Designing Resilient Autonomous Systems with the Reflex Pattern

Julian Demicoli, Sebastian Steinhorst
Department of Computer Engineering, Technical University of Munich, Germany
firstname.lastname@tum.de

***Abstract*—Autonomous systems face significant challenges due to fluctuating resources and unstable environments, where traditional redundancy strategies for resilience can be inefficient. We present the Reflex pattern, inspired by biological reflexes, promoting system resilience by dynamically adapting to changing resource conditions. By switching between complex and resource-efficient algorithms based on availability, the pattern optimizes efficient resource utilization without extensive redundancy, ensuring essential functionalities remain operational under constraints. To facilitate adoption, we introduce ReflexLang, a domain-specific language (DSL) enabling automated code generation for reflex-pattern-based systems. We validate the pattern's effectiveness in a drone image processing scenario, demonstrating its potential to enhance operational integrity and resilience.**

***Index Terms*—Reflex Pattern, System Resilience, Dual-Algorithm Approach, Autonomous Systems, Design Specification Language, Software Architecture**

I. Introduction

Autonomous systems—including vehicles, drones, and robotics—must operate reliably amid environmental and resource fluctuations. Challenges such as real-time processing demands, energy limitations, and the critical need to maintain operational integrity make resilience a primary concern in these systems. Adaptive mechanisms are essential to maintain stability in uncertain conditions. The Reflex pattern proposed in this paper provides a dynamic, low-latency response to resource degradation, making it particularly well-suited for the challenges faced by autonomous systems.

Traditional approaches to enhancing system resilience often rely on redundancy, such as N-version programming [1] and Triple Modular Redundancy (TMR) [2], commonly used in safety-critical environments. N-version programming runs multiple independent software versions in parallel, while TMR uses hardware duplication with majority voting to mitigate faults. While effective for fault tolerance, these methods incur significant resource overhead and may not be practical in resource-constrained environments.

Recent advancements highlight a shift towards adaptable resilience strategies that enhance system reliability without extensive redundancy, such as the agent-based graceful degradation [3], the Simplex architecture [4], and the X-MAPE framework [5]. Despite their significant contributions, these approaches often fall short in addressing challenges posed by highly dynamic and distributed environments with dramatic fluctuations in resource availability. This limitation can hinder their ability to adapt efficiently to sudden drops in computational resources or network bandwidth, potentially causing performance degradation or failures.

In response, we introduce the *Reflex pattern*, inspired by biological reflexes, which dynamically adapts to resource constraints by switching to a secondary, resource-efficient reflex algorithm. This ensures continuous operation under degraded conditions, maintaining essential functionalities without the resource demands of full redundancy. By mimicking biological reflex actions that provide immediate, simplified responses to stimuli, the Reflex pattern enables systems to react swiftly to resource fluctuations. Figure 1 depicts the Reflex pattern's architecture, including dynamic algorithm switching and resource monitoring. To facilitate implementation and support automated code generation, we present *ReflexLang*, a domain-specific language (DSL) that allows developers to specify services, algorithms, resources, and switching conditions concisely, promoting systematic software design and adoption.

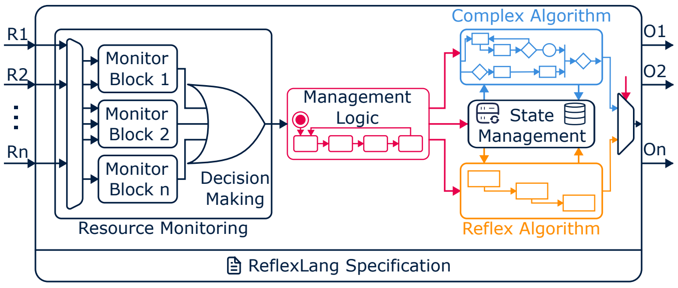

Fig. 1: The Reflex pattern, illustrating the Resource Monitoring System, the Management Logic, State Managment, and the Reflex and Complex algorithms, specified by ReflexLang.

