# SUAVE
# *Adaptation Strategy*

**Fundamentals of Adaptive Software 2023**

**Group:** 3_2
**Members:** [redacted]
**Emails:** [redacted]
**VUnetIDs:** [redacted]
**Master Program and Track:** [redacted]

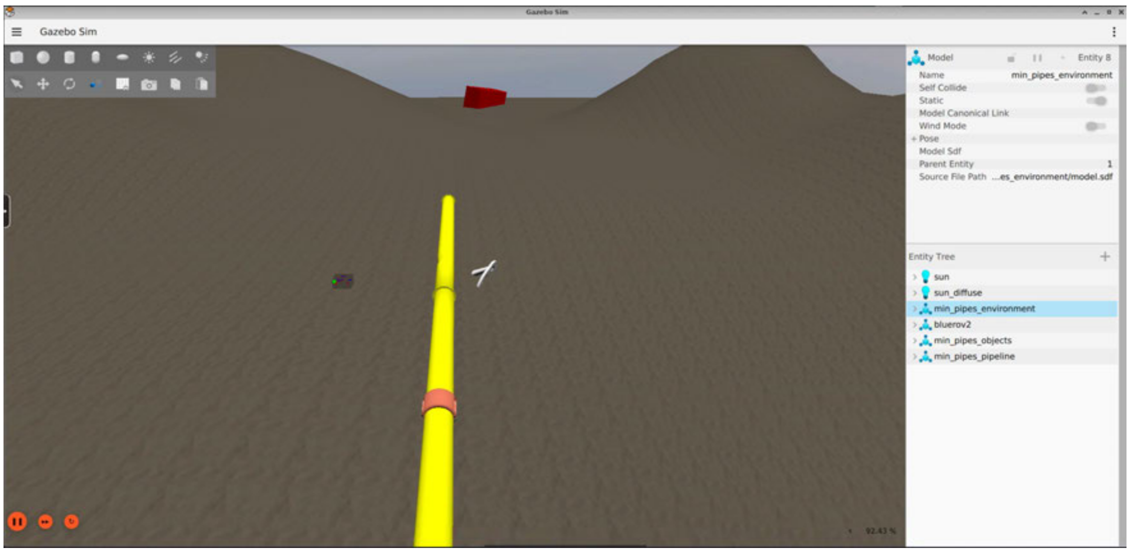
December 24, 2023

# Contents

## 1 Introduction

The Self-adaptive Underwater Autonomous Vehicle Exemplar (SUAVE) is a self-adaptive service-based system exemplar that performs inspections on pipelines that are underwater. This document proposes an extension of this project in order to make it more aligned with the actual situation.

Figure 1 is a visualization of the project in Gazebo. The yellow pipeline is the target of the vehicle for inspection and following. Figure 2 shows a more technical and architectural overview of the whole system.



Name min_pipes_environment
Self Collide
Static
Model Canonical Link
Wind Mode
+ Pose
Model Sdf
Parent Entity 1
Source File Path ...es_environment/model.sdf

Figure 1: The Gazebo Visualization of SUAVE

### 1.1 Managed System: SUAVE ROS2 Runtime

The managed system in the original SUAVE paper [1] refers to the operational components of the autonomous underwater vehicle (AUV), which includes hardware like sensors, actuators, and thrusters. This system is responsible for executing specific tasks such as navigating underwater environments, inspecting pipelines, and gathering data. It functions based on predefined paths and relies on environmental input from its sensors.

Internally, the managed system communicates through the ROS2 (Robot Operating System 2) network, which provides low-latency communication between all embedded components within the system. Within SUAVE, ROS2 facilitates communication between different components of the AUV, enabling data exchange and integration of various functionalities. It provides a robust and flexible platform for developing and implementing

