Degree Project in Computer Science

Second cycle, 30 credits

Designing a Priority-Based Message Scheduling System for Node-RED IoT Applications

DOMINIKA DRELA

Designing a Priority-Based Message Scheduling System for Node-RED IoT Applications

DOMINIKA DRELA

Master's Programme, Software Engineering of Distributed Systems, 120 credits
Date: June 23, 2026

Supervisors: Ki Won Sung, Thiemo Voigt, Luca Mottola, Diletta Romano
Examiner: Šarūnas Girdzijauskas
School of Electrical Engineering and Computer Science
Host organization: RISE Research Institutes of Sweden
Swedish title: Design av ett prioritetsbaserat meddelandeschemaläggningssystem för Node-RED IoT-applikationer

Abstract

Internet of Things applications frequently need to process messages within a given time. Some messages are more urgent than others and need to be handled first. Although Node-RED is commonly used as a flow-based programming tool for creating IoT systems, it currently handles all messages in the order of arrival (first-in-first-out). This is a significant drawback for time-sensitive applications since it makes it impossible to prioritise urgent messages over less crucial ones. Despite Node-RED's prominence in IoT development, no prior work has introduced deadline-aware scheduling at the flow execution level.

In this thesis, a priority-based message scheduling system was designed and implemented for Node-RED. The solution organises messages by priority using a binary min-heap data structure. Node-RED's Flow.js runtime was modified, and a centralised MessageScheduler was added, capable of supporting multiple scheduling policies. Two approaches were implemented: static priority scheduling, allowing users to assign fixed priority levels to messages, and Earliest Deadline First (EDF) scheduling, where messages with earlier deadlines are processed first. Laxity-based priority recalculation was incorporated as an extension to the EDF approach.

The evaluation comprised 6000 test runs across five message counts and four simulated processing times. Correctness testing confirmed that both schedulers reliably enforce message ordering - priority 10 messages achieved an average latency of 6.7ms compared to 33.0ms for priority 90, and EDF results ranged from 4.8ms at deadline 50ms to 30.1ms at deadline 250ms. Both schedulers maintained comparable throughput to vanilla FIFO, averaging between 7000 and 14000 messages per second, with CPU usage remaining up to 6% and peak memory of 148MB throughout testing. The implementation preserves full backward compatibility with existing Node-RED flows, ensuring seamless integration into current deployments.

Keywords

Priority queue scheduling, Internet of Things, Flow-based programming, Node-RED platform, Message scheduling algorithms, Binary heap data structures

Sammanfattning

Internet of Things-applikationer behöver ofta behandla meddelanden inom en given tid – vissa meddelanden är mer brådskande än andra och måste hanteras först. Även om Node-RED är ett vanligt verktyg för flödesbaserad programmering inom IoT-system, hanterar det alla meddelanden i ankomstordning (först in, först ut). Detta är en betydande nackdel för tidskritiska applikationer eftersom det omöjliggör prioritering av brådskande meddelanden framför mindre viktiga. Trots Node-REDs framträdande roll inom IoT-utveckling har inget tidigare arbete introducerat deadlinebaserad schemaläggning på flödeskörningsnivå.

I denna avhandling har ett prioritetsbaserat meddelandeschemaläggningssystem designats och implementerats för Node-RED. Lösningen organiserar meddelanden efter prioritet med hjälp av en binär min-heap-datastruktur. Node-REDs Flow.js-runtime modifierades och en centraliserad MessageScheduler lades till, kapabel att stödja flera schemaläggningspolicyer. Två metoder implementerades: statisk prioritetsschemaläggning, som låter användare tilldela fasta prioritetsnivåer till meddelanden, samt Earliest Deadline First-schemaläggning (EDF), där meddelanden med tidigare deadlines behandlas först. Laxitetsbaserad prioritetsomräkning inkorporerades som en utvidgning av EDF-metoden.

Utvärderingen omfattade 6000 testkörningar över fem meddelandeantal och fyra simulerade bearbetningstider. Korrekthetstestning bekräftade att båda schemaläggarna tillförlitligt upprätthåller meddelandeordning - meddelanden med prioritet 10 uppnådde en genomsnittlig latens på 6,7 ms jämfört med 33,0 ms för prioritet 90, och EDF-resultaten varierade från 4,8 ms vid deadline 50 ms till 30,1 ms vid deadline 250 ms. Båda schemaläggarna upprätthöll jämförbar genomströmning med vanilla FIFO, i genomsnitt mellan 7000 och 14000 meddelanden per sekund, med CPU-användning upp till 6% och maximalt minnesutnyttjande på 148MB under testningen. Implementationen bevarar full bakåtkompatibilitet med befintliga Node-RED-flöden, vilket säkerställer sömlös integration i nuvarande driftsättningar.

Nyckelord

Prioritetskö, schemaläggning, Internet of Things, flödesbaserad programmering, Node-RED, meddelandeschemaläggning, binär heap

Acknowledgments

I would like to express my sincere gratitude to my industrial supervisors at RISE Research Institutes of Sweden - Thiemo Voigt, Luca Mottola, and Diletta Romano - for their guidance, feedback, and support throughout this project. Their expertise and constructive criticism were invaluable in shaping both the implementation and the evaluation of this work.

I would also like to thank my academic adviser, Ki Won Sung, and my examiner Šarūnas Girdzijauskas at KTH Royal Institute of Technology for their support and oversight during the thesis process.

On a personal note, I would like to thank my father and brother for always encouraging me to chase my dreams and never give up. Their belief in me has been a constant source of motivation throughout this journey. I am also deeply grateful to my closest friends, Amira and Mateusz, for always being there for me during the whole process - your support meant more than words can express.

Stockholm, June 2026
Dominika Drela

Use of Generative AI Tools

The research presented in this thesis - including its design, system implementation, data collection, evaluation, analysis, and conclusions - is entirely the author's own work. Generative AI tools (Anthropic Claude) were used in a limited, assistive capacity only: for language refinement and phrasing, for guidance on structure, and as a sounding board for clarifying concepts during writing. The author takes full responsibility for the final content.

Contents

1 Introduction **1**
1.1 Context: Distributed IoT and Node-RED 1
1.2 Problem: Latency for Critical Messages under FIFO Pipelines 3
1.3 Objectives and Research Questions 4
1.4 Contributions 5
1.5 Thesis Structure 5

2 Background **7**
2.1 Node-RED: Architecture, Flow.js, Message Pipeline 7
2.2 Priority Queue Data Structures (Heaps, Hybrids) 8
2.3 Queues in Event-Driven Environments (Node.js) 9
2.4 Priority Scheduling (Preemptive/Non-preemptive) and Priority Inversion 10
2.5 Existing Solutions in Node-RED/IoT and Gaps 11

3 Method **13**
3.1 Research Process 13
3.2 Research Paradigm 14
3.3 Data Collection 14
3.3.1 Sampling 14
3.3.2 Sample Size 14
3.3.3 Target Population 15
3.4 Experimental Design and Planned Measurements 15
3.4.1 Test Environment 15
3.4.2 Hardware and Software 15
3.5 Assessing Reliability and Validity 16
3.5.1 Validity of Method 16
3.5.2 Reliability of Method 16
3.5.3 Data Validity 16

3.5.4 Reliability of Data 17
3.6 Planned Data Analysis 17
3.6.1 Data Analysis Technique 17
3.6.2 Software Tools 17
3.7 Evaluation Framework 17
3.8 System Documentation 18

4 Implementation **19**
4.1 Functional and Non-Functional Requirements 19
4.2 Architecture 19
4.3 Classes and APIs 20
4.4 Pseudocode of Core Algorithms 20
4.5 Batching Policies and Event-Loop Interaction 23
4.6 Overflow Handling 23
4.7 Concurrency and Backward Compatibility 24

5 Results **25**
5.1 Scheduling Correctness 25
5.2 Latency and Throughput vs Load 27
5.3 Overhead vs Vanilla FIFO 31
5.4 Resource Usage (CPU/Memory) 31
5.5 Statistical Analysis 34

6 Discussion **37**
6.1 Interpretation and Practical Implications for IoT Flows 37
6.2 Limitations 38
6.3 Guidelines for Node-RED Users 38

7 Conclusions and Future Work **41**
7.1 Answers to the Research Questions 41
7.2 Main Contributions 42
7.3 Extensions 42
7.4 Ethical Considerations, Societal Benefits, and Sustainability . 43

References **45**

A Test Flow Configurations (JSON) **49**
A.1 Static Priority Queue Correctness Test Flow 49
A.2 EDF Correctness Test Flow 50

B Pseudocode and Code Extracts **51**

C Extended Result Tables **52**

List of Figures

5.1 Correctness verification - average end-to-end latency per bucket for Static Priority Queue (top) and EDF (bottom). 30 iterations, 100 messages per bucket. 26
5.2 Static Priority Queue - total processing time (top) and average message latency (bottom). 50ms app delay, 100 iterations. . . 27
5.3 Static Priority Queue - throughput (top) and overhead vs vanilla FIFO (bottom). 50ms app delay, 100 iterations. 28
5.4 EDF Scheduler - total processing time (top) and average message latency (bottom). 50ms app delay, 100 iterations. . . 29
5.5 EDF Scheduler - throughput (top) and overhead vs vanilla FIFO (bottom). 50ms app delay, 100 iterations. 30
5.6 Scheduler overhead vs vanilla FIFO - 0ms app delay, 100 iterations. 31
5.7 Scheduler overhead vs vanilla FIFO - 50ms app delay, 100 iterations. 32
5.8 Scheduler overhead vs vanilla FIFO - 100ms app delay, 100 iterations. 32
5.9 Scheduler overhead vs vanilla FIFO - 200ms app delay, 100 iterations. 33
5.10 Node-RED CPU and memory usage during a full 50ms test run. 180 samples over 371 seconds, sampled every 2 seconds. 33

List of Tables

C.1 Full performance results – 0ms and 50ms app delay (means over 100 iterations) 53
C.2 Full performance results – 100ms and 200ms app delay (means over 100 iterations) 54

Listings

A.1 Static Priority Queue Correctness Test Flow 49
A.2 EDF Correctness Test Flow 50
B.1 Flow.js message interception and routing 51

List of acronyms and abbreviations

API	Application Programming Interface
CPU	Central Processing Unit
CSV	Comma-Separated Values
DSR	Design Science Research
EDF	Earliest Deadline First
FIFO	First-In-First-Out
HTTP	Hypertext Transfer Protocol
IoT	Internet of Things
JSON	JavaScript Object Notation
KTH	Kungliga Tekniska Högskolan (Royal Institute of Technology)
RAM	Random Access Memory
RISE	Research Institutes of Sweden
RMS	Rate Monotonic Scheduling
RSS	Resident Set Size
UI	User Interface
VM	Virtual Machine

Chapter 1

Introduction

In today's world, the Internet of Things is being used in various applications such as industrial automation or health monitoring, where proper timing is crucial. Deterministic message processing is needed to make sure all the processes are handled urgently according to their priority and deadlines. Node-RED, a development tool for flow-based programming used in IoT, currently processes messages using a FIFO (First-In-First-Out) queuing mechanism. As a result, users cannot prioritize critical messages over routine traffic, even for critical events such as leak detection alerts or temperature thresholds being exceeded.

This thesis aims to address this gap by designing and implementing priority-based message scheduling for Node-RED, which enables a deterministic control flow when needed. This chapter introduces the context of distributed IoT and Node-RED, problems regarding latency for critical messages under FIFO pipelines, objectives, research questions, contributions, and thesis structure.

1.1 Context: Distributed IoT and Node-RED

The Internet of Things (IoT) has transformed numerous industries by enabling billions of devices to connect and communicate with each other. These devices collect sensor data, process it, and exchange it across distributed systems. The sensors are deployed at the edge, processing happens on gateways or edge devices, and data is being stored in a centralised cloud infrastructure. This distributed architecture does offer flexibility and scalability, yet it also creates challenges for proper timing, message coordination, and consistent control flow across the system [1].

Traditionally, extensive coding expertise and deep knowledge of protocols and APIs were needed to connect and program IoT devices. Node-RED emerged as a solution to this complexity, providing a visual, flow-based programming environment. It was developed by IBM for wiring together hardware devices, APIs, and online services [2]. The platform's interface consists of a browser and a drag-and-drop feature that allows users to create IoT flows without coding. It makes it more accessible to engineers without any programming background. Node-RED was released as open-source in 2013 and experienced significant growth with the community creating over 4,300 nodes that extend its functionality [3]. Nowadays, Node-RED is used in various industries such as manufacturing, energy management, as well as smart building automation, proving its transformation from a simple prototyping tool into a production-grade platform [4].