Our contributions are as follows:

- We propose the Reflex pattern, equipping autonomous systems with a secondary reflex algorithm to ensure fail-operational behavior during resource fluctuations. By maintaining essential functionality under degraded conditions, the Reflex pattern supports the resilience of autonomous systems.
- We introduce ReflexLang, a domain-specific language that facilitates the specification and implementation of the Reflex pattern. By supporting automated code generation, ReflexLang simplifies the development process and promotes adoption of the Reflex pattern.
- We validate the Reflex pattern's effectiveness in a drone image processing scenario, demonstrating its ability to maintain operational integrity during fluctuating computational loads induced by resource degradation.

II. Related Works

Building upon the limitations of traditional redundancy methods, researchers have explored various strategies that enhance system resilience through adaptability rather than duplication. These approaches aim to maintain system functionality in the face of resource degradation, environmental fluctuations, or component failures without incurring the high resource costs associated with redundancy.

In this section, we discuss several such strategies, including the *Simplex architecture*, *agent-based graceful degradation*, and the *X-MAPE* framework. Each offers a different perspective on achieving resilience, focusing on adaptive algorithms, resource reallocation, and efficient system monitoring to handle uncertainties and maintain operational integrity.

A. Simplex Architecture

The Simplex architecture [4] is a control framework designed to ensure system safety through a dual-controller approach. It comprises a high-performance controller and a high-assurance controller. The high-performance controller handles the primary operation, delivering optimal performance under normal conditions. By contrast, the high-assurance controller acts as a safety backup, taking over control when the system detects a fault in the high-performance controller's output.

Initially developed to support dependable system upgrades [6], [7] and safe online control system updates [8] for real-time systems, the Simplex architecture allows a system to recover to a safe state even if the high-performance controller fails—such as after a faulty software update—by switching control to the high-assurance controller. This ensures system safety, albeit with potentially reduced performance.

Wh

The Simplex architecture employs a dual-controller approach, with the high-assurance controller running in parallel with the high-performance controller to ensure fault tolerance [4]. While this approach guarantees safety, it increases the system's resource footprint, making it less suitable for resource-constrained environments. In contrast, our Reflex pattern activates the secondary, less resource-intensive algorithm only when needed, optimizing resource utilization.

B. Agent-Based Graceful Degradation for Fail-Operational Automotive Software

In autonomous driving systems, where there is no human driver to serve as a fallback, traditional fail-safe mechanisms are insufficient. In [3], an agent-based graceful degradation strategy is explored as an alternative to resource redundancy. Their approach repurposes computational resources from non-critical tasks to safety-critical applications, ensuring fail-operational behavior without additional hardware investment. This is achieved by delegating services to agents for execution, with a secondary, shadowing agent residing on another core. The shadowing agent remains inactive during normal operation but synchronizes its state with the primary agent and monitors its availability. If the primary agent ceases execution—due to software faults, for example—the shadowing agent takes over the service.

While this method shares our goal of minimizing resource redundancy, it primarily focuses on maintaining the software execution of a specific service through agent replication. In contrast, our *Reflex pattern* extends this concept by enabling systems to handle a broader range of failures, including hardware degradation and other resource constraints. Unlike the agent-based approach, which requires continuous synchronization of the shadowing agent (thereby consuming additional resources), our pattern activates the secondary algorithm only when necessary. This on-demand activation optimizes resource utilization and enhances resilience by adapting to fluctuating resource availability.

C. X-MAPE

The X-MAPE framework [5] extends the MAPE-K loop [9], which is designed to achieve system autonomy through a feedback loop consisting of monitoring, analysis, planning, and execution phases supported by a Knowledge base. A significant limitation of the standard MAPE-K loop is the latency introduced by each iteration, which can be substantial in dynamic environments.

To mitigate this issue, X-MAPE incorporates an additional reflex path that directly connects the Monitor and Execute phases. This low-latency path allows the system to react quickly to uncertain or rapidly changing states, such as sudden environmental fluctuations, by bypassing the time-consuming analysis and planning stages.

While X-MAPE employs a reflexive mechanism to reduce response time, its primary aim is to enhance the adaptability of systems to uncertain contexts by improving feedback loop latency. In contrast, our *Reflex pattern* not only reduces latency but also addresses resource constraints by switching to a less resource-intensive secondary algorithm. Our approach is designed to maintain essential functionality under degraded conditions, thereby achieving fail-operational behavior through both timely reactions and efficient resource management.