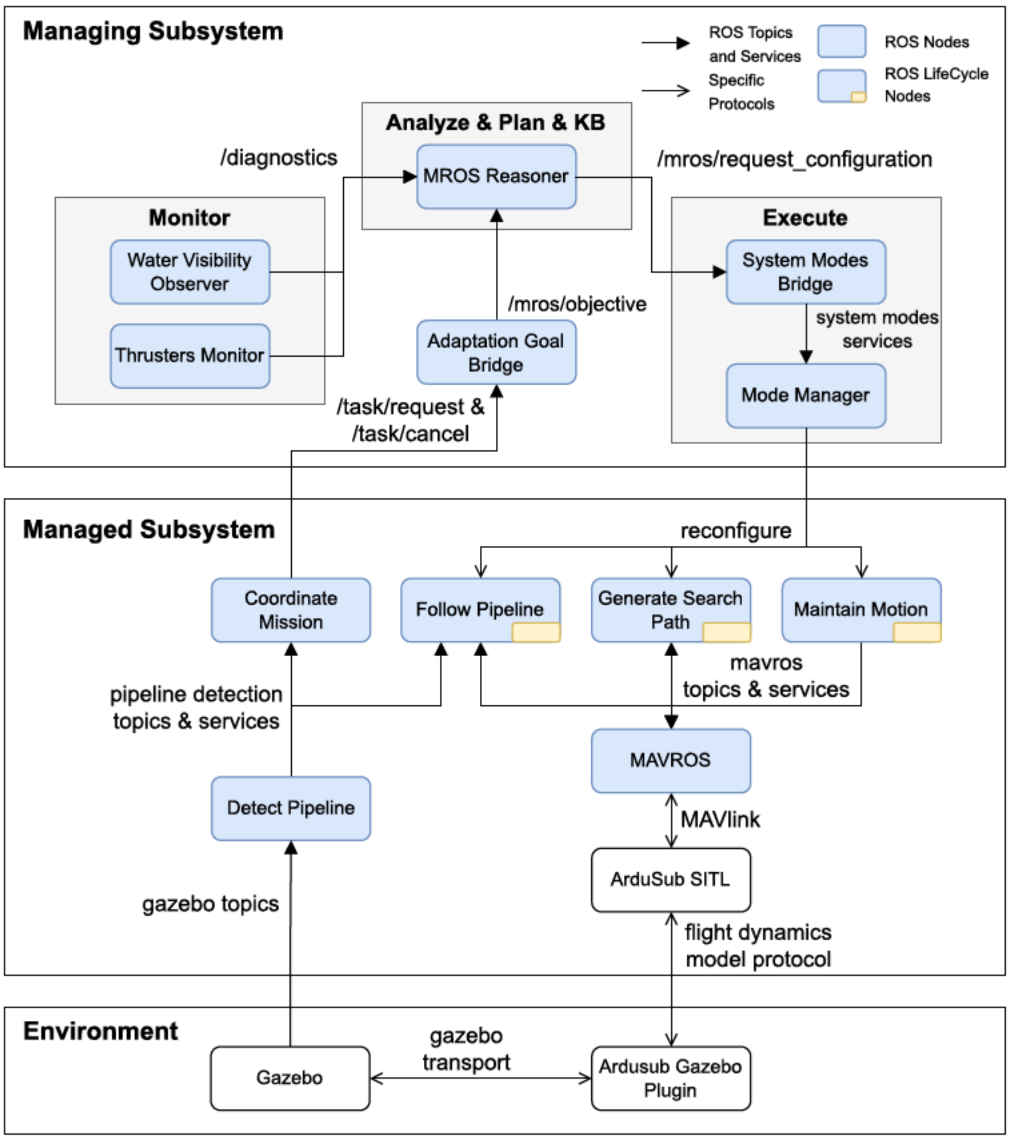


Figure 2: Architecture of the Original SUAVE Project [1]

the managing system's adaptive capabilities, such as monitoring the AUV's state, handling component failures, and dynamically adjusting its behavior in response to environmental changes, thus playing a crucial role in enhancing the AUV's operational efficiency and reliability.

From the perspective of this assignment, the managed system is not changed much from the original SUAVE.

## 1.2 Managing System: UPISAS

The new managing subsystem introduced in this assignment, named UPISAS (Unified Python Interface for Self-adaptive Systems), will mostly replace the original managing system found in the SUAVE project.

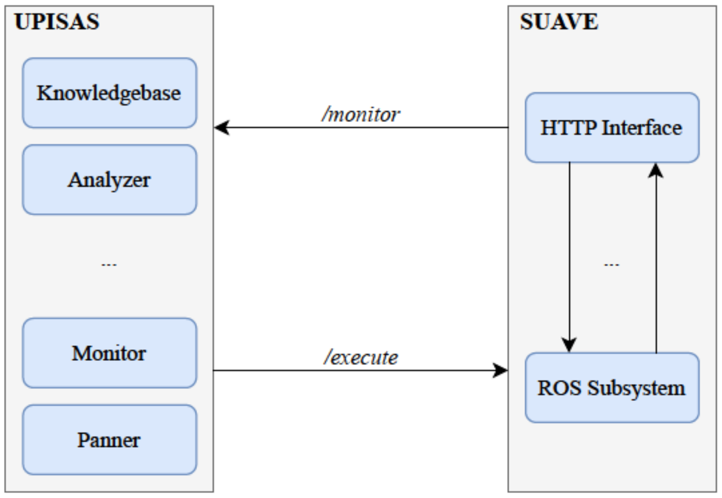


Figure 3: High-level Overview between SUAVE and UPISAS

The new managing system will be decouple from the original SUAVE, and will be spawned separately. UPISAS communicates with SUAVE through an HTTP interface with optional JSON body. Figure 3 provides a simplified high-level overview of the whole system.

# 2 Analysis of Uncertainties

In this context of SUAVE project, we propose some adaptive measures to mitigate the impact of the following uncertainties: the failure of on-board sensors, which potentially includes:

1. Extreme change of the water visibility (incl. the erroneous reporting/failure of the water visibility sensor)
2. The general failure of AUV's thrusters

In the following sections (subsection 2.1 and subsection 2.2), the two uncertainties will be analyzed in detail with formal descriptions, respectively.

## 2.1 Uncertainty #1: Water Visibility

In this report, a variant of the uncertainty water visibility is introduced, which encompasses two main aspects.

First, there's the change in water visibility, which can vary due to environmental factors (such as the altitude of the vehicle) and directly impact the autonomous underwater vehicle's (AUV) navigation and task performance.

Second, the potential failure of the water visibility sensor itself is considered, such as erroneously reporting a visibility of zero. This sensor malfunction could lead to incorrect data interpretation, affecting the AUV's decision-making and operational efficiency. These uncertainties necessitate robust adaptive mechanisms within the AUV's system to ensure reliable performance under varying underwater conditions.

In addition, remote debugging underwater vehicles cost significant money and time. Mitigating uncertainties in a self-adaptive way can minimize unnecessary expenses and increase the overall workflow efficiency.

Table 1 provides a detailed description of the uncertainty.

## 2.2 Uncertainty #2: Failure of Thrusters

The second uncertainty present in this report is the failure of thrusters, which can be caused by various factors like unexpected obstacles or interactions with large ocean animals, such as sharks, or variable underwater terrains, such as rocks.

Addressing this uncertainty is crucial because thruster failure can severely impact the autonomous underwater vehicle's (AUV) ability to navigate and complete its mission. An effective response system is essential to detect and compensate for such failures, ensuring the AUV can adapt its operational strategies to maintain functionality and mission integrity in these challenging scenarios.

Table 2 provides a detailed description of the uncertainty.

| Identifier | Description |
|---|---|
| Name | Water Visibility |
| Classification | Execution (Runtime) |
| Context | Change of water visibility can happen due to change of underwater altitude. Failure of water visibility sensor can happen due to external forces being applied to the vehicle. |
| Impact | Low water visibility can lead to inefficient or blind underwater vision, further leading to failure of tasks. |
| Degree of severity | Medium - High |
| Sample illustration | When the underwater sand or rock get too dense, the water visibility will drop drastically. |
| Evaluation | In a self-adaptive system, the response to water visibility changes involves dynamic adaptation, such as modifying navigation paths or increase operational altitude. In contrast, a non-SAS system would continue operating based on pre-set parameters, regardless of visibility changes, leading to inefficient task performance. |
| Also known as | Variable environment, Uncontrollable environment |