Node-RED is implemented on top of Node.js and uses an event-driven, non-blocking model, making it suitable for IoT applications deployed on resource-constrained edge devices [5]. Node-RED's computation is structured around events and flows between nodes connected by wires. Each node performs a specific task such as data processing, reading sensor data, or actuating physical devices [6]. The event-driven architecture works well with IoT systems where real-world events trigger various processes and produce outputs [7]. For example, a typical IoT flow could involve a temperature sensor node that detects a value, compares it with given thresholds, and ends with an actuator node that regulates a thermostat. The visual representation makes it easier to understand and modify, while ensuring an efficient execution.

Despite all the advantages, event-driven architectures are prone to wrong message order and latency-related issues. In distributed systems, network delays as well as asynchronous processing might result in messages being delivered in the wrong order, leading to inconsistent system states [8]. For time-sensitive applications such as industrial automation or manufacturing control systems, the proper order of messages and predictable timing is crucial for ensuring correct and reliable operation [9]. One of the most popular queuing mechanisms - First-In-First-Out (FIFO) - does help with preserving the message order, yet it lacks when it comes to prioritising critical events or timing constraints needed for real-time IoT workflows [10]. Those limitations of Node-RED highlight the importance and need for more advanced scheduling mechanisms that enable deterministic control flow while maintaining the ease of use.

1.2 Problem: Latency for Critical Messages under FIFO Pipelines

Node-RED currently processes arriving messages using the First-In-First-Out (FIFO) queuing mechanism. It means that the messages arriving first are going to be processed first. It makes the whole process predictable and the chronological order maintained, which is very important in many IoT applications. Developers seeking a straightforward approach with deterministic behavior usually choose FIFO queuing when designing flows [9]. However, sequential processing might not be suitable in situations when the message's urgency or importance is at stake. Every message has to wait for previous ones to be processed before its turn.

The FIFO approach is not ideal when the less urgent and less critical messages are being processed before the more urgent ones. For example, in an industrial monitoring system that has multiple emergency alerts, these have to wait in a queue behind all the previously arrived messages. It becomes a huge problem when those important alerts notify operators of equipment overheating or a safety breach. The delays can cause serious problems, especially in situations when the urgency levels differ but the order of operations matters. In a manufacturing system, alerts about machine malfunctions must be processed immediately to prevent damage. However, if such alerts arrive after routine status updates, they may be processed too late [10]. In distributed systems, the problem is quite similar. Messages can arrive out of their logical order due to asynchronous processing, which can lead to inconsistent system states, causing high-priority actions to be delayed behind lower-priority operations [10].

It is important to prevent these delays, especially in industrial IoT applications, since they can result in safety hazards, equipment damage, and financial losses. In order to maintain safe and efficient operations, real-time control systems require response times to be under 100 milliseconds [8]. Urgent message delays in FIFO queues can cause control actions to occur too late, resulting in failures. In automated manufacturing any delayed emergency stop command could result in a safety hazard or damaged machinery. In building automation systems, the delayed alert about fire detection could compromise safety. These examples show that in time-sensitive applications, strict FIFO ordering does not take into consideration the varying criticality of different messages.

Current Node-RED cannot set message priority and importance within the

flow. FIFO queuing does ensure that messages maintain ordering, but it offers no means of defining scheduling criteria that take into account the real-time requirements of IoT applications. Users cannot mark which messages should be processed first or implement deadline-based scheduling. These restrictions limit the full potential use of Node-RED in time-sensitive industrial IoT, where deterministic, priority-aware message scheduling is essential. This gap motivates the work presented in this thesis.

1.3 Objectives and Research Questions

The primary objective of this thesis is to design and implement multiple message scheduling approaches for Node-RED to enable deterministic control flow in time-critical IoT applications. The specific objectives include:

- Designing and implementing static priority queue scheduling using binary min-heap data structures,
- Implementing EDF (Earliest Deadline First) scheduling for deadline-aware message handling,
- Developing laxity-based scheduling with dynamic priority recalculation as an extension of EDF,
- Comparing the performance of the static priority and EDF schedulers in terms of message latency, throughput, and ordering correctness relative to vanilla FIFO, with a systematic deadline-miss-rate comparison and an empirical laxity benchmark identified as future work,
- Identifying architectural limitations of Node-RED for real-time scheduling.

This thesis addresses the following research questions:

RQ1: How can priority-based, deadline-based, and laxity-based scheduling be integrated into Node-RED's event-driven architecture?

RQ2: What are the performance trade-offs between static priority and EDF scheduling, in terms of message latency, throughput, and ordering correctness, relative to vanilla FIFO?

RQ3: Under what conditions (message load and simulated processing time) does each scheduling approach perform best?

RQ4: What are the architectural constraints of Node-RED that affect real-time message scheduling implementation?

1.4 Contributions

The contributions of this thesis can be summarised as follows.

Contribution 1: Priority Queue Implementation. Designed and implemented a binary min-heap priority queue integrated directly into Node-RED's Flow.js runtime, supporting static priority levels for message classification with O(log n) insertion and extraction, evaluated across 6000 test runs, demonstrating correct priority ordering across all configurations.

Contribution 2: EDF Scheduling with Laxity Extension. Implemented Earliest Deadline First (EDF) scheduling for deadline-aware message handling, extended with laxity-based priority recalculation that dynamically adjusts message urgency based on remaining slack time, with modifications made directly to Node-RED's Flow.js.

Contribution 3: Performance Evaluation Framework. Developed a comprehensive evaluation methodology covering correctness, latency, throughput, overhead, and resource usage, conducting 6000 test runs across three schedulers, five message counts, and four simulated processing times, with CPU and memory measurements confirming suitability for constrained IoT hardware.

Contribution 4: Architectural Analysis. Identified key limitations of Node-RED for real-time scheduling, including the single-threaded event loop and sandboxed function nodes, documented trade-offs between scheduling flexibility and system complexity, and provided practical guidelines for Node-RED users and insights for future extensions.

All contributions preserve full backward compatibility with existing Node-RED flows, ensuring seamless integration into current deployments.

1.5 Thesis Structure

The remainder of this thesis is organized as follows.

Chapter 2 presents the background and related work, covering Node-RED's architecture and message pipeline, priority queue data structures, queues in event-driven environments, priority scheduling approaches, and existing solutions in Node-RED and IoT systems, along with their limitations.

Chapter 3 describes the research methodology, including the research process and paradigm, data collection approach, experimental design, hardware and software configuration, reliability and validity assessment, and the evaluation framework used to compare scheduling approaches.

Chapter 4 details the implementation of the scheduling system, outlining functional and non-functional requirements, system architecture, key classes and APIs, core algorithms, batching policies, overflow handling, backward compatibility, and configurable parameters.

Chapter 5 presents the experimental results, including scheduling correctness verification, latency and throughput analysis under varying message loads, overhead comparison against vanilla FIFO, resource usage measurements, and statistical analysis of the collected data.

Chapter 6 discusses the findings and their implications for IoT applications built on Node-RED, addresses limitations of the approach, analyses validity threats, and provides practical guidelines for Node-RED users selecting an appropriate scheduling approach.

Chapter 7 concludes the thesis by answering the research questions, summarizing the main contributions, and suggesting directions for future work, including hard real-time guarantees, dynamic priority adjustment, and distributed scheduling extensions.

Chapter 2

Background

2.1 Node-RED: Architecture, Flow.js, Message Pipeline

Node-RED provides a browser-based flow editor for wiring together hardware devices, APIs, and online services. The runtime environment uses Node.js, utilizing its event-driven architecture [11]. The architecture consists of three main components: the flow editor, the runtime, and the node library. Users interact with the editor to create flows through visual programming [12]. The runtime engine executes the deployed flows and handles message passing. Each node represents a reusable component that performs specific functions [13]. Flows are stored in JSON format, which defines nodes and connections [12]. During deployment, the editor sends the JSON configuration to the runtime over HTTP [14]. Message passing occurs through message payloads with properties like payload and topic [15]. The node library contains a vast collection of nodes covering different IoT protocols and services [5].

Flow.js is the core module that orchestrates flow execution by managing node instances and their interconnections throughout the deployment process. A running Node-RED configuration consists of multiple Flow objects where each Flow corresponds to an editor tab, with an additional global Flow object managing shared configuration nodes and subflow definitions [14]. The Flow class controls node instantiation, execution, and cleanup, managing their complete lifecycle from instantiation through execution to termination. During flow initialization after the user clicks deploy, all configured nodes are instantiated as Node objects and wired together according to the flow definition stored in JSON format [14]. When the Flow encounters nodes that reference subflow definitions, local copies of the subflow's internal nodes are created

and wired according to the subflow's internal connection definition [14].

In Node-RED, data flows between nodes in the form of message objects where each message is a JavaScript object with properties such as payload for data, topic for categorization, and additional custom fields [15]. Nodes receive incoming messages through registered event listeners, execute their configured logic to transform or process the data, and emit output messages that are automatically delivered to all connected downstream nodes [13]. Message delivery between nodes happens asynchronously on the Node.js event loop without blocking while waiting for recipients to process them, enabling concurrent message handling [13].

Node-RED's current architecture provides no built-in mechanism for message prioritization, processing all messages equally without distinguishing between critical control commands and routine data updates [12]. The FIFO constraint prevents the system from responding immediately to urgent events since critical operations like emergency stop commands or alarm notifications may be delayed by routine sensor readings [16]. This limitation creates a clear need for implementing priority-based message scheduling in Node-RED [5, 12].

2.2 Priority Queue Data Structures (Heaps, Hybrids)

A priority queue is a data structure where all elements have a priority assigned to them, and the ones with higher priority are processed first, before the low-priority ones. Unlike FIFO queue scheduling where elements are processed in arrival order, a priority queue dynamically orders elements based on their priorities [17]. It allows the system to perform three operations: push(x), which puts an element into the set, top(), which returns an element of either highest or lowest priority, and pop(), which deletes an element with either highest or lowest priority [17]. These data structures are used in algorithms such as Dijkstra's shortest path and real-time systems where the priority of specific tasks is important [17].

A binary heap is a tree structure that maintains the heap property when implementing priority scheduling [17]. A complete binary tree is fully filled level by level, where the last one is filled from left to right. There are two types of binary heaps: max-heaps, where parents are always bigger than their children, and the biggest number is at the root; and min-heaps, where parents are smaller than their children, and the smallest number is at the root [18].

Binary heaps enable quick operations, making them ideal for the implementation of a priority queue. The insert operation adds a new element and executes in O(log n) time, where n indicates the number of elements [17]. Extract-min removes and returns the element with the highest priority and also executes in O(log n) time [17]. Peek retrieves the element with the highest priority without deletion and executes in constant time since the top always contains the minimum element [17].

For this research, the implementation uses a binary min-heap to support priority-based message handling in Node-RED. The min-heap was chosen because it quickly extracts messages with the lowest priority values, where lower values indicate more urgent messages [18]. This implementation matches well with Node-RED's architecture, permitting priority-based insertion and extraction instead of FIFO order. The binary heap implementation provides efficient O(log n) performance, keeping overhead low for priority operations while providing deterministic behaviour in real-time IoT systems.

2.3 Queues in Event-Driven Environments (Node.js)

Node.js operates on a single-threaded event loop architecture, allowing it to process many concurrent requests efficiently. The event loop processes callbacks from various queues in a specific order as callbacks become ready for execution [19, 11]. Although Node.js runs on a single thread, concurrency is achieved through event-driven architecture. This model makes it suitable for real-time applications, including message brokers and IoT platforms [11, 16].

The Node.js event loop consists of multiple phases, with each phase responsible for particular operations. Key phases are the timers phase, the poll phase for I/O, and the check phase. The microtask queue contains async/await operations and Promise callbacks that run before moving to the next phase [19]. In the timers phase, scheduled timer callbacks are processed; the poll phase handles I/O operations; and the check phase runs setImmediate callbacks [11]. Each phase executes its callbacks synchronously in the order they were queued.