III. The Reflex pattern

A. Intent

The Reflex pattern enhances system resilience by enabling services to dynamically adapt to resource constraints through algorithm switching. It allows a system to maintain acceptable Quality of Service (QoS) even under suboptimal conditions by transitioning to a less resource-intensive mode of operation. This pattern ensures that essential functionalities remain available, preventing complete system failure during resource degradation.

B. Motivation

Autonomous systems often face fluctuating resource availability due to increased load, hardware limitations, or network issues. Traditional systems may struggle to maintain optimal performance under these conditions, leading to degraded system performance. The Reflex pattern addresses this challenge by introducing a mechanism for systems to detect resource constraints and adapt accordingly by switching to a less resource-intensive algorithm. This adaptive behavior ensures continuity of essential services while optimizing resource utilization.

C. Structure

At its core, the Reflex pattern consists of the Complex and Reflex algorithms, the Resource Monitoring System, the Management Logic and the State Management, as depicted in Figure 1. To provide a clear structural overview of the Reflex pattern, Figure 2 presents the class diagram detailing the main components and their relationships. The State Management is not depicted due to its dependency on the chosen state strategy.

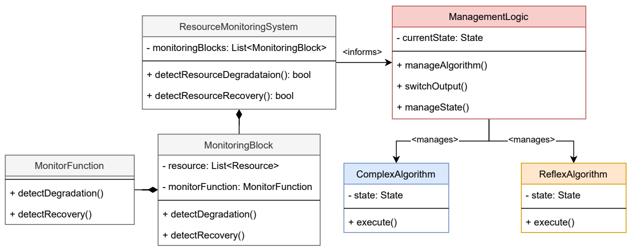

Fig. 2: Class diagram illustrating the Management Logic, the Complex and Reflex Algorithm, and the Resource Monitoring System and their interactions.

1) Complex and Reflex Algorithms: The *Complex Algorithm* represents the standard operation of the system, utilizing available resources to provide complete functionality and the highest QoS. It is designed to perform optimally under normal resource conditions.

The *Reflex Algorithm* provides a fallback mode that requires fewer resources, ensuring continued operation during periods of resource constraints. While it may offer reduced performance or limited functionality compared to the Complex algorithm, it maintains essential services to prevent system failure.

Both algorithms should be designed to produce compatible outputs to facilitate seamless transitions. Compatibility ensures that switching between algorithms does not disrupt the system's interaction with external components.

2) Resource Monitoring System: The *Resource Monitoring System* is responsible for continuously observing the system's resource availability and utilization to identify when resource constraints occur. It plays a critical role in determining the appropriate timing for algorithm switching.

This system comprises multiple monitoring blocks, each connected to resource inputs of the Reflex unit. These monitoring blocks implement configurable functions ranging from simple threshold-based monitors to sophisticated mechanisms such as anomaly detection or predictive analytics. Operating independently and in parallel, the monitoring blocks detect resource degradation and recovery efficiently. The outputs of these blocks are connected via a logical OR operation to the Management Logic, which then initiates and manages the algorithm switch.

By making advanced monitoring capabilities optional and modular, the Resource Monitoring System balances complexity with simplicity, allowing developers to tailor it to their specific application needs.

3) Management Logic: The *Management Logic* orchestrates the transition between the Complex and Reflex Algorithms based on signals from the Resource Monitoring System. Its design emphasizes simplicity and application independence, delegating application-specific logic to the algorithms themselves.

The switching decision is implemented as a logical OR operation over the outputs of the Resource Monitoring System. If any monitored resource indicates a constraint, the Management Logic initiates a switch.

Figure 3 visualizes the behavior of the Management Logic after a resource degradation has been detected. For simplicity, the State Management between the Complex and Reflex Algorithms is omitted.

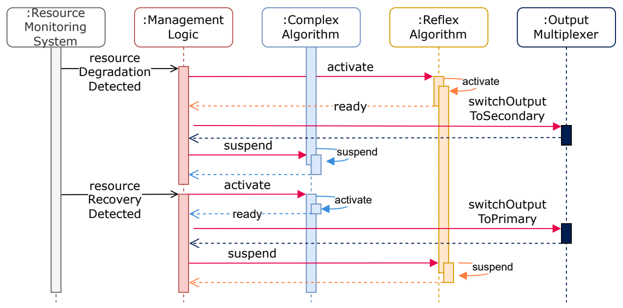

Fig. 3: Sequence diagram illustrating the actors of the Reflex pattern switching to the Reflex algorithm and back when a resource degradation is detected and later recovered.