Table 1: Description of Uncertainty: Water Visibility

| **Identifier** | **Description** |
|---|---|
| Name | Failure of Thruster |
| Classification | Execution (Runtime) |
| Context | Failure of the thruster can result from the failure of its built-in feedback sensor, or external obstacles or forces, such as a shark or rocks. |
| Impact | The general failure of thrusters can lead to reduced ability or inability to navigate in underwater environment, which eventually has negative impact on the completion of tasks. |
| Degree of severity | Medium |
| Sample illustration | When the vehicle hits a huge rock, or when small rocks accidentally get into the thruster, it could cause the thruster to slow down or stop altogether. |
| Evaluation | In a self-adaptive system, the response to thruster failures involves real-time monitoring and adaptation. When a failure is detected, the system can activate contingency protocols, like redistributing power to functional thrusters, to maintain stability and navigation. Conversely, a non-SAS system typically lacks these adaptive capabilities. In the event of a thruster failure, it would continue with its predefined operations, potentially leading to inaccurate underwater navigation or inability to maintain control, as it cannot autonomously adjust to the compromised functionality of the damaged thruster. |
| Also known as | Variable environment, Uncontrollable environment |

Table 2: Description of Uncertainty: Failure of Thrusters

## 3 Requirements

In conventional system requirement specifications, modal verbs from natural language such as *SHALL* or *MUST* are commonly used to delineate the essential functionalities that a system is expected to consistently deliver, known as traditional requirements. In this context, the identifiers TR1, TR2, and TR3 are used to denote these traditional requirements, as depicted in Table 3.

| **Requirement** | **Description** |
|---|---|
| Recover from erroneous visibility reporting | TR1: The AUV *MUST* recover to a working state (normal or degraded) when an erroneous visibility reporting is detected. |
| Recover from a thruster failure | TR2: The AUV *MUST* recover to a working state (normal or degraded) when a thruster failure is detected. |

Table 3: Traditional Requirements

For self-adaptive systems, strictly adhering to defined requirements can be challenging due to inherent uncertainties. Therefore, their requirements specifications necessitate a structure different from traditional ones. To achieve this, a specialized language called *RELAX* [2] is used, specifically designed for self-adaptive systems. In this context, the requirements initially presented in Table 3 are modified in Table 4 using *RELAX*, with the new identifiers RR1, RR2, and RR3 correlating to TR1, TR2, and TR3, respectively.

| **Requirement** | **Description** |
|---|---|
| Recover from erroneous visibility reporting | RR1: The AUV *MUST* recover to a working state (normal or degraded) AS EARLY AS POSSIBLE AFTER an erroneous visibility reporting is detected. *EVENTUALLY*, the AUV will cancel or finish the current task. *MONITOR*: water visibility reporting. *ENVIRONMENT*: the current task and altitude. |
| Recover from a thruster failure | RR2: The AUV *MUST* recover to a working state (normal or degraded) AS EARLY AS POSSIBLE AFTER a thruster failure is detected. *EVENTUALLY*, the AUV will cancel or finish the current task. *MONITOR*: number of failed thrusters. *ENVIRONMENT*: the current task. |

Table 4: Relaxed Requirements

## 4 Analysis of Potential Solutions

This section outlines our general direction for implementing adaptation strategies for the identified uncertainties. We will explore the various solutions we've identified and explain our reasons for choosing specific methods for these adaptation strategies.

For the uncertainty of water visibility (subsection 2.1), Table 5 will explain our design choice.

| | |
|---|---|
| **Issue** | Which solution to use to implement the adaptation strategies for the identified uncertainties? |
| **Ranking Criteria** | To ensure that the AUV can continue with its operate smoothly underwater under different kinds of uncertainties, it was required to analyze different solutions that could potentially achieve the adaptation goals. The issue at hand is that uncertainties that can be predicted are easy to tackle, but the goal of the solution is to adapt to uncertainties that cannot be predicted during development, and training would be required to overcome them. |

| | |
|---|---|
| **Option #1: Finite State Machine (FSM)** | **Description**: A finite state machine describes systems with a finite number of distinct states and transitions between them. FSMs operate by transitioning between states based on inputs or events. This option was considered due to the fact that the AUV has a number of distinct possible states, and during adaptation, the behavior of the self-adaptive AUV is sequential.<br><br>**Status**: *Accepted*<br><br>**Evaluation**: The implementation of the finite state machine was easy as compared to the other available options, but also because of the fact that there are many libraries available that support FSMs in Python. Furthermore, within the scope of this project, the FSM-based implementation required lesser learning and computation power because of the deterministic nature and low overhead. Lastly, it is known that there are only two 'inputs' from the AUV to the managing system: the water visibility and the thruster status. The only drawback of implementing an FSM is that if implemented statically, the adaptation space ca be limited.<br><br>**Rationale of Decision**: Even with a limited adaptation space, this option has still been selected because of the fact that the number of positives that come out from this implementation trump the negatives. The ease of implementation and lesser learning and computation is why this option is feasible. |