Event-driven architectures use callback queues to hold callbacks until they can be processed. By default, these queues operate on FIFO order with no built-in mechanism for prioritization [11, 16]. As a result, all messages receive equal treatment independent of their urgency - urgent callbacks must wait

behind routine operations if they arrive later [16]. Node.js provides no native way to prioritize critical operations, and time-consuming operations in the queue block the event loop, resulting in poor responsiveness for urgent events [11, 16]. Node-RED's event-driven architecture shares these constraints with the Node.js event loop [12, 16]. Crucially, because every callback is ultimately serialised on this single event loop, any scheduler built on top of it can influence message ordering but cannot provide hard real-time guarantees - only soft real-time behaviour - a constraint whose implications for the proposed scheduler are revisited in Sections 4.5 and 6.2.

2.4 Priority Scheduling (Preemptive/Non-preemptive) and Priority Inversion

Priority scheduling is a common scheduling algorithm used in operating systems and real-time systems, enabling systems to assign a priority to all tasks. Tasks with the highest priority are always executed first. The value is usually assigned based on various criteria such as processing time or memory requirements. There are two versions of priority scheduling - non-preemptive and preemptive. In the non-preemptive approach, even when a high-priority task arrives it has to wait for the current process to be completed. In the preemptive approach, when a low-priority process is running and a high-priority one arrives, the CPU automatically switches to the high-priority one [20].

Priority inversion happens when a high-priority task is blocked by a low-priority task holding a shared resource, such as memory or hardware access [21]. The low-priority task has to release this resource first. It becomes worse when a medium-priority task interrupts this process, leading to an indefinite waiting time for the high-priority task. A famous example is the Mars Pathfinder mission conducted in 1997, where, due to priority inversion, the lander was resetting repeatedly. One way to solve it is to use a priority inheritance protocol, where the low-priority task temporarily inherits the higher priority of the waiting task [21].

The choice between preemptive and non-preemptive scheduling involves several trade-offs. Non-preemptive scheduling methods are simpler to implement, require fewer resources, and have lower overhead, making it easier to predict scheduling outputs [22]. Disadvantages include the possibility of deadlock or an infinite loop causing system crashes [23]. Preemptive scheduling excels at improving average response time, avoiding deadlocks,

and ensuring CPU is not occupied by one process [22]. Disadvantages include time-consuming context switching and increased implementation complexity [23].

In Node-RED, messages are processed in sequential order using an event-driven non-preemptive execution model [12]. As mentioned in Chapter 1, Node-RED uses the FIFO scheduling method, which means that critical events have to wait for the routine operations to be finished before they start [10]. This limits Node-RED's applicability in time-critical IoT applications. This thesis demonstrates that priority scheduling is a worthwhile addition, helping Node-RED's usage in safety-critical and real-time architectures.

2.5 Existing Solutions in Node-RED/IoT and Gaps

The field of IoT real-time scheduling has been extensively studied and designed for different application scenarios [24]. Popular algorithms such as Earliest Deadline First (EDF) prioritize tasks based on which task has the closest deadline, ensuring all deadlines are met when feasible [25, 26]. Rate Monotonic Scheduling uses fixed priorities according to execution frequency, prioritizing tasks that execute more often [24]. The majority of proposed approaches operate at the system or network infrastructure level rather than application-level message prioritization [24].

The Node-RED platform provides scheduling solutions limited to time-based task scheduling rather than message prioritization [12]. Community nodes such as node-red-contrib-scheduler and node-red-contrib-bigtimer provide calendar-based flow triggering functionality without affecting message processing order [12]. No existing community contribution offers runtime message priority handling at the application level [12]. Users requiring priority handling must create custom solutions via external queueing systems [16]. These workarounds cannot alter the fundamental message delivery mechanism, reducing their reliability [14].

These solutions fall short for Node-RED's real-time IoT requirements [12, 16]. Scheduling nodes focus on when flows execute without addressing message-level priority during execution [12]. Without built-in priority support, users resort to complex workarounds involving custom code and state management, increasing complexity and reducing maintainability [12, 14]. For time-critical IoT applications requiring deterministic behaviour, these constraints limit applicability [12, 24].

This thesis addresses the gap by introducing priority queue functionality at the Node-RED flow execution level. Different from existing schedulers that address when flows execute, this work focuses on prioritizing messages during flow execution. The implementation modifies Node-RED's runtime architecture, enabling priority-based message handling [14]. By embedding prioritization within the core execution layer, the solution enables deterministic message handling for time-critical IoT applications [12]. In contrast to middleware-based approaches, this implementation requires no additional infrastructure [9, 16].

Chapter 3

Method

This chapter describes the research methodology employed in this thesis. Section 3.1 describes the research process. Section 3.2 details the research paradigm. Section 3.3 focuses on the data collection techniques used for this research. Section 3.4 describes the experimental design. Section 3.5 explains the techniques used to evaluate the reliability and validity of the data collected. Section 3.6 describes the method used for the data analysis. Finally, Section 3.7 describes the framework selected to evaluate the scheduling approaches.

3.1 Research Process

The research adopts a Design Science Research (DSR) framework, commonly used in research that combines system design with empirical evaluation. DSR emphasizes solving well-defined problems through artifact creation, using iterative cycles of design, implementation, and evaluation. The research process consisted of four distinct phases. First, a literature review was conducted to explore scheduling algorithms, data structures, and related work in flow-based environments. In the second phase, the research problem was structured by defining message models, priority classes, and performance metrics applicable to flow-based IoT systems. Third, two scheduling approaches were built and integrated into Node-RED: a static priority queue and an Earliest Deadline First (EDF) scheduler, where laxity was further incorporated as an extension to the EDF approach. Finally, each artifact was evaluated using a series of controlled performance tests measuring correctness, latency, throughput, and overhead.

3.2 Research Paradigm

This research is grounded in a pragmatic research philosophy given that the focus is on building a functional system rather than purely theoretical contributions. This is consistent with the artifact-driven nature of DSR, where success is measured by the utility and performance of the artifact. The objective is not to generalize findings across all scheduling systems but to develop and assess a practical solution within the context of flow-based IoT systems.

3.3 Data Collection

This section presents how data was collected to assess the performance of the implemented schedulers.

3.3.1 Sampling

The test messages used in this study were classified into three priority categories corresponding to priority values 1–33, 34–66, and 67–100. These categories represent real-world IoT message traffic where different message types require different levels of urgency. Messages were assigned a numerical priority value at the time of creation, allowing the scheduling system to maintain correct ordering.

3.3.2 Sample Size

The final evaluation was designed across five message counts — 500, 1000, 1500, 2000, and 2500 messages — repeated over 100 iterations per configuration. This range was chosen to assess scheduler performance under increasing load while remaining within the memory constraints of the test hardware. Four simulated processing times were applied — 0ms, 50ms, 100ms, and 200ms — to evaluate how scheduling behaviour changes with varying application workloads. Correctness tests used a separate configuration of 500 messages divided across five priority or deadline buckets of 100 messages each, repeated over 30 iterations.

3.3.3 Target Population

The target population of this study consists of flow-based IoT applications built on Node-RED, where messages with varying levels of urgency are processed. These environments commonly feature scenarios where critical sensor alerts, control commands, and routine status updates coexist. This study focuses specifically on linear Node-RED flows without branching or parallel execution paths to maintain a controlled and reproducible evaluation environment. Typical application domains include smart building management, industrial process control, and connected healthcare devices where sensor alerts and control commands must be prioritised over routine telemetry.

3.4 Experimental Design and Planned Measurements

This section presents the experimental design employed to measure scheduler performance.

3.4.1 Test Environment

Testing was carried out on a controlled virtual machine environment to ensure reproducibility and eliminate external interference. The test environment consisted of an Ubuntu 22.04 LTS operating system with 4GB of RAM and 2 virtual processors. Node-RED 3.1 was deployed alongside Node.js version 18, representing the stable release available at the time of implementation. Test flows were triggered via inject nodes configured to simulate IoT message traffic, with priority values and deadlines assigned at generation time.

3.4.2 Hardware and Software

Testing was carried out on a virtual machine running on a personal laptop computer, providing a controlled environment to ensure reproducibility and eliminate external interference. The test environment consisted of an Ubuntu 22.04 LTS operating system with 4GB of RAM and 2 virtual processors, demonstrating that the solution does not depend on high-performance infrastructure. The primary tools used included the Node-RED 3.1 platform running on the Node.js 18 runtime with Flow.js modified to integrate the priority scheduling logic. Output data was recorded via a custom statistics API built into the scheduler and saved as CSV files to enable correctness

verification and performance analysis. Data analysis and visualisation were performed through Python scripts for statistical analysis and graph generation.

3.5 Assessing Reliability and Validity

This section discusses measures taken to ensure the validity and reliability of the experimental results.

3.5.1 Validity of Method

The research method was validated through careful alignment between the defined research questions and the evaluation approach. The test scenarios were designed to address the core scheduling properties of correctness, latency, throughput, and deadline adherence. Following DSR principles ensured that the artifact is evaluated against practical criteria rather than purely theoretical ones. A potential threat to validity is the use of inject-node-generated messages rather than actual IoT device output, which may not fully capture the variability of real-world IoT traffic patterns.

3.5.2 Reliability of Method

The method was designed to be repeatable through the use of a fixed and documented test environment so that experiments could be repeated under identical conditions. All test inputs were synthetically generated, eliminating variability introduced by manual input or external data sources. The Node-RED flow configurations used in testing are documented and available for replication.

3.5.3 Data Validity

The validity of collected data was maintained by exporting all processed messages to CSV and manually verifying correct priority ordering. Additionally, the built-in statistics API provided independent verification of throughput, latency, and overflow behaviour, allowing cross-validation of results. All test scenarios achieved perfect message ordering in tests up to 1000 messages, with the overflow test confirming expected drop behaviour.

3.5.4 Reliability of Data

The reliability of the collected data was ensured by executing each experiment consistently within the same environment, confirming consistent results across runs. The agreement between data from the CSV files and the statistics module provided additional confidence in the accuracy of the collected data. No irregularities were detected in the collected data, suggesting that the data collection process was stable and dependable.

3.6 Planned Data Analysis

This section describes the data analysis techniques and software tools used to process and interpret the experimental results.

3.6.1 Data Analysis Technique

The experimental results were evaluated by comparing key performance metrics across the implemented scheduling approaches (static priority and EDF, with laxity as an extension of EDF). Key performance indicators examined were ordering accuracy, end-to-end latency, and messages processed per second. Deadline miss rate was defined as an intended metric, but a systematic measurement is identified as future work (Section 6.2). Performance data from each test run were compared to identify trends in scalability and performance degradation. Priority ordering was confirmed by inspecting the sequence of messages in the CSV exports.

3.6.2 Software Tools

Data processing and visualisation were performed through Python scripts utilising pandas for data manipulation and matplotlib for graph generation. CSV files were inspected using Microsoft Excel for manual verification. During development and testing, the Node-RED debug panel provided real-time visibility into message routing and scheduling behaviour.

3.7 Evaluation Framework

The evaluation framework was designed to provide a structured basis for comparing the implemented scheduling mechanisms using a consistent set of metrics applied to all approaches. The framework consisted of four core

metrics: ordering correctness, average message latency, message throughput, and deadline miss rate. Message ordering correctness refers to the proportion of messages delivered in the order dictated by the scheduling policy. Latency was measured as the average time elapsed between a message entering the queue and being delivered. The deadline miss rate metric captured how many messages exceeded their deadline relative to total messages processed. While the framework was designed to include deadline miss rate as a core metric, the empirical evaluation reported in Chapter 5 focuses on ordering correctness, latency, throughput, and overhead. A systematic deadline-miss-rate characterisation under controlled overload conditions is identified as future work. The implemented approaches were assessed under identical test conditions with the same message volumes and priority distributions, allowing meaningful side-by-side evaluation of scheduling performance.

3.8 System Documentation

Documentation of the developed system was carried out across all phases of development using inline comments within the modified Flow.js source and the written thesis. The modified Flow.js source code was annotated to explain the scheduling logic, priority queue operations, and batch processing mechanisms. The written thesis constitutes the primary documentation artefact, providing a complete account of the system from design through evaluation. The modified Flow.js source code, HybridPriorityQueue, and MessageScheduler implementations are available at: `https://github.com/DrelaDominika/node-red-priority-scheduler`. The Node-RED test flow configurations used in the evaluation are provided in Appendix A. Code extracts and pseudocode are included in Appendix B. Extended result tables are available in Appendix C.

Chapter 4

Implementation

This chapter details the design and implementation of the priority-based message scheduling system for Node-RED.