4) State Management: Effective *State Management* ensures that the system's internal state remains consistent and accurate during algorithm switching. This is particularly important for stateful services where continuity of operation depends on state preservation. We focus on *Stateless Design*, *Checkpointing and State Transfer* [10], and *Shared Data Store* within the Reflex pattern. Each method offers different trade-offs between complexity, performance, and resource utilization. Below, we discuss each methodology and provide guidance on when to use them.

a) Stateless Design: Stateless design involves structuring algorithms so they do not maintain internal state between processing cycles. Each input is processed independently, with all necessary data provided as input. This approach is suitable when the application can process inputs without relying on historical data and simplicity is a priority.

b) Checkpointing and State Transfer: Checkpointing involves periodically saving the state of the Complex Algorithm so it can be transferred to the Reflex Algorithm upon activation. This method balances state consistency with resource utilization and should be used when the state size is manageable and brief switching delays are acceptable.

c) Shared Data Store: Using a shared data store allows both algorithms to access common storage for state information, ensuring they operate on the same data. This approach should be used when strong state consistency is required and the performance impact of shared storage access is acceptable.

IV. ReflexLang

ReflexLang is a domain-specific language (DSL) designed to facilitate the specification and implementation of services employing the Reflex pattern introduced in Section III-C. It abstracts the complexities involved in implementing the pattern, allowing developers to focus on the core functionality of their applications.

We provide a brief overview of ReflexLang's key components and refer interested readers to our Git repository[1] for the complete grammar definitions, examples, and detailed explanations.

[1]https://github.com/demicoli/ReflexLang

A. Overview

ReflexLang provides constructs for defining:

- **Services**: Encapsulate the overall functionality employing the Reflex pattern.
- **Algorithms**: Specify the Complex and Reflex algorithms with support for code inclusion and shadowing mechanisms. (Section III-C1)
- **Resource Monitoring System**: Implement Resources and Monitoring Blocks with monitoring logic, including threshold-based and custom monitoring functions. (Section III-C2)
- **State Management Strategies**: Specify how state is managed between algorithm transitions, supporting stateless design, checkpointing, and shared data stores. (Section III-C4)

B. Core Grammar

The extended Backus–Naur form (EBNF) grammar of ReflexLang is presented in Listing 1. This grammar captures the essential syntax needed to define services and their components.

```
Service_Definition ::= 'Service' ServiceName '{'
    Complex_Algorithm Reflex_Algorithm {
    Resource_Definition } { Monitoring_Block }
    State_Management '}';
Complex_Algorithm ::= 'Complex_Algorithm' AlgorithmName
    '{' Algorithm_Body '}';
Reflex_Algorithm ::= 'Reflex_Algorithm' AlgorithmName '{'
    Algorithm_Body [ 'Shadowing' ':' ( 'true' | 'false'
    ) ] '}';
Algorithm_Body ::= Code_Block | Code_Reference
Code_Block ::= 'Code' '{' [ 'Language' ':' LanguageName ]
    Code_Content '}' ;
Code_Reference ::= 'Include' StringLiteral [ 'as'
    ModuleName ] ;
Resource_Definition ::= 'Resource' ResourceName '{'
    'Type' ':' Resource_Type 'DataType' ':' DataType [
    Resource_Parameters ] '}' ;
Resource_Type ::= 'Variable' | 'Buffer' | 'List' |
    'Queue' | 'Custom' ;
Resource_Parameters ::= { ParameterName ':' Value } ;
Monitoring_Block ::= 'Monitoring_Block' BlockName '{'
    'Inputs' ':' ResourceList Monitoring_Function '}' ;
ResourceList ::= ResourceName { ',' ResourceName } ;
Monitoring_Function ::= Threshold_Monitoring |
    External_Monitoring ;
Threshold_Monitoring ::= 'Threshold_Monitoring' '{'
    'Activate_Threshold' ':' Value
    'Deactivate_Threshold' ':' Value '}' ;
External_Monitoring ::= 'Monitoring_Algorithm'
    AlgorithmName '{' Algorithm_Body [ Parameters ] '}' ;
Parameters ::= 'Parameters' '{' { ParameterName ':' Value
    } '}' ;
State_Management ::= 'State_Management' '{' 'Strategy'
    ':' State_Strategy [ Data_Definition ] '}' ;
State_Strategy ::= 'Stateless' | 'Checkpointing' |
    'Shared_Data_Store' ;
Data_Definition ::= 'Data' '{' { Data_Item } '}' ;
Data_Item ::= VariableName ':' DataType [ 'Structure' ':'
    Data_Structure ] ;
```