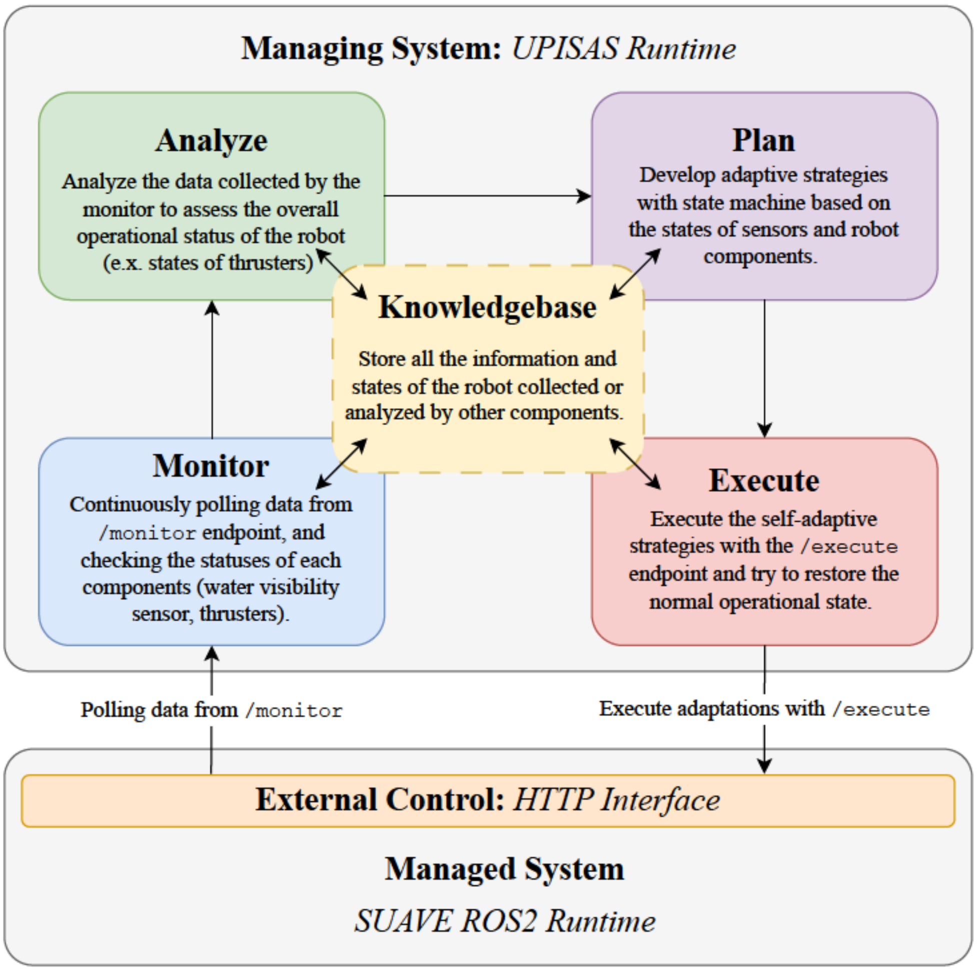
| | |
|---|---|
| **Option #2: Regression Model** | **Description**: A regression model is a statistical method that quantifies the relationship between a dependent variable and one or more independent variables to make predictions or infer patterns in data. This option was considered because regression models can learn and predict how changes in input variables relate to desired adaptations, allowing the AUV to make data-driven and continuous adjustments.<br><br>**Status**: *Rejected*<br><br>**Evaluation**: The only positive outcome of implementing a regression model is that the adaptation space is bigger as compared to that of the finite state machine. But on the other hand, the regression model is much harder to implement, demands more compute power and performance, and it also requires various kinds of inputs to achieve good effects, which is not provided by the AUV. Finally, to build a regression model, the best option would be to have a hybrid of offline and online learning, which, in comparison to an FSM, would be more time & resource expensive.<br><br>**Rationale of Decision**: Due to the fact that there are many negative impacts of implementing a regression model as compared to an FSM, it can be justified that it is not the best option to implement the adaptation strategies for the AUV. |

Table 5: Analysis and Decision Rationales for Potential Solutions

After thorough discussion and careful analysis, we settled on using Finite State Machine as a basis for the implementation of our adaptive strategy.

## 5 Design of the Proposed Adaptation Strategy

The following diagram (Figure 4) shows the high-level overview of our proposed adaptation strategy (the MAPE-K loop [3]).



**Managing System:** *UPISAS Runtime*
Analyze
Analyze the data collected by the monitor to assess the overall operational status of the robot (e.x. states of thrusters)
Plan
Develop adaptive strategies with state machine based on the states of sensors and robot components.
Knowledgebase
Store all the information and states of the robot collected or analyzed by other components.
Monitor
Continuously polling data from /monitor endpoint, and checking the statuses of each components (water visibility sensor, thrusters).
Execute
Execute the self-adaptive strategies with the /execute endpoint and try to restore the normal operational state.
Polling data from /monitor
Execute adaptations with /execute
**External Control:** *HTTP Interface*
Managed System
*SUAVE ROS2 Runtime*

Figure 4: The MAPE-K Model Diagram

Based on the analysis in section 4, a finite state machine is used for the analyze and plan part of the loop.

### 5.1 Design of Finite State Machine

In our proposed adaptation strategy, a finite state machine is used to perform the analysis and planning of the adaptation strategy. The design is largely inspired by RINGA [4].

The possible states within the state machine are:

- Idle
- Ascending (*Asc.*): The AUV is increasing altitude
- Descending (*Desc.*): The AUV is decreasing altitude
- Executing Task (*Exec.*): The AUV is executing a certain task
- Degraded Operation (*Degraded*): The normal operation of the AUV is degraded
- Returning (*Return*)

For the conditions/events of the state change, Table 6 provides a presentation of the finite state machine with a State/Event table. Figure 5 provides a visualization of the internal logic of the state machine.

| Input \ Current | *Idle* | *Desc.* | *Asc.* | *Exec.* | *Degraded* | *Return* |
|---|---|---|---|---|---|---|
| $Visibility > N$ | Desc. | - | - | Desc. | - | - |
| $Visibility = N$ | - | Exec. | - | - | - | - |
| $Visibility < N$ | - | - | - | Asc. | - | - |
| *Erroneous Water Visibility Reporting Detected* | - | - | - | Degraded | - | - |
| *Thruster Failure Detected* | - | - | - | Degraded | - | - |
| *Thruster Failures* $> M$ | - | - | - | - | Return | - |
| *SensorError* $> S$ *seconds* | - | - | - | - | Return | - |

Table 6: State-transition Table

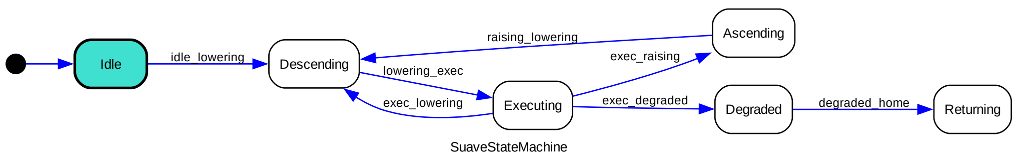


Figure 5: Visualized Diagram of the State Machine

## 5.2 Expected Effects

The overall expected effects of the strategy is to ensure the normal operation of the AUV and minimizing any potential downtime when uncertainties happen, and if anything catastrophic happens (e.g. losing more than 1 thrusters), the AUV protects itself by entering a degraded operation mode and execute the sequence of returning.

Specifically, the strategy aims to achieve the following goals:

- Recover from thruster failures either:
  1. by resolving the individual failure of the thruster, or
  2. by entering degraded operation mode
- Recover from water visibility sensor failures or from environments with extremely low water visibility:
  1. by trying to recover the working state of the sensor itself
  2. by trying to ascend to a higher altitude
  3. by entering degraded operation mode

From the perspective of the goal execution, the effect of our adaptation strategy would be reflected by *longer pipeline distance inspected on average compared to the original SUAVE* on a single run of the underwater exploration task.

It is worth noting that our design of adaptation strategy can also bring some downsides, specifically due to the introduction of a degraded operation mode. In this mode, the AUV will evaluate the current state of the robot, terminate the current task and possibly initiating returning procedure. This might lead to the failure of the overall pipeline inspection task, but it's a necessary trade-off to ensure minimal loss.