4.1 Functional and Non-Functional Requirements

The primary functional requirement was to intercept and reorder messages based on assigned priority values, replacing the default FIFO message handling. The implementation had to accommodate multiple scheduling policies, including static priority and EDF, where laxity-based scheduling was implemented as an extension of the EDF approach. The system had to maintain compatibility with flows that do not use priority scheduling. Additionally, the implementation had to cap the queue at a maximum capacity and drop excess messages in a controlled manner.

In terms of performance, the scheduler had to maintain low per-message processing latency even under high message volumes. A key non-functional requirement was to ensure that scheduling operations did not interfere with Node-RED's single-threaded event loop by processing messages in batches and yielding control between batches. The solution had to follow a modular design to facilitate future extension and maintenance.

4.2 Architecture

This section describes the high-level design of the implemented scheduling solution. The solution is structured around three main components:

Flow.js, the MessageScheduler, and the HybridPriorityQueue. The outermost layer, Flow.js, acts as the routing layer responsible for directing messages, determining whether each message carries a priority value, and routing it accordingly. The MessageScheduler is responsible for batch processing and controlling the flow of messages from the queue. The innermost component, the HybridPriorityQueue, stores and orders messages according to the active scheduling policy. This layered structure enables priority-based message processing without disrupting existing flow behaviour.

4.3 Classes and APIs

The following section outlines the classes and APIs that form the building blocks of the scheduling solution. The HybridPriorityQueue class encapsulates the binary min-heap logic and exposes methods for message insertion and retrieval. Its primary methods include insert() for adding messages, extractMin() for retrieving the highest priority message, and peek() for inspecting the next message without removing it, operating at O(log n) for insert and extract operations and O(1) for peek. Peek is constant time because the highest-priority element always resides at the root of the min-heap and can be read directly, without any traversal or reordering.

The second core class, MessageScheduler, manages the lifecycle of the priority queue and coordinates message dispatch. Key responsibilities include methods for enqueueing messages, processing batches via setImmediate(), and exposing a statistics API. These two classes work together, enabling Flow.js to delegate priority message handling to a dedicated scheduling layer. The active scheduling algorithm is selected through a configurable comparison function injected into the HybridPriorityQueue, where static priority compares numerical priority values, while EDF compares message deadlines.

4.4 Pseudocode of Core Algorithms

This section describes the core algorithms implemented in the scheduling system using pseudocode.

Algorithm 1 describes the insert operation of the binary min-heap, ensuring the heap remains correctly ordered after every insert.

Algorithm 1: Heap Insert

Input: message msg

1. Add msg to end of heap array
2. Set i = last index
3. While i $>$ 0 and compare(heap[parent(i)], heap[i]) $>$ 0:
   (a) Swap heap[i] with heap[parent(i)]
   (b) Set i = parent(i)
4. Return

Algorithm 2 describes the extraction of the highest priority message from the HybridPriorityQueue, which restores the heap property after removal.

Algorithm 2: Heap ExtractMin

Input: none *Output:* message with highest priority

1. Save heap[0] as result
2. Move last element to heap[0]
3. Remove last element
4. Set i = 0
5. While left child of i exists:
   (a) Find smallest child
   (b) If compare(heap[i], smallest child) $>$ 0: swap heap[i] with smallest child and set i = index of smallest child
   (c) Else break
6. Return result

Algorithm 3 illustrates how the scheduling policy is applied through configurable comparison logic.

Algorithm 3: Comparison Functions

Static Priority Comparison — Input: messages a, b

1. If a.priority $<$ b.priority: return -1

2. If a.priority $>$ b.priority: return 1
3. Else: return 0

EDF Comparison — Input: messages a, b

4. If a.deadline $<$ b.deadline: return -1
5. If a.deadline $>$ b.deadline: return 1
6. Else: return 0

Algorithm 4 presents the laxity calculation used to dynamically adjust message priority within the EDF framework, with laxity representing the slack time available before a message misses its deadline.

Algorithm 4: EDF with Laxity Extension

Input: message msg, current time t *Output:* laxity value

1. Calculate remainingDeadline = msg.deadline $-$ t
2. Calculate remainingExecution = msg.executionTime
3. Calculate laxity = remainingDeadline $-$ remainingExecution
4. If laxity $<$ 0: message has missed deadline
5. Return laxity

Laxity Comparison — Input: messages a, b, current time t

6. Calculate laxity(a) and laxity(b)
7. If laxity(a) $<$ laxity(b): return -1
8. If laxity(a) $>$ laxity(b): return 1
9. Else: return 0

Together, these algorithms provide the foundation for all scheduling operations within the system, supporting static, EDF, and laxity-based scheduling through a unified heap structure.

4.5 Batching Policies and Event-Loop Interaction

The following section outlines how messages are processed in batches to avoid blocking Node-RED's single-threaded event loop. Since Node-RED runs on a single-threaded event loop inherited from Node.js, any synchronous processing of large message volumes would freeze the interface. To address this, messages are dispatched in configurable batches of 100 at a time, using setImmediate() to yield control back to the event loop between batches. setImmediate() schedules a callback to execute on the next iteration of the event loop, after I/O events are processed, allowing other pending operations to run before the next batch is dispatched.

The batching strategy ensures that scheduling operations do not starve other processes of execution time while still maintaining high throughput. The batch size of 100 was chosen as it provided a balance between throughput and responsiveness during testing. Once a batch has been dispatched, the system evaluates whether further messages are awaiting dispatch and schedules the next batch via another setImmediate() call if necessary. While batching introduces a slight delay compared to immediate single-message processing, this trade-off between latency and responsiveness was validated during testing.

4.6 Overflow Handling

This section explains the approach taken to handle message overflow when the queue is full. The HybridPriorityQueue is configured with a configurable capacity limit set to 1000 messages by default, ensuring memory usage remains within acceptable bounds. Upon reaching the maximum capacity, incoming messages are dropped while the statistics tracker increments the dropped message counter. Discarding overflow messages was selected rather than blocking incoming messages or expanding the queue dynamically because it preserves event-loop responsiveness and avoids memory exhaustion. The overflow protection was tested via the largest test scenario involving 1200 messages, verifying correct overflow behaviour under load.

4.7 Concurrency and Backward Compatibility

Since Node.js operates on a single-threaded model, race conditions on the queue are inherently avoided. This means no explicit locking mechanisms or synchronisation primitives were required. Existing Node-RED flows are supported through a hybrid routing approach where non-priority messages bypass the scheduler entirely and follow the standard Node-RED FIFO pipeline, so their processing is unaffected by the priority queue. This approach allows priority scheduling to be introduced into existing deployments without modification, ensuring no disruption to currently deployed Node-RED applications.

Validation confirmed that all existing flows without priority metadata continued to execute correctly and in the expected order after the scheduler was integrated. This combination of concurrency safety and backward compatibility provides a robust and deployment-friendly scheduling solution. The scheduler exposes three configurable parameters: queue capacity (default 1000 messages), batch size (default 100 messages per batch), and scheduling algorithm selection (static priority, EDF, or laxity-based), allowing behaviour to be adjusted without structural changes to the implementation.

Chapter 5

Results

5.1 Scheduling Correctness

Each test flow consisted of an inject node that triggered the test, an initialisation function node that built a queue of all test configurations, and a run function node that stepped through each configuration iteratively. A generator function node created messages with injected timestamps and assigned either random priority values (1-100) for the static priority scheduler or random deadline values (50-300ms) for the EDF scheduler, with no priority metadata for vanilla FIFO tests. Messages passed through an application processing node simulating workload, followed by a collector function node that recorded arrival order and computed per-message latency. A 50ms delay node separates iterations to allow the event loop to stabilise between runs. On completion, a statistics function node aggregated results across all 30 iterations and saved them as JSON files for subsequent analysis in Python.

The correctness of both schedulers was evaluated through dedicated experiments. Each test consisted of 500 messages divided across five distinct priority levels, with 100 messages per bucket. The expected outcome is that messages with higher urgency consistently arrive at the collector before lower priority ones. The measured latencies per priority bucket are shown in Figure 5.1.

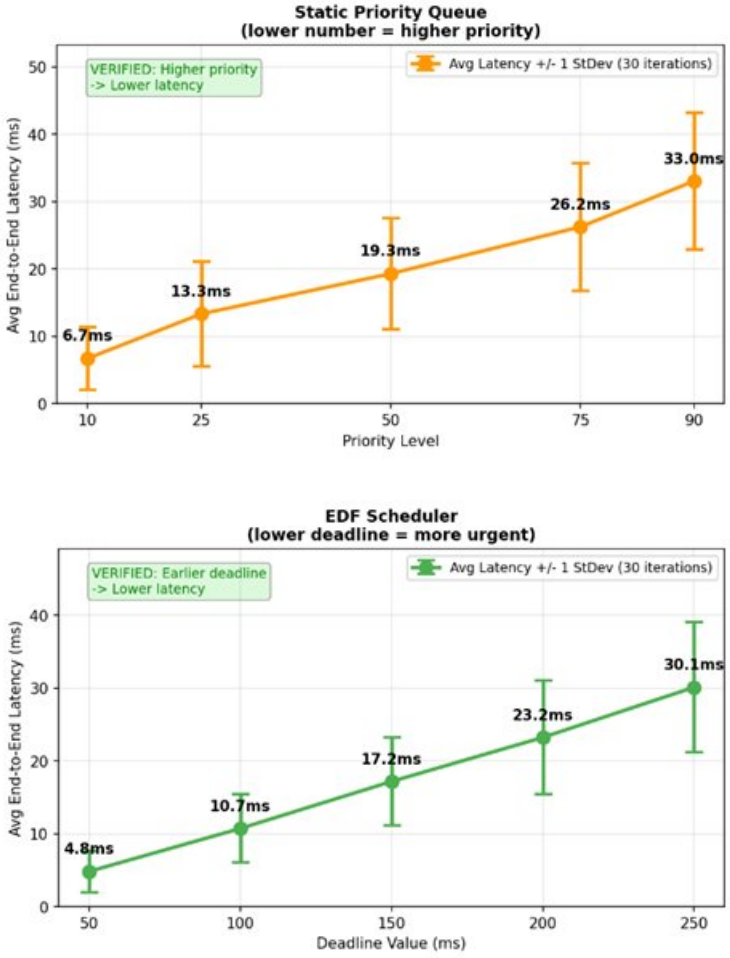

Figure 5.1: Correctness verification - average end-to-end latency per bucket for Static Priority Queue (top) and EDF (bottom). 30 iterations, 100 messages per bucket.

Messages assigned priority 10 achieved an average latency of 6.7ms, compared to 33.0ms for priority 90. The nearly fivefold difference in latency between the highest and lowest priority levels demonstrates that the scheduler enforces ordering as intended. For the EDF scheduler, messages were assigned deadline values between 50ms and 250ms across five buckets. Messages with a deadline of 50ms achieved an average latency of 4.8ms, while those with deadline of 250ms averaged 30.1ms. This confirms that the EDF scheduler correctly prioritizes messages with earlier deadlines. In summary, both scheduling approaches correctly enforce their respective ordering criteria across all tested configurations.

5.2 Latency and Throughput vs Load

This section presents the performance results of both schedulers under varying message loads. Tests were repeated 100 iterations per configuration, using a 50ms simulated processing delay to represent realistic workloads. Figures 5.2 and 5.3 present the total processing time, average latency, throughput, and overhead for the static priority queue.