Listing 1: EBNF Grammar for ReflexLang.

V. Evaluation

To assess the effectiveness of the Reflex pattern and understand how specific factors influence system performance, we conducted two experiments using a simulated *drone image processing system*. In this system, images are processed using object detection algorithms to simulate real-world computational loads and resource constraints.

The experiments demonstrate how the Reflex pattern enhances system resilience under resource constraints, such as high computational load. We specifically investigate the impact of switching delays between the Reflex and Complex algorithms and the effect of optional mechanisms like shadowing on Quality of Service (QoS) metrics.

To support reproducibility and further research, we have developed a repository that implements the ReflexLang DSL in Python using the Lark framework[2]. Additionally, we created an experimental code generation module to showcase how the language facilitates development by enabling automatic code generation. The code for the drone image processing system, along with detailed explanations, is available online in our repository.

While this paper focuses on a drone image processing scenario, the Reflex pattern is broadly applicable to other autonomous systems—including self-driving vehicles, robotic manufacturing systems, and smart infrastructure—where adaptive algorithm switching can enhance system resilience.

A. Reflex Pattern Specification

We defined the reflex behavior of the drone image processing system using our ReflexLang DSL, reflecting the constructs described in Section IV. The specification is presented in Listing 2.

```
Service DroneImageProcessing {
    Complex_Algorithm ComplexDetection {
        Include "complex_detection.py" as ComplexDetect }
    Reflex_Algorithm SimpleDetection {
        Include "simple_detection.py" as SimpleDetect
            Shadowing: true }
    Resource Queue_Length {
        Type: Variable
        DataType: Integer }
    Monitoring_Block QueueLengthMonitor {
        Inputs: Queue_Length
        Threshold_Monitoring {
            Activate_Threshold: 80
            Deactivate_Threshold: 60 } }
    State_Management {
        Strategy: Stateless  } }
```

Listing 2: ReflexLang specification for drone image processing service.

In this specification, the system switches from the Complex algorithm, which utilizes the YOLOv5l model [11], to the Reflex algorithm, which uses the lightweight YOLOv5s model, when the queue length exceeds 80%. The `Shadowing: true` directive enables the shadowing mechanism, allowing the Reflex algorithm to run in parallel and be ready for immediate activation. This minimizes the switching delay

[2]https://github.com/lark-parser/lark

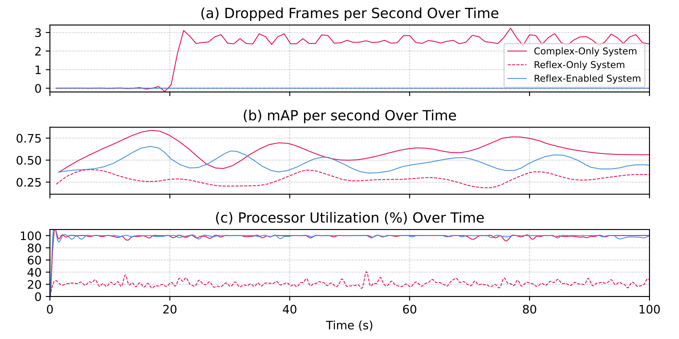

Fig. 4: Comparison of (a) dropped frames per second, (b) mAP per second, and (c) processor utilization over time for the Complex-only, Reflex-only, and Reflex-enabled systems under high load conditions. The Reflex-enabled system maintains a lower amount of dropped frames by preventing buffer overflows through adaptive algorithm switching. Compared to the Reflex-only system, it optimizes resource utilization with processor utilization close to 100%, resulting in a higher mAP per second without dropping frames.

when the system needs to adapt to high computational loads, enhancing responsiveness. We employ a *Stateless* strategy in the `State_Management` block, facilitated by an external buffer.