# 6 Implementation

For code repository, all links are listed in Appendix B. For documentations on development and deployment, all related links are listed in Appendix C.

Overall, the implementation is split into two parts, which are the part in SUAVE (subsection 6.2), and the part in UPISAS (subsection 6.3), respectively.

## 6.1 Technology Stacks

Here are a list of technology stacks used for the implementation of the proposed adaptation:

- OS: Ubuntu 22.04.3 LTS
- Libraries:
  - ROS 2 Humble: `https://docs.ros.org/en/humble/index.html`
  - Flask: `https://github.com/pallets/flask`
  - Python StateMachine: `https://github.com/fgmacedo/python-statemachine`
- Software:
  - Docker Community Edition: `https://github.com/docker/cli`
  - Docker Compose: `https://github.com/docker/compose`
  - Kasm Workspace: `https://github.com/kasmtech/workspaces-images`

## 6.2 External Control HTTP Interface in SUAVE

The HTTP server inside the External Control node in the managed system is responsible for reporting real-time monitoring data to and receiving external adaptation commands from UPISAS. Figure 6 provides an overview of the architecture of the external control we implemented in SUAVE.

For the specifications of the API design, please refer to Table 7 and Table 8.

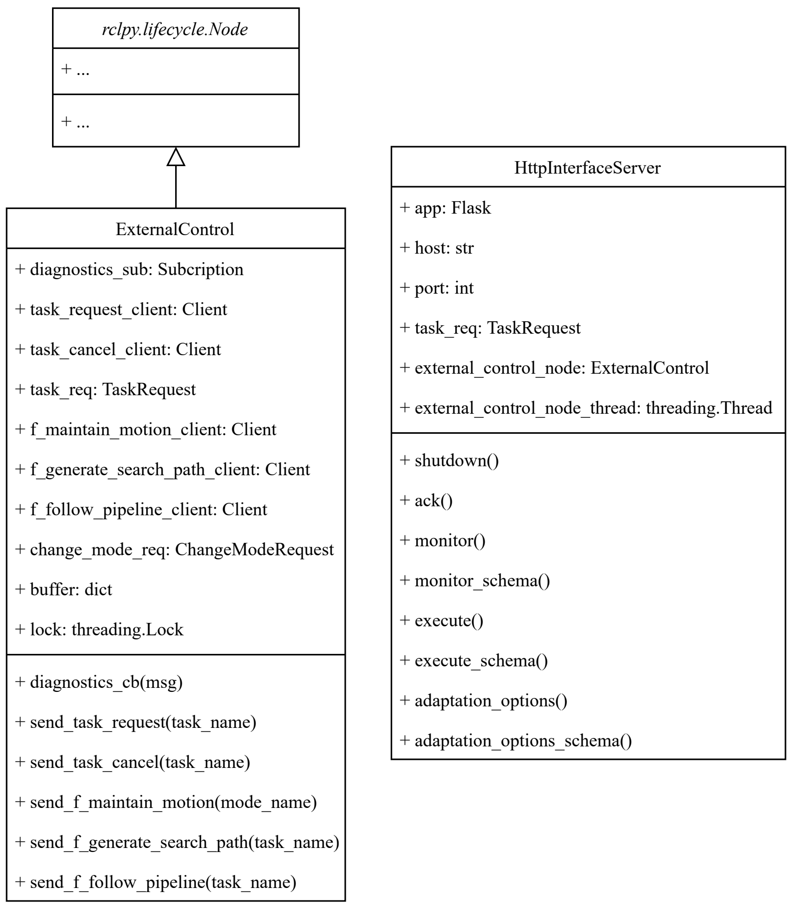

rclpy.lifecycle.Node
+ ...
+ ...
ExternalControl
+ diagnostics_sub: Subcription
+ task_request_client: Client
+ task_cancel_client: Client
+ task_req: TaskRequest
+ f_maintain_motion_client: Client
+ f_generate_search_path_client: Client
+ f_follow_pipeline_client: Client
+ change_mode_req: ChangeModeRequest
+ buffer: dict
+ lock: threading.Lock
+ diagnostics_cb(msg)
+ send_task_request(task_name)
+ send_task_cancel(task_name)
+ send_f_maintain_motion(mode_name)
+ send_f_generate_search_path(task_name)
+ send_f_follow_pipeline(task_name)
HttpInterfaceServer
+ app: Flask
+ host: str
+ port: int
+ task_req: TaskRequest
+ external_control_node: ExternalControl
+ external_control_node_thread: threading.Thread
+ shutdown()
+ ack()
+ monitor()
+ monitor_schema()
+ execute()
+ execute_schema()
+ adaptation_options()
+ adaptation_options_schema()

Figure 6: Class Diagram of External Control in SUAVE

| Field | Description |
|---|---|
| Description | This endpoint is used to monitor the diagnostic information for this exemplar. Specifically, the monitorable information exposed by the SUAVE system are:<br>• Status of the six thrusters (true or false)<br>• Visibility of the water (a float value) |
| Endpoint | `/monitor` |
| Allowed HTTP Method | `GET` |
| Sample Response Body | <pre>{<br>  "thrusters": {<br>    "c_thruster_1": true,<br>    "c_thruster_2": true,<br>    "c_thruster_3": true,<br>    "c_thruster_4": true,<br>    "c_thruster_5": true,<br>    "c_thruster_6": true<br>  },<br>  "water_visibility": 1.4887287570312693<br>}</pre> |
| Sample Return Status | `200 OK` |

Table 7: Specification for the */monitor* API

| **Field** | **Description** |
|---|---|
| Description | This endpoint is used to execute various adaptations in the system. Meanwhile, the corresponding options and type of adaptation are specified in the request body. |
| Endpoint | `/execute` |
| Allowed HTTP Method | `PUT` |
| Sample Request Body | `{"adaptation": "/task/cancel",`<br>`↪ "option": "search_pipeline"}` |
| Sample Return Status | `200 OK` |

Table 8: Specification for the */execute* API

## 6.3 Finite State Machine in UPISAS

The state machine in UPISAS provides the self-adaptive system with the ability to adapt to various external changes. Figure 7 provides an architecture overview of the implementation in UPISAS.