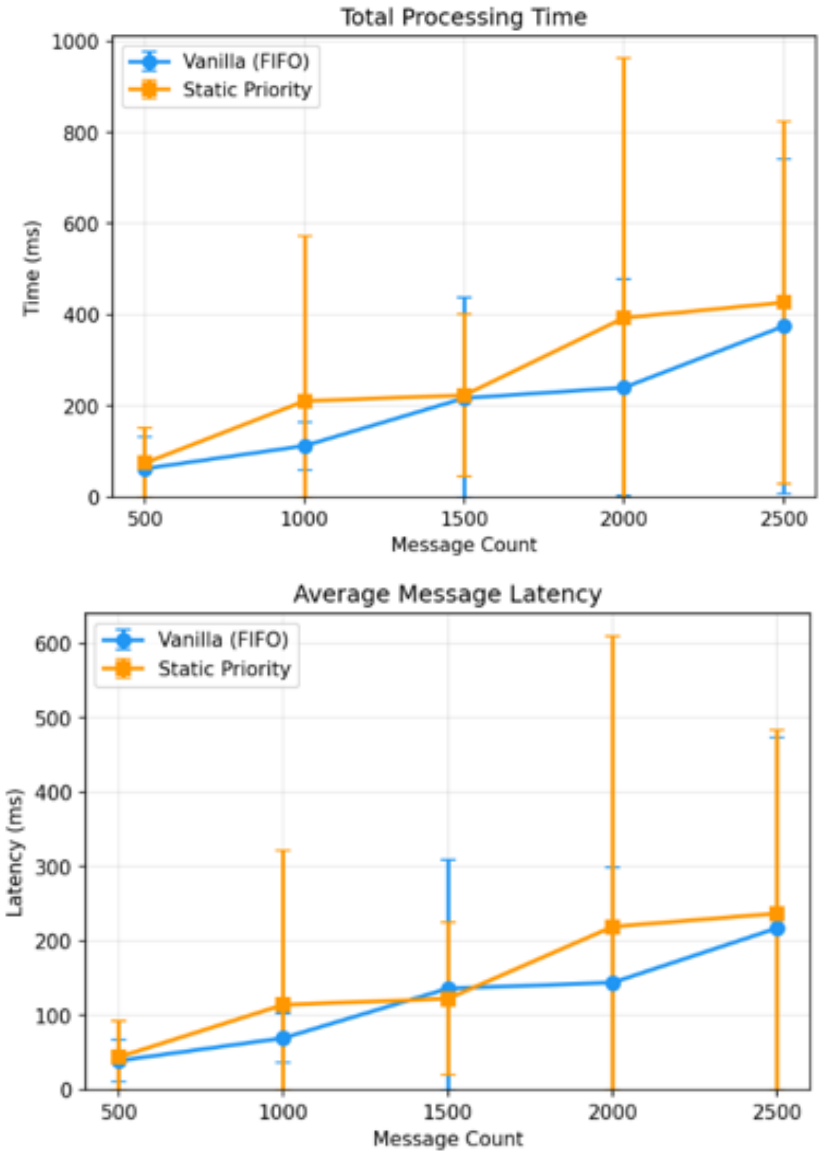

Figure 5.2: Static Priority Queue - total processing time (top) and average message latency (bottom). 50ms app delay, 100 iterations.

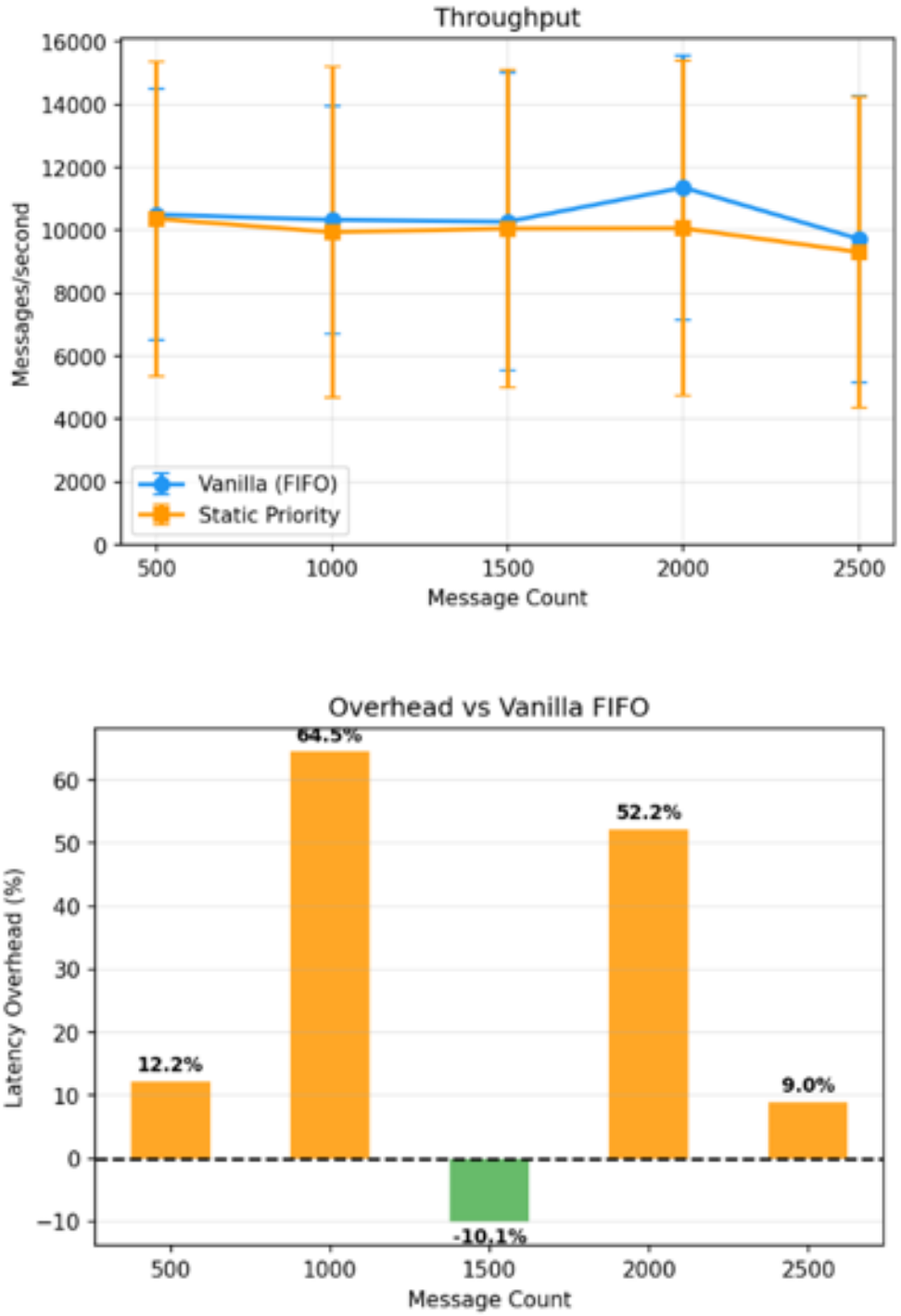

Figure 5.3: Static Priority Queue - throughput (top) and overhead vs vanilla FIFO (bottom). 50ms app delay, 100 iterations.

Total processing time increases with message count, rising from approximately 75ms at 500 messages to 427ms at 2500. The stable throughput indicates the priority queue introduces no significant bottleneck as load increases. The static priority queue overhead varies considerably across message counts; however, given the large variance observed across all configurations, this variation is attributable to environmental non-determinism in the Node.js event loop rather than a systematic dependency on message count. The EDF scheduler was evaluated under the same conditions, with results shown in Figures 5.4 and 5.5.

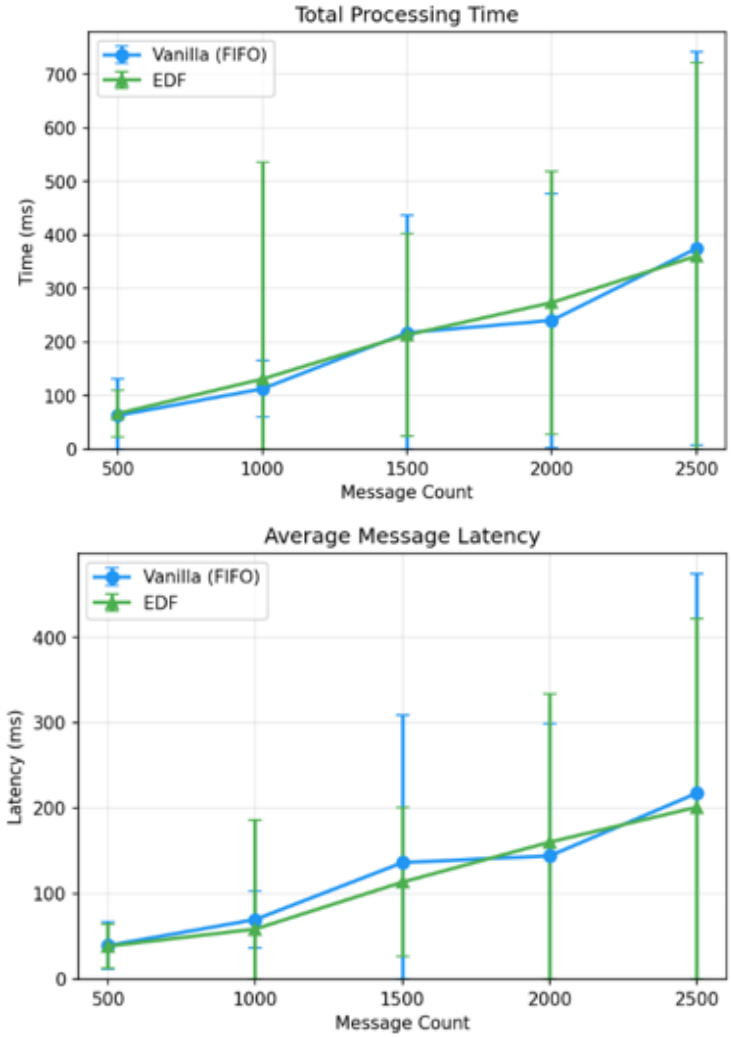

Figure 5.4: EDF Scheduler - total processing time (top) and average message latency (bottom). 50ms app delay, 100 iterations.

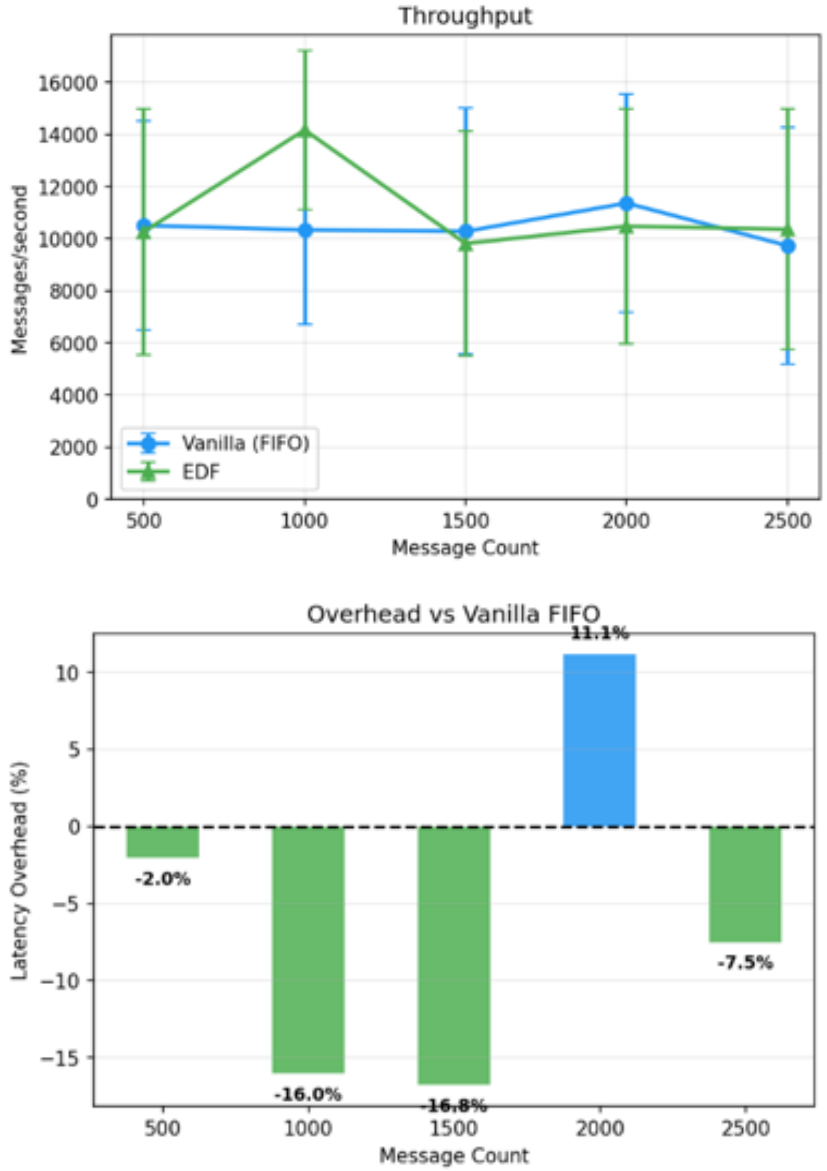

Figure 5.5: EDF Scheduler - throughput (top) and overhead vs vanilla FIFO (bottom). 50ms app delay, 100 iterations.