B. Dataset and Model Details

Images are sampled from the COCO dataset [12] to facilitate development and ensure compatibility with the YOLOv5 models, which are trained on the same dataset. This approach allows us to leverage consistent image boundaries and annotations, enhancing the reliability of our experiments.

To measure the performance of the object detection, we use the mean average precision (mAP) score. On our test system, which runs on a single core of a Intel Core i7 (11370H) processor, the YOLOv5s model processes approximately 17 iterations per second with an mAP of 0.292, while the YOLOv5l model achieves about 2 iterations per second with an mAP of 0.562. By switching to the simpler model during high load, the system maintains a higher throughput, albeit with a significant reduction in detection accuracy.

C. Experiment 1: Comparing Reflex vs. Non-Reflex Systems

a) Objective: This experiment compares the performance of three drone image processing systems under high load conditions: a *Complex-only* system using the YOLOv5l model, a *Reflex-only* system using the YOLOv5s model, and a *Reflex-enabled* system that dynamically switches between the two based on system load by applying our Reflex pattern. The goal is to demonstrate the Reflex pattern's ability to optimize resource utilization, maintain service availability, and improve overall performance compared to non-adaptive systems.

b) Methodology: All systems were simulated under a high load of 4 images per second, representing the input from the drone's camera. We evaluated them based on key QoS metrics: error rate (percentage of frames dropped due to buffer overflow), average processor utilization over time, and average mAP for object detection accuracy.

c) Results and Analysis: Figure 4 presents the queue fill level, frames per second (FPS), and processor utilization over time for the three systems. The *Complex-only* system maintained 100% CPU utilization but suffered an error rate of 61.42%, dropping frames when the buffer overflowed due to its inability to process all incoming images. It achieved a high mAP score of 0.506 owing to the more accurate YOLOv5l model but processed fewer frames than the arrival rate.

The *Reflex-only* system processed all incoming images without dropping any frames, resulting in a 0% error rate and matching the arrival rate with an average FPS of 4. However, it underutilized computational resources, with CPU utilization around 20%, leading to wasted capacity. Its mAP score was lower at 0.259 due to the less accurate YOLOv5s model.

The *Reflex-enabled* system dynamically adjusted CPU utilization, averaging close to 100% by switching between the Complex and Reflex algorithms based on buffer fill levels. It processed all incoming images without dropping frames (0% error rate) and matched the arrival rate with an average FPS of 4. The mAP score improved to 0.332—28% higher than the Reflex-only system—by utilizing the more accurate Complex algorithm whenever possible.

These results demonstrate that the Reflex-enabled system effectively balances resource utilization and performance. By activating the Complex algorithm when the buffer allows and reverting to the Reflex algorithm under high load, it ensures full utilization of computational resources without overloading the system, thereby preventing frame drops and maintaining higher detection accuracy compared to the Reflex-only system. By contrast, non-adaptive systems either cannot handle high load without dropping frames (Complex-only) or underutilize resources (Reflex-only), leading to suboptimal performance.

D. Experiment 2: Impact of Switching Delays

a) Objective: In dynamic and resource-constrained systems, timely response to resource degradation is crucial for maintaining system resilience. Delays in switching between Complex and Reflex algorithms can significantly impact overall system performance, especially under heavy computational loads. As noted in [13], reducing latency in dynamic adaptation leads to substantial gains in system resilience by minimizing downtime and improving the ability to quickly recover from faults or failures. This aligns with the Reflex pattern's design goal of maintaining operational integrity by adapting swiftly to resource fluctuations. Therefore, in Experiment 2, we explore how varying the delay in activating the Reflex algorithm influences system performance, resilience, and the overall QoS.

b) Methodology: We simulated the Reflex-Enabled drone image processing system under high load conditions with an image arrival rate of 5 images per second, slightly higher than in the first experiment. Switching delays ranging from 0 to 25 seconds were tested for transitioning from the Complex algorithm to the Reflex algorithm when the queue length exceeded 80%. The simulation ran for 500 seconds, and other parameters were consistent with Experiment 1.

