strategy pattern and the Factory pattern to determine the best-performing executor type and process assignment. The Measurement Mechanisms components will utilize the Observer pattern to measure and analyze performance metrics. Design tactics such as Precomputation and Stateful Component will be utilized to improve performance and facilitate memory pre-locking.

The data types transferred between components will include code input, executor type configurations, process assignments, measurement data, and advice.

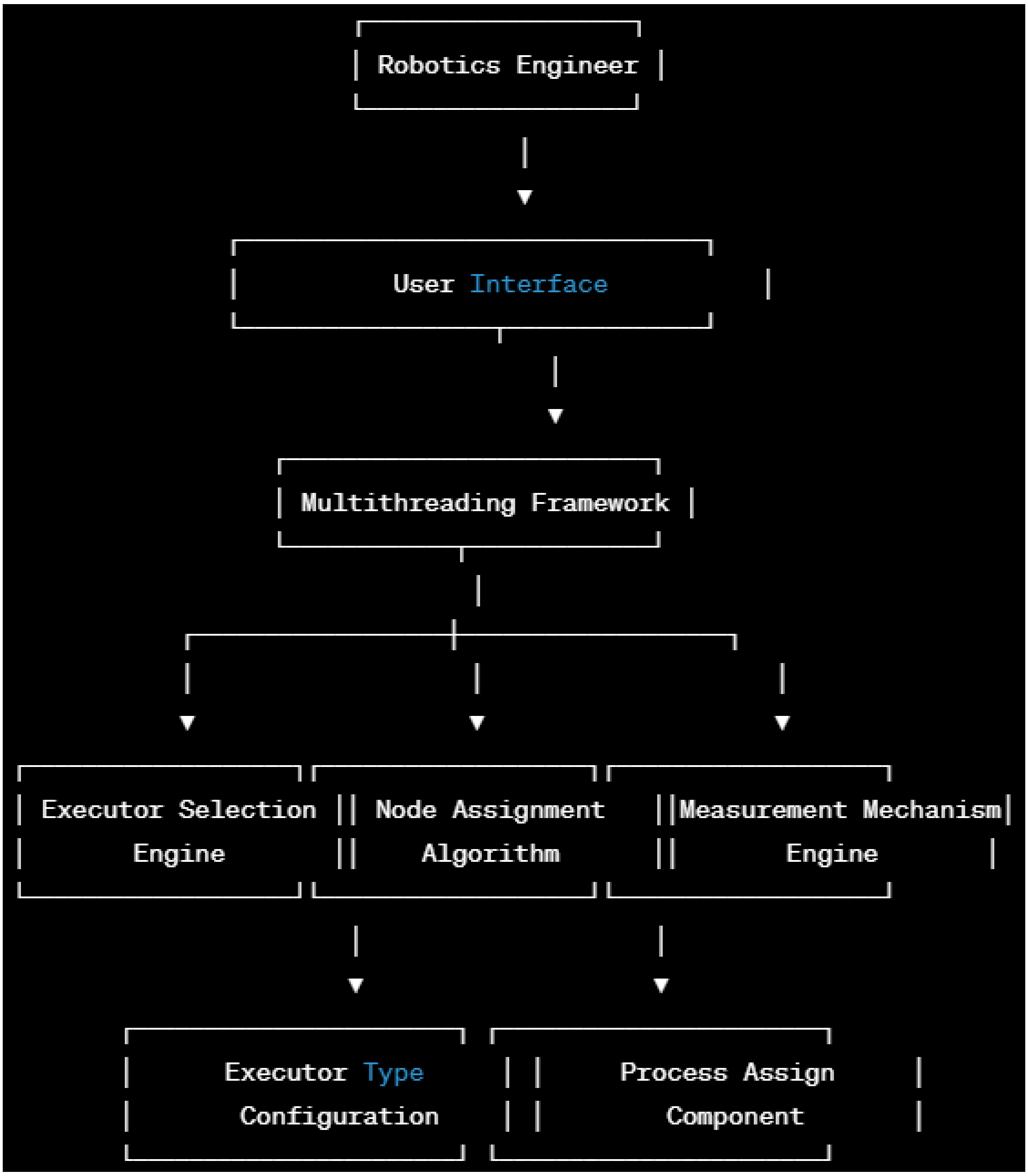
The interfaces between components include:

- User Interface: requires input code; provides advice
- Multithreading Framework: requires input code, executor type configurations, process assignments; provides advice
- Executor Type Configuration: requires executor type configurations; provides process assignments
- Node to Process Assignment: requires executor type configurations, process assignments; provides process assignments
- Measurement Mechanisms: requires input code; provides measurement data and advice
- Advice Components: requires measurement data; provides advice

Anti-patterns that should be avoided when implementing this solution include the Blob anti-pattern, which could occur if the Multithreading Framework becomes too large and unwieldy, and the God Object anti-pattern, which could occur if the Measurement Mechanisms become too complex and difficult to manage.

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.

Here's my attempt at creating a C4 component solution diagram for the proposed solution:



The main stakeholder, Robotics Engineers, will use the solution via the User Interface component. They will input their ROS code to the Multithreading Framework, which will utilize the Executor Selection Engine and Node Assignment Algorithm components to determine the best-performing executor type and process assignment. The framework will then utilize the Measurement Mechanism Engine to measure and analyze the performance of the application.

Important quality properties include increased CPU utilization, decreased heap allocation, increased general performance, decreased average execution time, increased CPU cache utilization, decreased memory fetches, and increased productivity. To address these properties, the architectural style used will be a component-based architecture, with the User Interface component at the top, the Multithreading Framework in the middle, and the Executor Selection Engine, Node Assignment Algorithm, Measurement Mechanism Engine, Executor Type Configuration, and Process Assign Component in the bottom layer.