The EDF scheduler shows comparable throughput to vanilla FIFO, remaining between 9800 and 14000 msg/s depending on load. The EDF scheduler shows predominantly negative overhead at most message counts, suggesting it occasionally outperforms vanilla FIFO due to more efficient batch dispatching. Overall, both schedulers demonstrate predictable scaling behaviour with no breakdown observed up to 2500 messages.

5.3 Overhead vs Vanilla FIFO

This section examines the latency overhead introduced by each scheduler relative to vanilla Node-RED FIFO processing. Figures 5.6-5.9 present the overhead for each processing time configuration separately.

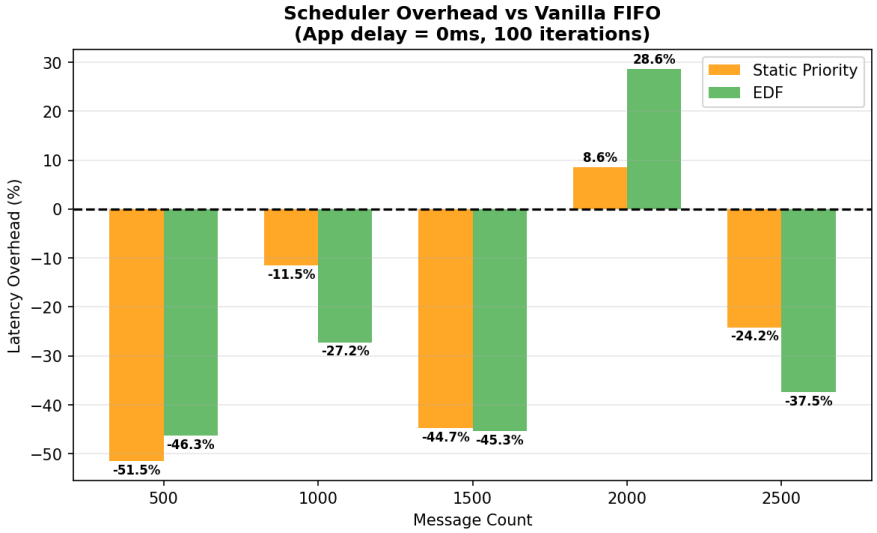

Figure 5.6: Scheduler overhead vs vanilla FIFO - 0ms app delay, 100 iterations.

The results show that neither scheduler consistently introduces high overhead, with many configurations performing at or below vanilla latency. Both schedulers show variable overhead across configurations; however, this variability is consistent with the non-deterministic behaviour of the Node.js event loop and should not be interpreted as a systematic trend. The EDF scheduler shows predominantly negative or near-zero overhead across most configurations, frequently matching or outperforming vanilla FIFO. Despite the variance, neither scheduler consistently degrades performance beyond vanilla FIFO, supporting their suitability for IoT deployments.

5.4 Resource Usage (CPU/Memory)

To assess the resource footprint of the scheduling system, CPU and memory consumption were monitored throughout testing. CPU and memory were recorded at two-second intervals throughout the entire test execution. The monitoring ran continuously alongside the test, capturing 180 data points over

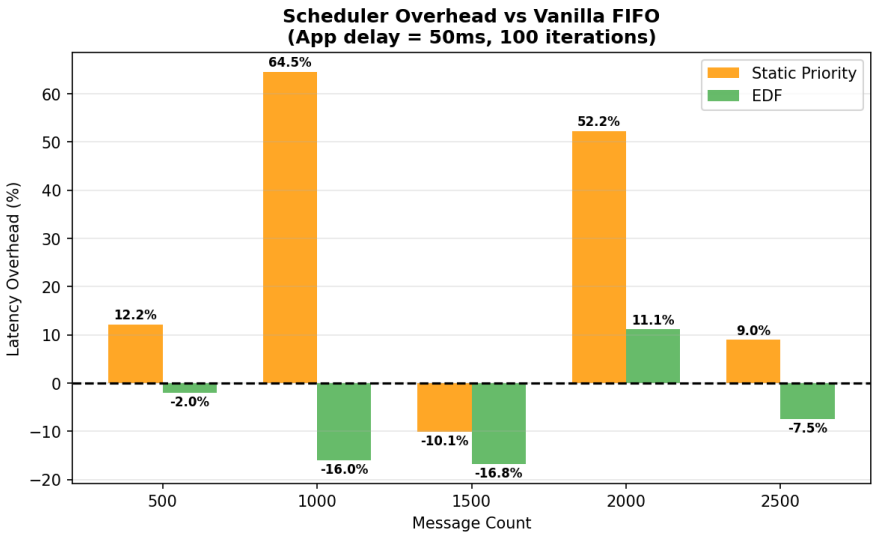

Figure 5.7: Scheduler overhead vs vanilla FIFO - 50ms app delay, 100 iterations.

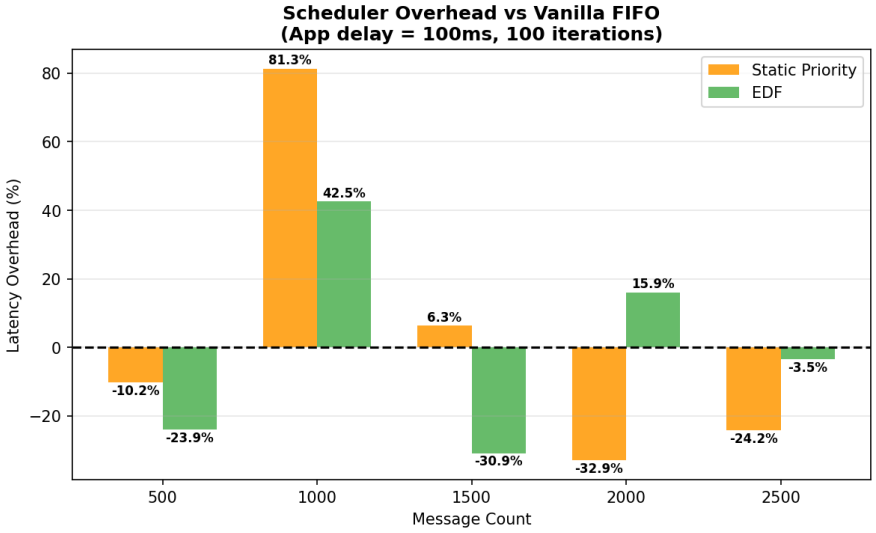

Figure 5.8: Scheduler overhead vs vanilla FIFO - 100ms app delay, 100 iterations.

371 seconds. Figure 5.10 illustrates how CPU and memory usage evolved over the test duration.

CPU usage remained consistently low throughout the test, averaging 5.6%

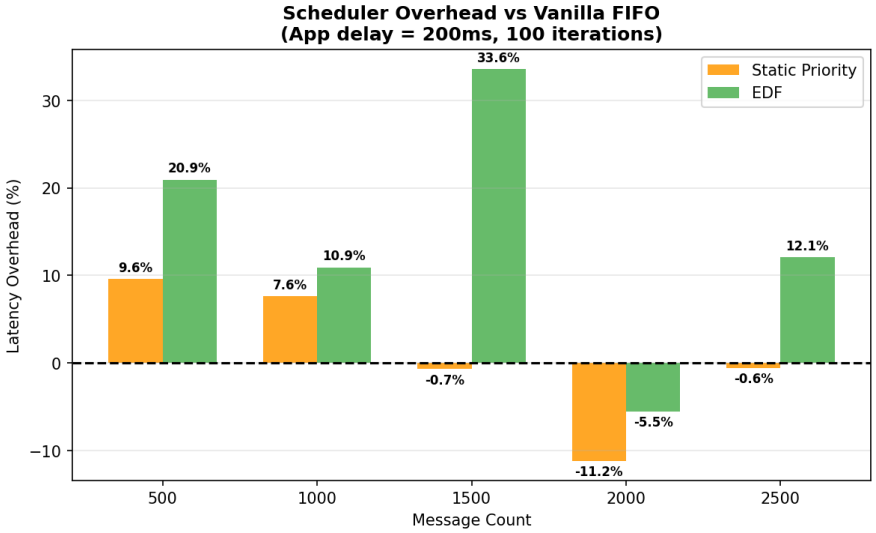

Figure 5.9: Scheduler overhead vs vanilla FIFO - 200ms app delay, 100 iterations.

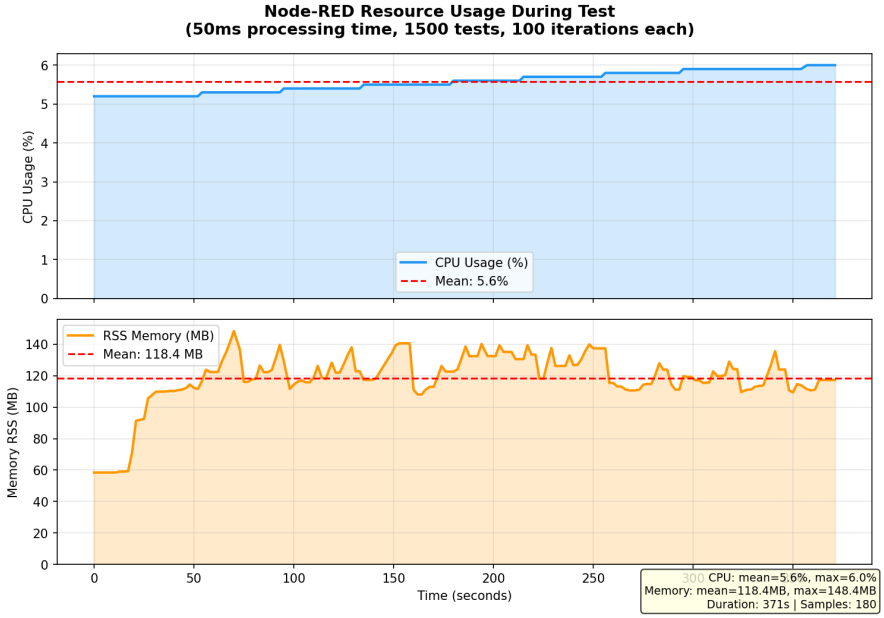

Figure 5.10: Node-RED CPU and memory usage during a full 50ms test run. 180 samples over 371 seconds, sampled every 2 seconds.

with a peak of 6.0%. Such stable and low CPU usage indicates that the

priority queue and EDF operations are computationally lightweight. At the beginning of the test run, memory consumption stood at approximately 60MB. Over the course of the test, memory usage rose to a peak of 148MB, after which it remained stable. The gradual memory increase reflects the accumulation of iteration results in Node-RED's flow context during testing. With a maximum of 148MB, the memory overhead remains acceptable for most Node-RED deployment scenarios. These results demonstrate that the scheduling system operates within acceptable resource limits for typical IoT hardware. It should be noted that a direct comparison against a baseline vanilla Node-RED instance without the scheduler additions was not performed, meaning the exact memory and CPU overhead introduced specifically by the HybridPriorityQueue and MessageScheduler cannot be isolated from the results. This is acknowledged as a limitation of the resource evaluation.

5.5 Statistical Analysis

This section summarises the statistical properties of the collected measurements across all test configurations. A total of 6000 test runs were collected, with 100 iterations per configuration across four processing times and five message counts. The results exhibit relatively high standard deviations, particularly at lower processing times such as 0ms and 50ms. Such variance is expected in event-driven runtimes, where garbage collection, timer resolution, and OS scheduling all affect message timing. With 100 iterations, mean values converge to stable estimates even where individual runs show significant deviation.

The 0ms processing time configuration exhibits the highest variance, as there is no dominant processing delay to stabilise timing. At 200ms processing time, standard deviations are substantially lower, reflecting the dominance of processing time over runtime noise. Despite high latency variance, throughput values are relatively consistent, suggesting the system processes messages at a stable rate overall. Both schedulers produce comparable levels of variability, suggesting that the source of variance is the runtime environment rather than the scheduling algorithm. Vanilla FIFO shows similar variance to both schedulers, confirming that the variability originates from the Node.js runtime rather than the scheduling overhead.

Correctness is treated as a binary outcome - either the scheduler enforces the correct message ordering, or it does not. Both schedulers passed all 30 correctness iterations without a single ordering violation, confirming that priority and deadline ordering are reliably enforced regardless of runtime

conditions. With 100 iterations, the means are sufficiently stable to draw valid conclusions about relative scheduler performance. Testing on a single local machine introduces a potential validity threat, as performance characteristics may differ on embedded or cloud-hosted Node-RED deployments. The combination of high iteration counts, stable correctness results, and consistent vanilla baselines provides confidence in the reliability of the evaluation findings.