During the switching delay period, the system remains idle, not processing incoming images until the Reflex algorithm is activated. This approach emphasizes the impact of switching delays on system performance, particularly on frame loss and

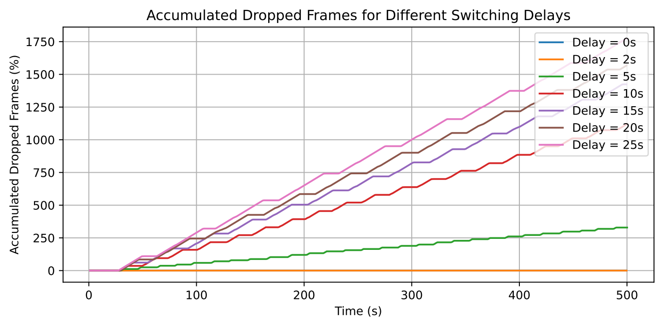

Fig. 5: Accumulated dropped frames over time for different switching delays, showing a linear increase in frame loss as the switching delay increases.

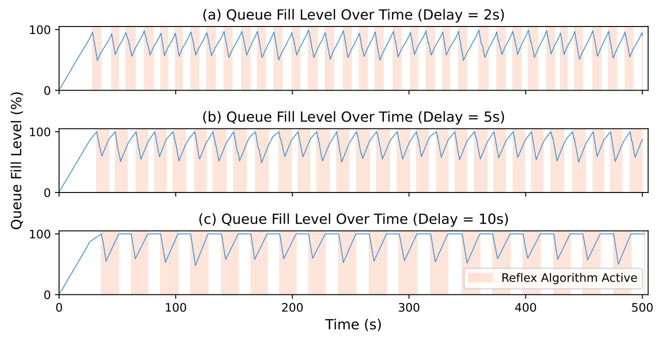

Fig. 6: Queue fill level over time for switching delays of (a) 2 seconds, (b) 5 seconds, and (c) 10 seconds. It illustrates that a higher delay for Reflex algorithm activation leads to a lower overall Reflex frequency which results in a higher queue fill level and, therefore, a higher error rate.

detection accuracy. The `Shadowing` mechanism was disabled to isolate the effect of switching delays.

c) Results and Analysis: Figure 5 shows the accumulated dropped frames over time for switching delays of 0, 5, 10, 15, 20, and 25 seconds. Initially, the system experiences no frame loss due to the buffer temporarily absorbing the incoming images. However, as the switching delay increases, the system takes longer to activate the Reflex algorithm in response to high queue levels, leading to more frames being dropped when the buffer overflows. The accumulated frame loss increases more rapidly with longer switching delays.

Figure 6 presents the queue fill level over time for switching delays of 2, 5, and 10 seconds. Shorter switching delays allow the system to activate the Reflex algorithm more promptly when the queue level exceeds the threshold, preventing buffer overflows and reducing frame loss. By contrast, longer delays result in the queue reaching maximum capacity more frequently, leading to increased frame drops.

d) Key Observations:

- **Timely Reflex Activation is Crucial**: Minimizing switching delays enables the system to adapt swiftly to high CPU usage, reducing error rates and maintaining efficient processing times.
- **Benefits of Shadowing**: Implementing the shadowing mechanism can effectively eliminate switching delays, leading to significant performance improvements.
- **Trade-off Between Detection Accuracy and Performance**: While longer switching delays improve average detection accuracy by operating under the Complex algorithm longer, they adversely affect error rates and processing times. The marginal gain in detection accuracy does not compensate for the significant degradation in other QoS metrics.

E. Discussion

The experiments demonstrate that the Reflex pattern significantly enhances system resilience and performance in resource-constrained drone image processing. By adaptively switching algorithms based on CPU load, the system maintains service availability and efficiently handles high demand. Minimizing switching delays, especially through shadowing, is crucial for optimal performance.

The Reflex-Enabled system outperforms the non-adaptive systems under heavy loads by reducing error rates and maintaining higher detection accuracy compared to the Reflex-only system.

These findings highlight the Reflex pattern's effectiveness in improving the robustness of autonomous systems. Its adaptability makes it applicable beyond drone imaging to other domains requiring resilience under varying resource conditions.