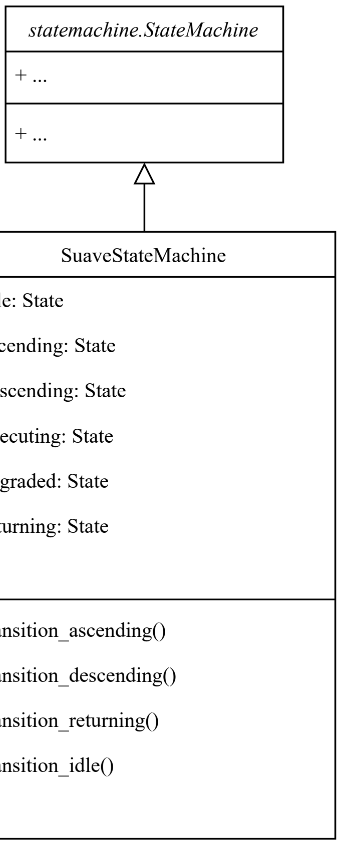


Figure 7: Class Diagram of Finite State Machine in UPISAS

By using the Python StateMachine library, it allowed easy and managed transitions between states, with associated entry and exit actions.

For a more detailed showcase of its mechanism, please refer to subsection 5.1.

## 6.4 Deployment Architecture

In our deployment architecture, both SUAVE and UPISAS operate within their respective Docker containers, ensuring an isolated and consistent environment for each component. Docker-compose orchestrates the deployment, managing the containers as a unified application. Figure 8 illustrates a typical deployment architecture of our system.

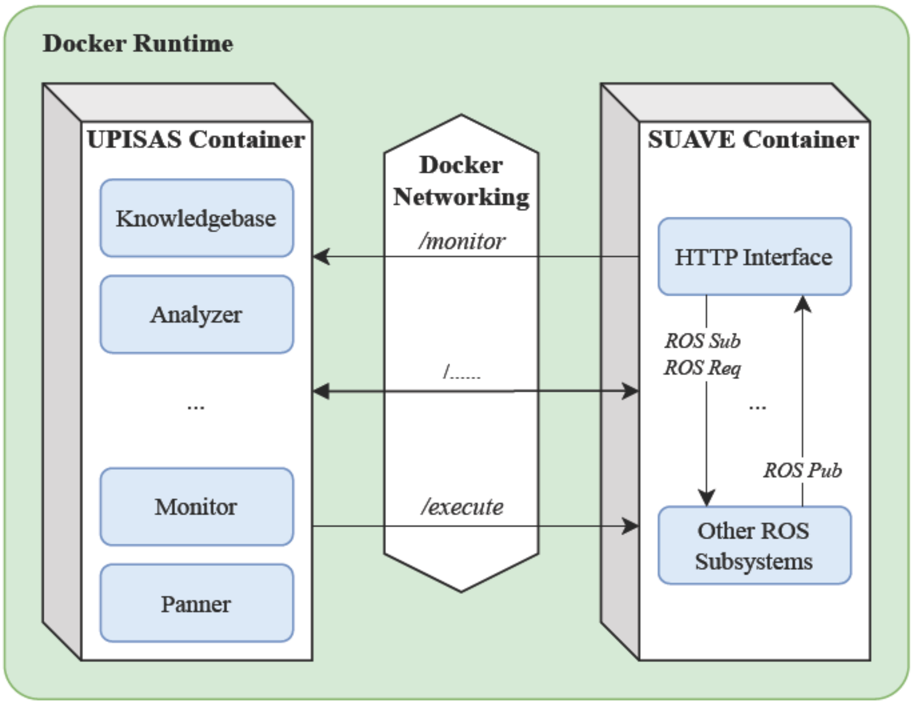


Figure 8: Deployment Architecture

This architecture allows for robust scalability and flexibility. By containerizing the components, we can deploy the system across various environments with ease, from development machines to production servers, without concerns about dependencies or environment-specific configurations, which eventually achieve platform-agnostic deployment.

## 6.5 Alternative Implementation Paths

Two alternatives are given for other implementation paths, which will be detailed in the following two sections, respectively.

### 6.5.1 Integrating UPISAS into ROS Runtime

One of the design concerns of the architecture of UPISAS and SUAVE is feasibility on real-world scenarios. Further discussions on this can be found at subsection 8.2.

To handle such concern, one of the alternative implementation paths is converting UPISAS into an internal node of the SUAVE system. Figure 9 demonstrates the high-level system architecture of the alternative implementation path.

However, due to the time constraint and alignment of the goals of the project, such alternative implementation path was not taken into further consideration. Nevertheless, it

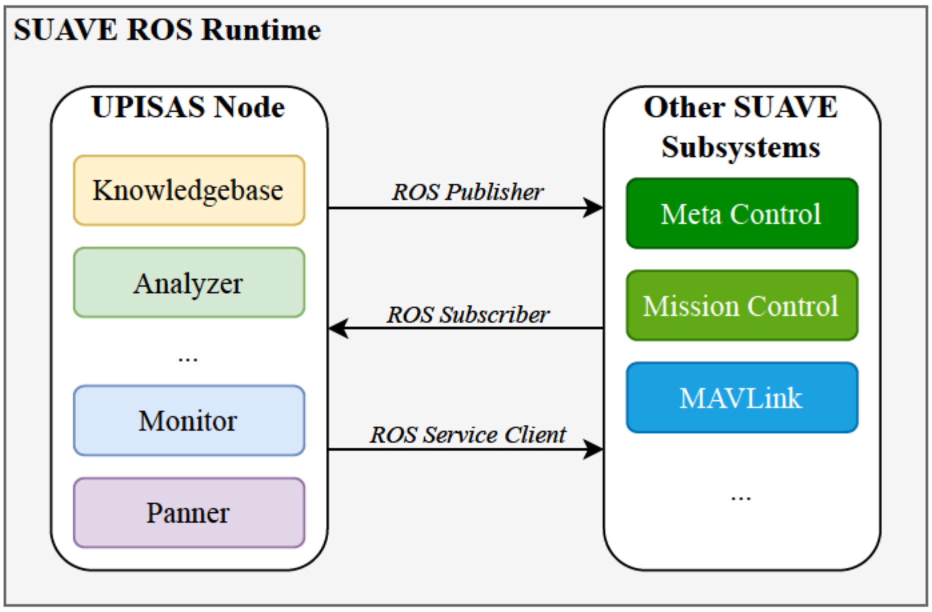


Figure 9: High-level System Architecture of the Alternative Implementation

is recommended to use such implementation path if the project were to be used in real-world robotic deployment and experiments.