Chapter 6

Discussion

This chapter interprets the results presented in Chapter 5 and discusses their implications for IoT applications built on Node-RED.

6.1 Interpretation and Practical Implications for IoT Flows

The primary finding is that priority and deadline-based ordering can be successfully integrated into Node-RED without breaking existing flows. In practice, this means Node-RED can now be used in scenarios where message ordering matters, such as industrial monitoring or healthcare systems. The performance results show that both schedulers introduce manageable overhead relative to vanilla FIFO processing.

The static priority queue provides reliable ordering with low configuration overhead, making it suitable for most IoT deployments. The EDF scheduler is more appropriate for time-critical applications where messages carry explicit deadline requirements.

The high variance observed in performance measurements is a known characteristic of the Node.js event loop and does not reflect a flaw in the scheduling design. Developers deploying these schedulers should be aware that individual message latencies may vary significantly from the mean. Since the scheduler is integrated at the Flow.js level, existing Node-RED deployments can benefit from priority scheduling without any flow reconfiguration. Overall, the results support the conclusion that priority-based message scheduling is a viable and practical extension to Node-RED for IoT applications.

6.2 Limitations

This section identifies the key limitations of the implemented scheduling system and the evaluation methodology. Testing on a single machine means that performance results may differ significantly on lower-powered IoT gateway hardware. The test workload consisted of synthetically generated messages rather than real IoT sensor data. Real IoT traffic may differ in several concrete ways: message arrival patterns are often bursty rather than uniform, message sizes vary depending on sensor payload, and network jitter can introduce irregular inter-arrival times. These characteristics may affect scheduler behaviour under realistic conditions in ways not captured by the synthetic workload used here. In addition, message priorities were drawn from a uniform random distribution across priority bands, whereas real IoT traffic is typically skewed - dominated by routine telemetry with only occasional urgent messages - which is precisely the regime in which priority scheduling is expected to provide the greatest benefit.

The single-threaded nature of Node.js restricts the scheduling system to sequential message processing, potentially limiting performance under very high loads. Due to event loop variability, the schedulers offer soft real-time ordering rather than strict deadline guarantees. Despite these limitations, the evaluation provides a solid foundation for understanding the behaviour of priority scheduling in Node-RED. The chosen metrics - latency, throughput, and ordering correctness - are standard measures in scheduling research, supporting construct validity. Although the evaluation framework included deadline miss rate as an intended metric, this thesis reports ordering correctness, latency, throughput, and overhead. Deadline miss rate was not separately quantified, as a meaningful measurement requires controlling feasibility by setting deadlines relative to processing time; a systematic miss-rate study under overload is left to future work.

6.3 Guidelines for Node-RED Users

Based on the evaluation findings, the following guidelines are offered to Node-RED users selecting an appropriate scheduling approach. Developers should choose the static priority queue when message urgency is known in advance and can be expressed as a numeric priority. Examples include industrial monitoring systems where alarm messages must be processed before routine status updates. The EDF scheduler is recommended for applications where

messages carry explicit timing requirements and earlier deadlines must be honoured first. Examples include industrial automation flows where control commands must be executed before their deadlines expire.

If all messages in a flow have equal importance, the overhead of priority scheduling is unnecessary and vanilla Node-RED is preferable. Regardless of scheduler choice, developers should set the queue capacity to at least twice the expected peak message volume to avoid overflow. The scheduling benefit is most visible at higher processing times - as shown in Figures 5.8 and 5.9, both schedulers maintain overhead close to or below vanilla FIFO at 100ms and 200ms processing times, while the correctness results in Figure 5.1 confirm consistent priority ordering across all tested configurations. Developers with fast-processing flows operating at 0ms or 50ms may therefore see limited practical ordering improvement due to event loop timing noise dominating over scheduling effects. The scheduler is fully backward compatible, allowing developers to enable priority scheduling without changing their existing flow configurations.

Chapter 7

Conclusions and Future Work

7.1 Answers to the Research Questions

The following summarises how the results and implementation address each of the research questions. Priority, deadline, and laxity-based scheduling were integrated into Node-RED by embedding a HybridPriorityQueue and MessageScheduler directly into Flow.js. This approach required no changes to existing flows, preserving backward compatibility while enabling priority-aware message handling across the entire runtime.

The performance evaluation revealed that the static priority queue and EDF scheduler exhibit similar throughput to vanilla FIFO, with overhead varying across configurations due to environmental non-determinism in the Node.js event loop rather than a systematic effect of message count. The correctness results confirm that higher priority messages consistently achieve lower latency, demonstrating that priority ordering is achieved without significant throughput degradation. Laxity-based scheduling was implemented as an extension of EDF, sharing the same queue and dispatch mechanism, which demonstrates the architecture's extensibility. A full empirical benchmark of the laxity variant, including an explicit deadline-miss-rate comparison against EDF, remains future work.

The static priority queue is most effective when messages have predefined importance levels, while EDF performs best when messages carry explicit timing deadlines. The scheduling benefit is most visible at processing times of 100ms and above, where the ordering effect dominates over event loop timing noise.

The main architectural constraints identified were the single-threaded event loop, which prevents parallel message processing, and the sandboxed

function nodes, which limit direct scheduler access. These constraints are discussed in detail in Sections 4.5 and 6.2. The single-threaded constraint limits scheduling to sequential batch processing, making hard real-time guarantees unachievable within the current Node.js architecture. The four research questions have been answered, demonstrating that priority-based scheduling is both feasible and practical within Node-RED's existing architecture.

7.2 Main Contributions

This thesis makes four main contributions to the design and implementation of priority-based scheduling in Node-RED. The first contribution is the design and implementation of a binary min-heap priority queue integrated directly into Node-RED's Flow.js runtime. The implementation supports O(log n) insertion and extraction, ensuring efficient operation even under high message volumes. The second contribution is the implementation of an Earliest Deadline First scheduler enabling deadline-aware message handling in Node-RED, extended with laxity-based priority recalculation for dynamic adjustment of message urgency based on remaining slack time.

The third contribution is a comprehensive evaluation framework covering correctness, latency, throughput, overhead, and resource usage. The fourth contribution is an architectural analysis of Node-RED's constraints for real-time scheduling, identifying the event loop and sandboxed function nodes as key limitations.

All contributions preserve full backward compatibility, meaning the scheduling system can be integrated into existing Node-RED environments without disruption. Together these contributions extend Node-RED's capabilities for time-sensitive IoT applications while preserving its ease of use.

7.3 Extensions

The following describes potential extensions that would further improve the scheduling system beyond the scope of this thesis. A key direction for future work is achieving hard real-time guarantees by replacing the Node.js event loop with a dedicated scheduling thread. The current implementation uses static priority values assigned at message creation, and future work could explore dynamic priority adjustment based on message age or system state. A priority inheritance protocol could be implemented to prevent high-priority

messages from being blocked by lower-priority ones holding shared resources.

Extending the scheduler to coordinate message priorities across multiple Node-RED instances would enable its use in larger distributed IoT architectures. Future evaluations should include real IoT sensor data rather than synthetic workloads - including realistic skewed priority distributions and bursty arrival patterns rather than the uniform distribution used here - to better assess scheduler performance under realistic conditions. A further direction is a full deadline-miss-rate benchmark comparing EDF and the laxity-based variant under controlled overload conditions, where the laxity policy is expected to differ most from EDF. A user-friendly configuration interface within the Node-RED editor would make the scheduling system more accessible to non-technical users.

7.4 Ethical Considerations, Societal Benefits, and Sustainability

The project does not involve human participants, personal data, or sensitive materials, and therefore does not raise ethical concerns. Since the evaluation relies on synthetic workloads rather than real IoT deployments, no data collection or privacy considerations apply. The implementation does not alter message content or enable any form of data inspection beyond what Node-RED already exposes to the developer. The source code is openly available, which promotes research transparency and allows future work to build on the implementation. Taken together, the nature of the data, the scope of the implementation, and the open-source publication model ensure the work meets ethical research standards.

Priority-based scheduling has direct relevance for safety-critical IoT deployments where delayed messages can lead to equipment damage or safety hazards. In industrial automation, a delayed emergency stop command caused by a backlog of routine sensor readings could result in machinery damage or worker injury. Connected healthcare devices generate a mix of routine measurements and urgent alerts, and the ability to distinguish between them at the scheduling level has direct patient safety implications. In smart building management, the timely delivery of safety alerts over routine updates can directly affect evacuation response times and occupant safety. The scheduling system presented in this thesis directly addresses these scenarios, extending Node-RED's applicability to environments where message timing has tangible safety consequences.

The solution runs on standard consumer hardware, as shown by the evaluation on a low-spec virtual machine, making it suitable for resource-constrained edge deployments. CPU usage remained below 6%, and peak memory consumption stayed under 148MB throughout the full test run, confirming that the scheduler introduces minimal resource overhead. Embedding the scheduler directly into Flow.js eliminates the need for additional system components, reducing both deployment complexity and the energy footprint of the overall IoT stack. More efficient message handling also means that critical events are resolved faster, potentially reducing the duration of fault conditions and the energy wasted during abnormal system states. Designing for resource efficiency from the outset ensures that the scheduling system can be adopted in real-world deployments without compromising the sustainability goals of modern IoT architectures.

References

[1] “Serverless architectures for IoT: Reducing latency with event-driven computing,” ResearchGate. https://www.researchgate.net/publication/388029295, accessed: Jan. 21, 2026.

[2] “Node-RED,” Wikipedia. https://en.wikipedia.org/wiki/Node-RED, 2025, accessed: Jan. 21, 2026.

[3] “The node-RED story: How visual programming escaped the lab and conquered industry,” FlowFuse. https://flowfuse.com/blog/2026/01/node-red-history-community-industrial-iot-flowfuse/, accessed: Jan. 21, 2026.

[4] OpenJS Foundation, “Node-RED unleashed: Transforming industrial IoT development,” https://beskar-openjsf.vercel.app/blog/node-red-unleashed, accessed: Jan. 21, 2026.

[5] “Low-code programming for event-driven applications: Node-RED,” https://nodered.org/, accessed: Jan. 21, 2026.

[6] “Chapter 1: Introducing node-RED and flow-based programming,” Packt. https://subscription.packtpub.com/book/web-development/9781800201590, accessed: Jan. 21, 2026.

[7] “What is node-RED? how can I use it to create IoT applications?” RealPars. https://www.realpars.com/blog/node-red, accessed: Jan. 21, 2026.

[8] “Event-driven architecture patterns: Real-world lessons,” DZone. https://dzone.com/articles/event-driven-architecture-real-world-iot, accessed: Jan. 21, 2026.

[9] “Amazon SQS FIFO queues,” Amazon Web Services. https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-fifo-queues.html, accessed: Jan. 21, 2026.

[10] O. Dudycz, “Ordering, grouping and consistency in messaging systems,” https://www.architecture-weekly.com/p/ordering-grouping-and-consistency, accessed: Jan. 21, 2026.

[11] “The node.js event loop,” https://nodejs.org/en/learn/asynchronous-work/event-loop-timers-and-nexttick, accessed: Jan. 30, 2026.

[12] “Node-RED documentation,” https://nodered.org/docs/, accessed: Jan. 28, 2026.

[13] “Javascript file: Node-RED,” https://nodered.org/docs/creating-nodes/node-js, accessed: Jan. 30, 2026.

[14] “How does node-RED deploy work (codewise)?” Google Groups. https://groups.google.com/g/node-red/c/cl3mFPVYhDg, accessed: Jan. 30, 2026.

[15] “Writing functions: Node-RED,” https://nodered.org/docs/user-guide/writing-functions, accessed: Jan. 30, 2026.

[16] “Node JS architecture — single threaded event loop,” DigitalOcean. https://www.digitalocean.com/community/tutorials/node-js-architecture-single-threaded-event-loop, accessed: Jan. 30, 2026.

[17] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022.

[18] “Priority queue using binary heap,” GeeksforGeeks. https://www.geeksforgeeks.org/priority-queue-using-binary-heap/, accessed: Jan. 29, 2026.

[19] “Node.js event loop,” GeeksforGeeks. https://www.geeksforgeeks.org/node-js-event-loop/, accessed: Jan. 30, 2026.