VI. Conclusion and Outlook

In this paper, we introduced the Reflex pattern, a design approach that enhances system resilience by enabling dynamic adaptation to resource constraints through algorithm switching. Particularly suited for autonomous systems, this pattern ensures continuous operation in environments where resource availability fluctuates, such as computational shortages and network instability. We developed ReflexLang, a domain-specific language that facilitates the specification and implementation of services employing the Reflex pattern.

Through simulation experiments with a drone image processing system, we demonstrated that the Reflex pattern significantly improves performance under high load conditions by reducing error rates and maintaining processing throughput. The ability to switch between algorithms based on resource availability allows the system to adapt dynamically, ensuring operational integrity.

The Reflex pattern's ability to automatically manage system behavior in resource-constrained conditions aligns with the critical demands of autonomous systems, ensuring operational continuity and service availability in real-world applications.

Future work will focus on implementing the Reflex pattern in diverse real-world autonomous systems to validate our simulation results and demonstrate its versatility across various domains. Additionally, we plan to explore advanced state management strategies and more sophisticated resource monitoring techniques to further enhance the pattern's effectiveness.

VII. Acknowledgement

The authors acknowledge the financial support by the Federal Ministry of Education and Research of Germany in the programme of "Souverän. Di

al. Vernetzt.". Joint project 6G-life, project identification number: 16KISK002

References

[1] A. Avizienis, “The n-version approach to fault-tolerant software,” *Software Engineering, IEEE Transactions on*, vol. SE-11, pp. 1491 – 1501, 01 1986.

[2] R. E. Lyons and W. Vanderkulk, “The use of triple-modular redundancy to improve computer reliability,” *IBM Journal of Research and Development*, vol. 6, no. 2, pp. 200–209, 1962.

[3] P. Weiss, A. Weichslgartner, F. Reimann, and S. Steinhorst, “Fail-operational automotive software design using agent-based graceful degradation,” in *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2020, pp. 1169–1174.

[4] L. Sha, “Using simplicity to control complexity,” *IEEE Software*, vol. 18, no. 4, pp. 20–28, 2001.

[5] M. Liess, J. Demicoli, T. Tiedje, M. Lohrmann, M. Nickel, M. Luniak, D. Prousalis, T. Wild, R. Tetzlaff, D. Göhringer, C. Mayr, K. Bock, S. Steinhorst, and A. Herkersdorf, “X-mape: Extending 6g-connected self-adaptive systems with reflexive actions,” in *2023 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, 2023, pp. 163–167.

[6] L. Sha, R. Rajkumar, and M. Gagliardi, “Evolving dependable real-time systems,” in *1996 IEEE Aerospace Applications Conference. Proceedings*, vol. 1, 1996, pp. 335–346 vol.1.

[7] L. Sha, “Dependable system upgrade,” in *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No.98CB36279)*, 1998, pp. 440–448.

[8] D. Seto, B. Krogh, L. Sha, and A. Chutinan, “The simplex architecture for safe online control system upgrades,” in *Proc. of American Control. ACC.* IEEE, 1998.

[9] J. O. Kephart and D. M. Chess, “The vision of autonomic computing,” *Computer*, vol. 36, no. 1, 2003.

[10] L. Zhang, Z. Wang, and F. Kong, “Optimal checkpointing strategy for real-time systems with both logical and timing correctness,” *ACM Trans. Embed. Comput. Syst.*, vol. 22, no. 4, Jul. 2023. [Online]. Available: https://doi.org/10.1145/3603172

[11] G. Jocher, “Yolov5 by ultralytics,” 2020. [Online]. Available: https://github.com/ultralytics/yolov5

[12] T. Lin, M. Maire, S. J. Belongie, L. D. Bourdev, R. B. Girshick, J. Hays, P. Perona, D. Ramanan, P. Doll’a r, and C. L. Zitnick, “Microsoft COCO: common objects in context,” *CoRR*, vol. abs/1405.0312, 2014. [Online]. Available: http://arxiv.org/abs/1405.0312

[13] L. Prenzel and S. Steinhorst, “Towards resilience by self-adaptation of industrial control systems,” in *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2022, pp. 1–8.