### 6.5.2 Regression Model

Another alternative implementation path to the finite state machine for managing self-adaptive behavior could be a using *regression model*. This statistical approach would be best suited for use cases where the system's responses to varying conditions can be captured through quantifiable metrics, such as adapting the speed and energy consumption of the AUV based on the varying resistance it encounters at different depths and current strengths. A regression model could learn from historical data to predict the optimal operational parameters, offering a potentially smoother and more fine-grained adaptation process.

However, the drawbacks of implementing a regression model include the need for a substantial amount of historical data to train the model, which requires not only more data, but also more capable on-board hardware, which is not ideal in the context of AUVs.

# 7 Showcase and Evaluation

As mentioned in subsection 5.2, the efficacy of the proposed state machine solution within the SUAVE project can be gauged by a key performance metric: the distance of the pipeline inspected (measured in meters) on a single run. This metric is pivotal because it directly correlates with the primary objective of an AUV – to inspect and monitor underwater infrastructure efficiently.

Choosing pipeline distance as an evaluation metric is logical for several reasons. Firstly, it provides a quantifiable measure of the AUV's operational capability, reflecting both its endurance and effectiveness in fulfilling its task. Secondly, this metric is indicative of the self-adaptive system's success in maintaining the AUV's operational state despite environmental uncertainties or hardware malfunctions. An increase in the inspected distance would suggest that the state machine effectively manages transitions between states such as *Ascending* or *Degraded Operation*, ensuring the AUV can continue its mission with minimal interruption. *All the runs in the following sections are time-constraint runs.*

## 7.1 Baseline

When evaluating the effectiveness of the proposed state machine solution, it is crucial to establish a baseline for comparison. A baseline lies in its role as a standard against which the adaptive capabilities of the state machine can be measured. In the ideal conditions of the baseline scenario, the AUV would navigate and inspect the pipeline without encountering any of the uncertainties. This provides a benchmark of the maximum potential pipeline distance that can be inspected in a single run, under the assumption of perfect conditions. The baseline is conducted for 5 times to prevent jitters.

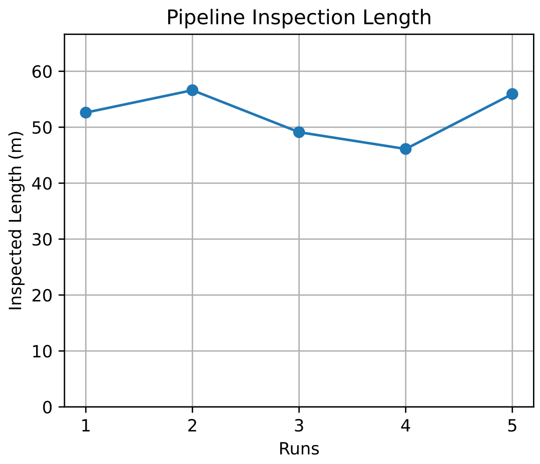


Figure 10: Baseline Readings of SUAVE

Figure 10 shows the baseline capability of the SUAVE system, with an average of 52.077 meters.

## 7.2 With the Failure of Water Visibility Sensor

This simulates the scenario with the potentially recoverable failure of the water visibility sensor.

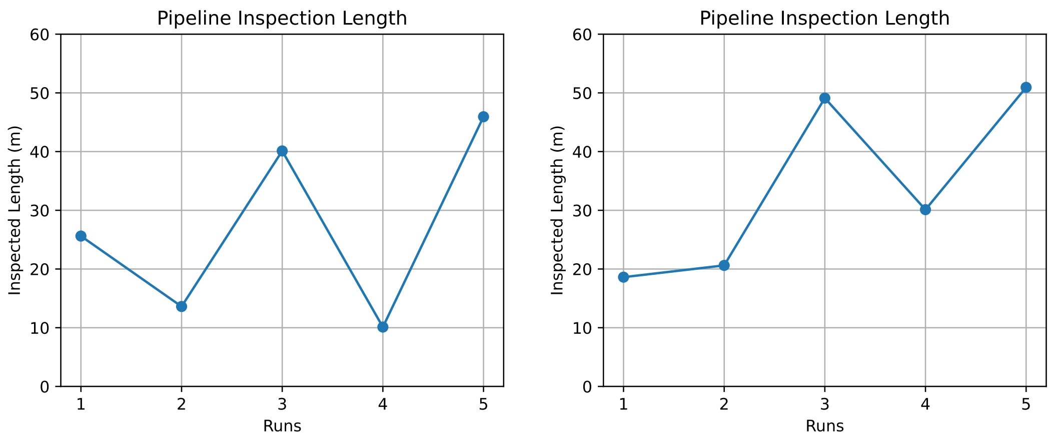


Figure 11: Pipeline Inspection Length: Left - Original SUAVE; Right - FSM SUAVE

This yields an average of 27.682 meters for the Original SUAVE, and 33.877 meters for the FSM SUAVE. The sub-optimal result from the FSM SUAVE on the first and fourth run is due to the inability to recover the sensor to a working state, thus aborting the task altogether according to the FSM. However, when compared to the original SUAVE, the FSM SUAVE still got higher average performance and higher stability among the five evaluation runs.

# 8 Discussion and Reflection

This section focuses on reflecting upon the project, highlighting key learnings and the challenges encountered throughout its execution.

## 8.1 Takeaways of the Project

Working on this self-adaptive systems project has been immensely beneficial for our team, each member bringing unique skills that enriched our learning and development. Our achievements aligned with the course goals:

- Gaining hands-on experience with a self-adaptive system in a project-oriented context.
- Identifying uncertainties and documenting them using a specific template.
- Designing and implementing new adaptations following the MAPE-K model in the SUAVE project.
- Get hands-on experience with ROS2 and various other robotics technology stacks.
- Showcasing our developmental on a milestone.