The Executor Selection Engine component will utilize design patterns such as the Strategy pattern and the Factory pattern to determine the best-performing executor type. The Node Assignment Algorithm component will utilize the Factory pattern to assign nodes to processes. The Measurement Mechanism Engine component will utilize the Observer pattern to measure and analyze performance metrics. Design tactics such as Precomputation and Stateful Component will be utilized to improve performance and facilitate memory pre-locking.

The data types transferred between components will include code input, executor type configurations, process assignments, measurement data, and advice.

The interfaces between components include:

- User Interface: requires input code; provides advice
- Multithreading Framework: requires input code, executor type configurations, process assignments; provides advice
- Executor Selection Engine: requires input code; provides executor type configurations
- Node Assignment Algorithm: requires input code, executor type configurations; provides process assignments
- Measurement Mechanism Engine: requires input code; provides measurement data and advice
- Executor Type Configuration Component: requires executor type configurations; provides executor type configurations
- Process Assign Component: requires executor type configurations, process assignments; provides process assignments

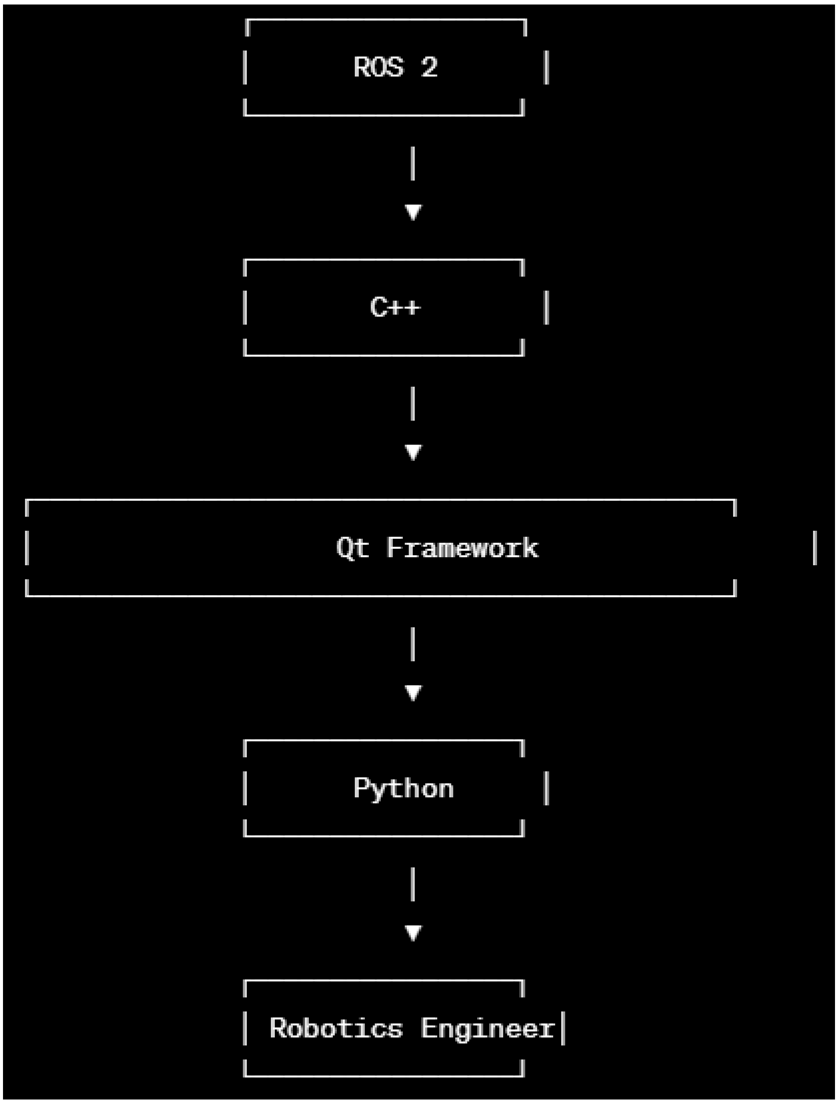
Anti-patterns that should be avoided when implementing this solution include the God Object anti-pattern, which could occur if the Measurement Mechanism Engine becomes too complex and difficult to manage, and the Functional Decomposition anti-pattern, which could occur

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 proposed solution, we can use the following toolchain:

- ROS 2: We can reuse ROS 2 as the underlying infrastructure for the Multithreading Framework and Executor Selection Engine components.
- C++: We can use C++ as the primary programming language for developing the Multithreading Framework, Executor Selection Engine, Node Assignment Algorithm, and Measurement Mechanism Engine components.
- Qt: We can use the Qt framework to develop the User Interface component, which can be integrated with the Multithreading Framework.
- Python: We can use Python as a scripting language for developing helper scripts to assist with the development process.

Here's a visualization of the toolchain:



The data types transferred between components include ROS messages for passing code input, executor type configurations, process assignments, and measurement data.

The interfaces between components include:

- User Interface: requires ROS messages for code input; provides ROS messages for advice
- Multithreading Framework: requires ROS messages for code input, executor type configurations, process assignments; provides ROS messages for advice
- Executor Selection Engine: requires ROS messages for code input; provides ROS messages for executor type configurations
- Node Assignment Algorithm: requires ROS messages for code input, executor type configurations; provides ROS messages for process assignments
- Measurement Mechanism Engine: requires ROS messages for code input; provides ROS messages for measurement data and advice
- Executor Type Configuration Component: requires ROS messages for executor type configurations; provides ROS messages for executor type configurations
- Process Assign Component: requires ROS messages for executor type configurations, process assignments; provides ROS messages for process assignments

Overall, this toolchain and language selection should provide a robust and efficient solution for the problem at hand, utilizing industry-standard tools and techniques.