[20] J. W. S. Liu, *Real-Time Systems*. Upper Saddle River, NJ: Prentice Hall, 2000.

[21] “Priority inversion,” Wikipedia. https://en.wikipedia.org/wiki/Priority_inversion, 2025, accessed: Jan. 25, 2026.

[22] “Priority scheduling in operating system,” GeeksforGeeks. https://www.geeksforgeeks.org/priority-scheduling-in-operating-system/, accessed: Jan. 25, 2026.

[23] "Preemptive and non-preemptive scheduling," TutorialsPoint. https://www.tutorialspoint.com/operating_system/os_preemptive_and_non_preemptive_scheduling.htm, accessed: Jan. 25, 2026.

[24] S. A. Khajeh, M. Saberikamarposhti, and A. M. Rahmani, "Real-time scheduling in IoT applications: A systematic review," *Sensors*, vol. 23, no. 1, 2022. doi: 10.3390/s23010232

[25] "Earliest deadline first scheduling," Wikipedia. https://en.wikipedia.org/wiki/Earliest_deadline_first_scheduling, 2026, accessed: Jan. 30, 2026.

[26] "Earliest deadline first (EDF) CPU scheduling algorithm," GeeksforGeeks. https://www.geeksforgeeks.org/earliest-deadline-first-edf-cpu-scheduling-algorithm/, accessed: Jan. 30, 2026.

Appendix A

Test Flow Configurations (JSON)

A.1 Static Priority Queue Correctness Test Flow

Listing A.1: Static Priority Queue Correctness Test Flow

```
[
  {
    "id": "correctness_tab",
    "type": "tab",
    "label": "CORRECTNESS␣TEST"
  },
  {
    "id": "inject_correctness",
    "type": "inject",
    "z": "correctness_tab",
    "name": "START␣CORRECTNESS␣TEST",
    "props": [{"p": "payload"}],
    "repeat": "",
    "once": false,
    "x": 180,
    "y": 80,
    "wires": [["init_correctness"]]
  },
  {
    "id": "init_correctness",
    "type": "function",
    "z": "correctness_tab",
    "name": "Init",
    "func": "const␣iters␣=␣30;\nconst␣buckets␣=␣[10,␣25,␣50,␣75,␣90];\nconst␣msgsPerBucket␣=␣
      ↪ 100;\nflow.set('cr_iters',␣iters);\nflow.set('cr_current',␣0);\nflow.set('cr_buckets',␣
      ↪ buckets);\nflow.set('cr_msgsPerBucket',␣msgsPerBucket);\nflow.set('cr_results',␣
      ↪ {});\nbuckets.forEach(b␣=>␣flow.set('cr_res_'␣+␣b,␣[]));\nnode.warn('CORRECTNESS␣TEST:␣'␣
      ↪ +␣iters␣+␣'␣iterations,␣buckets:␣'␣+␣buckets.join(','));\nreturn␣msg;",
    "outputs": 1,
    "x": 370,
    "y": 80,
    "wires": [["run_correctness"]]
  },
  {
    "id": "run_correctness",
    "type": "function",
    "z": "correctness_tab",
    "name": "Run",
    "func": "const␣iter␣=␣flow.get('cr_current')␣||␣0;\nconst␣iters␣=␣flow.get('cr_iters');\nif␣(iter␣
      ↪ >=␣iters)␣{␣return␣[null,␣msg];␣}\nflow.set('cr_iter',␣iter);\nconst␣buckets␣=␣
      ↪ flow.get('cr_buckets');\nconst␣msgsPerBucket␣=␣flow.get('cr_msgsPerBucket');\nconst␣total␣
      ↪ =␣buckets.length␣*␣msgsPerBucket;\nflow.set('cr_total',␣total);\nflow.set('cr_count',␣
      ↪ 0);\nbuckets.forEach(b␣=>␣flow.set('cr_iter_res_'␣+␣b,␣[]));\nreturn␣[msg,␣null];",
```

```
    "outputs": 2,
    "x": 180,
    "y": 180,
    "wires": [["gen_correctness"], ["stats_correctness"]]
  },
  {
    "id": "gen_correctness",
    "type": "function",
    "z": "correctness_tab",
    "name": "Gen",
    "func": "const␣buckets␣=␣flow.get('cr_buckets');\nconst␣msgsPerBucket␣=␣
        ↪ flow.get('cr_msgsPerBucket');\nconst␣now␣=␣Date.now();\nfor␣(const␣b␣of␣buckets)␣{\n␣␣for␣
        ↪ (let␣i␣=␣0;␣i␣<␣msgsPerBucket;␣i++)␣{\n␣␣␣␣node.send({␣payload:␣'msg-p'␣+␣b␣+␣'-'␣+␣i,␣
        ↪ priority:␣b,\n␣␣␣␣␣␣msgIndex:␣i,␣injectedAt:␣now,␣bucket:␣b␣});\n␣␣}\n}\nreturn␣null;",
    "outputs": 1,
    "x": 370,
    "y": 180,
    "wires": [["collect_correctness"]]
  }
]
```

A.2 EDF Correctness Test Flow

Listing A.2: EDF Correctness Test Flow

```
[
  {
    "id": "correctness_edf_tab",
    "type": "tab",
    "label": "CORRECTNESS␣EDF␣TEST"
  },
  {
    "id": "inject_correctness_edf",
    "type": "inject",
    "z": "correctness_edf_tab",
    "name": "START␣EDF␣CORRECTNESS␣TEST",
    "props": [{"p": "payload"}],
    "repeat": "",
    "once": false,
    "x": 190,
    "y": 80,
    "wires": [["init_correctness_edf"]]
  },
  {
    "id": "init_correctness_edf",
    "type": "function",
    "z": "correctness_edf_tab",
    "name": "Init",
    "func": "const␣iters␣=␣30;\nconst␣buckets␣=␣[50,␣100,␣150,␣200,␣250];\nconst␣msgsPerBucket␣=␣
        ↪ 100;\nflow.set('ce_iters',␣iters);\nflow.set('ce_current',␣0);\nflow.set('ce_buckets',␣
        ↪ buckets);\nflow.set('ce_msgsPerBucket',␣msgsPerBucket);\nbuckets.forEach(b␣=>␣
        ↪ flow.set('ce_res_'␣+␣b,␣[]));\nnode.warn('EDF␣CORRECTNESS:␣'␣+␣iters␣+␣'␣iters,␣deadline␣
        ↪ buckets:␣'␣+␣buckets.join(',')␣+␣'ms');\nreturn␣msg;",
    "outputs": 1,
    "x": 390,
    "y": 80,
    "wires": [["run_correctness_edf"]]
  }
]
```

Full flow JSON files are available in the project repository at `https://github.com/DrelaDominika/node-red-priority-scheduler`.

Appendix B

Pseudocode and Code Extracts

The core algorithms implemented in this thesis are presented in full in Section 4.4, including pseudocode for heap insert (Algorithm 1), heap extract-min (Algorithm 2), comparison functions for static priority and EDF scheduling (Algorithm 3), and the laxity calculation extension (Algorithm 4).

The following extract shows the key modification made to Node-RED's Flow.js to intercept messages and route them through the priority scheduler:

Listing B.1: Flow.js message interception and routing

```
// Flow.js -- message interception and routing
function handlePreRoute(flow, sendEvent, reportError) {
    const msg = sendEvent.msg;

    // Check if message carries priority or deadline metadata
    if (msg.priority !== undefined || msg.deadline !== undefined) {
        // Route through priority scheduler
        flow.scheduler.enqueue(sendEvent);
    } else {
        // Default FIFO routing
        sendEvent.destination.node =
            flow.getNode(sendEvent.destination.id);
        handlePreDeliver(flow, sendEvent, reportError);
    }
}
```

Full source code for the modified Flow.js, HybridPriorityQueue, and MessageScheduler classes is available in the project repository at `https://github.com/DrelaDominika/node-red-priority-scheduler`.

Appendix C

Extended Result Tables

Tables C.1 and C.2 present the full performance results across all schedulers, message counts, and simulated processing times. All values represent means over 100 iterations. StDev = standard deviation.

Table C.1: Full performance results – 0ms and 50ms app delay (means over 100 iterations)

App Delay	Scheduler	Messages	Avg Latency (ms)	StDev	Throughput (msg/s)	StDev
0ms	Vanilla	500	111.0	313.4	8676	5337
0ms	Vanilla	1000	147.8	299.2	11301	5811
0ms	Vanilla	1500	237.2	317.6	9139	5673
0ms	Vanilla	2000	232.4	318.5	8743	4287
0ms	Vanilla	2500	374.1	426.0	6947	3856
0ms	Static PQ	500	53.9	62.3	9020	4056
0ms	Static PQ	1000	130.8	180.3	9026	5584
0ms	Static PQ	1500	131.1	140.6	11399	5506
0ms	Static PQ	2000	252.4	336.3	9698	5759
0ms	Static PQ	2500	283.7	333.6	9768	5747
0ms	EDF	500	59.6	250.3	13480	4561
0ms	EDF	1000	107.6	222.3	12351	4972
0ms	EDF	1500	129.7	168.1	10719	4797
0ms	EDF	2000	299.0	403.6	8622	5511
0ms	EDF	2500	234.0	248.7	9690	4979
50ms	Vanilla	500	39.1	27.5	10507	3999
50ms	Vanilla	1000	69.5	33.5	10337	3621
50ms	Vanilla	1500	136.2	172.9	10281	4722
50ms	Vanilla	2000	144.1	154.9	11365	4197
50ms	Vanilla	2500	217.6	256.6	9731	4554
50ms	Static PQ	500	43.8	49.0	10372	5015
50ms	Static PQ	1000	114.3	207.6	9947	5270
50ms	Static PQ	1500	122.5	102.7	10060	5054
50ms	Static PQ	2000	219.4	390.5	10069	5330
50ms	Static PQ	2500	237.2	246.4	9312	4951
50ms	EDF	500	38.3	25.8	10268	4731
50ms	EDF	1000	58.4	127.9	14172	3061
50ms	EDF	1500	113.4	87.2	9814	4308
50ms	EDF	2000	160.2	173.5	10476	4523
50ms	EDF	2500	201.3	220.3	10364	4621

Table C.2: Full performance results – 100ms and 200ms app delay (means over 100 iterations)

App Delay	Scheduler	Messages	Avg Latency (ms)	StDev	Throughput (msg/s)	StDev
100ms	Vanilla	500	56.9	65.3	10137	5261
100ms	Vanilla	1000	63.3	51.8	12525	4178
100ms	Vanilla	1500	147.7	225.8	10938	5019
100ms	Vanilla	2000	163.2	153.6	10290	4444
100ms	Vanilla	2500	215.3	189.4	9267	4447
100ms	Static PQ	500	51.1	39.5	7956	4197
100ms	Static PQ	1000	114.7	268.2	10595	5374
100ms	Static PQ	1500	157.0	208.9	9322	5127
100ms	Static PQ	2000	109.5	58.2	11745	3876
100ms	Static PQ	2500	163.2	146.0	11226	4354
100ms	EDF	500	43.3	56.5	11909	5054
100ms	EDF	1000	90.2	93.5	10394	5038
100ms	EDF	1500	102.1	57.0	9771	3647
100ms	EDF	2000	189.2	175.1	8608	4062
100ms	EDF	2500	207.8	182.9	8806	3741
200ms	Vanilla	500	45.3	17.6	7652	2507
200ms	Vanilla	1000	77.0	19.4	8152	2098
200ms	Vanilla	1500	113.4	26.8	8004	1870
200ms	Vanilla	2000	186.0	101.4	7287	2924
200ms	Vanilla	2500	188.1	117.1	9294	3853
200ms	Static PQ	500	49.7	37.8	8237	3698
200ms	Static PQ	1000	82.9	59.5	8597	3299
200ms	Static PQ	1500	112.6	98.8	10225	3884
200ms	Static PQ	2000	165.1	120.2	9170	4076
200ms	Static PQ	2500	187.0	103.6	9021	3575
200ms	EDF	500	54.8	64.0	9006	4225
200ms	EDF	1000	85.4	72.4	9493	4129
200ms	EDF	1500	151.5	207.9	8706	3935
200ms	EDF	2000	175.7	176.5	8903	3697
200ms	EDF	2500	210.9	211.9	8897	3764

TRITA-EECS-EX-2026:404
Stockholm, Sweden 2026

www.kth.se