## 8.2 Feasibility of the Project in Real-world

In this assignment, it is required to implement an HTTP API to expose part of the SUAVE's system control to a higher-level adaptive controller (UPISAS). This could be feasible in the context of other self-adaptive systems, such as Simulator of Web Infrastructure and Management (SWIM); However, in the context of robotics systems, where real-time responsiveness (ideally, within microseconds) is a strict requirement, an extra HTTP overhead would be far less than ideal. Much latency and overhead could be involved in such design:

- Latency of the TCP protocol itself (three-way hand-shaking, management of stateful connections, etc.)
- Latency and overhead introduced by the HTTP server (in our case, by using `Flask` library)

In addition, research has been made to benchmark the communication model used by ROS2, which is DDS (Data Distribution Service) and RTPS (Real Time Publish Subscribe protocol). The benchmark results shows significant latency superiority over traditional TCP and HTTP based transport [5] [6].

Therefore, the feasibility of the project in reality remains questionable. Is is recommended to specialize and integrate UPISAS as an internal node of the ROS system to achieve the best possible performance and accuracy of real-time data. More details of such implementation path can be found in subsubsection 6.5.1

## 8.3 Degree of Realism/Fineness of the Simulation Provided by SUAVE

The Degree of realism/fineness in SUAVE's simulation is a critical factor affecting the implementation of self-adaptive strategies for thruster failures (subsection 2.2). SUAVE's simulation lacks the granularity to individually simulate motors in terms of physics, mechanics, or control theory, such as PID (Proportional–integral–derivative) controller or Field Oriented Control. This limitation restricts the ability to simulate a wide range of scenarios or develop diverse adaptive solutions for thruster failures. Therefore, while SUAVE offers certain capabilities, its simulation does not fully capture the complexities needed for extensive self-adaptive strategy development in response to thruster failures.

This can be further exemplified by considering a scenario where a thruster fails due to entanglement with underwater debris. In real-world situations, this would require intricate adjustments in motor control, potentially involving complex PID control responses to compensate for the lost thrust and maintain stability. However, SUAVE's simulation, lacking detailed modeling of individual motor responses and control theory aspects, cannot accurately replicate such nuanced reactions. This gap highlights the challenge in using SUAVE for developing comprehensive self-adaptive strategies for specific thruster failure scenarios.

## 8.4 Technical Difficulties for Implementation of SAS

The following two sections will document some of the major issues we encountered during the implementation of the project.

### 8.4.1 The Managed System: SUAVE

During the implementation of the SUAVE part, the major difficulty is threading and synchronization. ROS in Python uses `rclpy`, which is essentially a Python binding of the C++ implementation of ROS ecosystem. This means the threading model of ROS in Python in essentially the same as the one that ROS used in C++, which is native threading.

However, in the design of our implementation, the Python-native HTTP library `Flask` is used, which uses Python's native threading model. Inevitably, these two components caused threading compatibility issue.

Our solution to the problem is separating the two components, resulting in adding locks for both components in the critical part of the code.

```
                if value.key == "water_visibility":
                    with self.lock:
                        self.buffer[str(value.key)] =
                         ↪   float(value.value)
```

This might result in performance trade-off during high-concurrent scenario. However, it is necessary safety measure for the stability of the system.

### 8.4.2 The Managing System: UPISAS

The bring-up time of the SUAVE system can be a little long (around 20 seconds) depending on the capability of the hardware that it runs on. If regular version of UPISAS is used, it might consider the bring-up process of the SUAVE container failed, thus repeating the bring-up and kill process.

```
    def _start_server_and_wait_until_is_up(self,
     ↪   base_endpoint="http://localhost:3000"):
        self.exemplar.start_run()
        while True:
            # The reason for the sleep is to wait each component to
             ↪   go online
            # In the launch script it should wait 5 * 3 = 15 secs,
             ↪   however, to be on the safer side, we use 20
            time.sleep(20)
            print("trying to connect...")
            response = get_response_for_get_request(base_endpoint)
            print(response.status_code)
            if response.status_code < 400:
                return
```

By patching the waiting time of UPISAS, the bring-up issue was resolved. However, the work to optimize the bring-up time still remains to be done.

## References

[1] G. R. Silva, J. Päßler, J. Zwanepol, E. Alberts, S. L. T. Tarifa, I. Gerostathopoulos, E. B. Johnsen, and C. H. Corbato, "SUAVE: An Exemplar for Self-Adaptive Underwater Vehicles," Mar. 2023. arXiv:2303.09220 [cs].

[2] J. Whittle, P. Sawyer, N. Bencomo, B. H. C. Cheng, and J.-M. Bruel, "RELAX: a language to address uncertainty in self-adaptive systems requirement," *Requirements Engineering*, vol. 15, pp. 177–196, June 2010.

[3] J. Kephart and D. Chess, "The vision of autonomic computing," *Computer*, vol. 36, pp. 41–50, Jan. 2003.

[4] E. Lee, Y.-G. Kim, Y.-D. Seo, K. Seol, and D.-K. Baik, "RINGA: Design and verification of finite state machine for self-adaptive software at runtime," *Information and Software Technology*, vol. 93, pp. 200–222, Jan. 2018.

[5] M. Xiong, J. Parsons, J. Edmondson, H. Nguyen, and D. C. Schmidt, "Evaluating the Performance of Publish/Subscribe Platforms for Information Management in Distributed Real-time and Embedded Systems,"

[6] G. Sciangula, D. Casini, A. Biondi, C. Scordino, and M. Di Natale, "Bounding the Data-Delivery Latency of DDS Messages in Real-Time Applications," pp. 26 pages, 3054317 bytes, 2023. Artwork Size: 26 pages, 3054317 bytes ISBN: 9783959772808 Medium: application/pdf Publisher: Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

## A Division of Work

Table 9 shows the internal division of work in this project.

| Name | Division of Work |
|---|---|
| [redacted] | Project analysis, Implementation of UPISAS strategies, Writing of report. |
| [redacted] | Project analysis, Implementation of UPISAS strategies, Writing of report. |
| [redacted] | Project analysis, Updated implementation of SUAVE HTTP interface, Implementation of UPISAS, Writing of report. |
| [redacted] | Project analysis, Updated implementation of SUAVE HTTP interface, Implementation of UPISAS, Writing of report. |

Table 9: Internal Division of Work

## B Code Repositories

The following implementation analysis corresponds to the following source code:

- UPISAS
  - Repository: [redacted]
  - Branch: `assignment-3`
  - Commit: *the latest commit*
- SUAVE
  - Repository: [redacted]
  - Branch: `assignment-3`
  - Commit: *the latest commit*

## C Documentation

For the documentation for development and deployment, please refer the following:

- SUAVE: [redacted]
  [redacted]
- UPISAS: [redacted]
  [redacted]