Energy Efficiency of REST, GraphQL, and gRPC API Architectures: A Comparative Empirical Study

University of Turku
Department of Computing
Master of Science (Tech) Thesis
Software Engineering
June 2026
Humza Manzoor

UNIVERSITY OF TURKU
Department of Computing

Humza Manzoor: Energy Efficiency of REST, GraphQL, and gRPC API Architectures: A Comparative Empirical Study

Master of Science (Tech) Thesis, 121 p., 6 app. p.
Software Engineering
June 2026

Growing digital infrastructure energy demands have made energy-aware software design a key concern in sustainable software engineering. Application Programming Interfaces (APIs) serve as the communication backbone of modern distributed systems, yet their energy implications remain insufficiently studied. This thesis presents a controlled empirical study comparing the energy consumption of six .NET 8 API implementations across three architectural paradigms - REST, GraphQL, and gRPC - using hardware-based power measurement through the PowerGoblin framework.

The study is structured around four research questions. The baseline comparison evaluates all six implementations under identical conditions, finding that architectural energy differences are small at small payload sizes, with backend CPU energy ranging from 5.995 J to 7.338 J per 100-request run across all implementations. Among the three implementations selected for further study, GraphQL.GraphQLNet is the most energy-efficient across all three measured metrics (frontend, backend, and network energy) at this baseline payload, while Rest.Controllers is the least efficient. The payload scaling experiment investigates how energy consumption changes across six payload levels from 10 to 320 posts and comments, finding that this baseline ordering inverts completely by a payload of 40 posts and comments: Grpc.AspNetCore becomes the most energy-efficient implementation across all three metrics, while GraphQL.GraphQLNet becomes the least efficient due to its resolver-based execution model. At the maximum payload level, GraphQL.GraphQLNet's backend CPU energy is approximately 84% higher than Grpc.AspNetCore's and approximately 53% higher than Rest.Controllers'. The concurrency experiment examines energy consumption across eleven concurrency levels from 5 to 4000 virtual users, finding that all three architectures perform comparably up to 30 concurrent users but that gRPC shows a clear energy advantage at extreme concurrency levels. An additional empirical experiment measures energy consumption across three multi-hop network topologies, finding that an additional wired switch hop adds approximately 0.75 J while an additional wireless bridge hop adds approximately 8 J, roughly ten times more, refining the additive per-hop energy model proposed in prior network-level energy research.

These findings show that API architectural energy differences are workload-dependent, becoming significant only under high payload or concurrency, and offer data-driven guidance for energy-aware API design.

Keywords: energy efficiency, REST, GraphQL, gRPC, API architecture, sustainable software engineering, green ICT, empirical study

Contents

List of Figures

List of Tables

1 Introduction

1.1 Background and Motivation

The exponential growth of digital applications and cloud-based systems has driven an unprecedented demand for networked communication, resulting in a significant increase in energy consumption within the Information and Communication Technology (ICT) sector. Recent projections suggest that in a worst-case scenario, ICT could account for up to 23% of global greenhouse gas emissions by 2030 if energy efficiency improvements are not prioritized [1]. As software systems become increasingly complex and data-intensive, their energy demand is no longer confined to hardware infrastructure alone; the software architecture and communication design play an equally critical role in determining the overall environmental impact of digital systems.

Among the foundational components of modern software systems are Application Programming Interfaces (APIs), which serve as the communication backbone between distributed services, client applications, and cloud infrastructures. The efficiency of APIs directly influences both system performance and energy consumption. The landscape of API and inter-service communication technologies is broader than any single study can cover, encompassing not only REST, GraphQL, and gRPC but also older and more specialised paradigms such as SOAP (Simple Object Access Protocol), a heavyweight, XML-based messaging standard once

dominant in enterprise integration; WebSocket, a persistent, full-duplex protocol suited to real-time, bidirectional communication; and message-queue-based middleware such as MQTT and AMQP, designed for asynchronous, loosely coupled, often resource-constrained environments such as IoT. SOAP has been largely superseded by lighter-weight alternatives for general-purpose web APIs due to its verbosity and processing overhead, while WebSocket and message-queue protocols address fundamentally different communication patterns, persistent bidirectional streaming and asynchronous publish-subscribe messaging respectively, rather than the synchronous request-response model that characterises the bulk of everyday API interactions.

REST, GraphQL, and gRPC were selected for this study because, unlike SOAP, all three remain in active, widespread use, and unlike WebSocket and message-queue middleware, all three are designed primarily around the request-response interaction pattern, allowing them to be compared under a common, controlled experimental workload (Chapter 3) without the comparison being confounded by fundamentally different communication semantics. With the rapid evolution of web technologies, REST, GraphQL, and gRPC have emerged as the most widely adopted API architectures within this request-response paradigm, each offering distinct design philosophies, communication protocols, and serialization mechanisms [2], [3]. While REST remains dominant due to its simplicity and compatibility, GraphQL offers flexible query structures that improve data retrieval efficiency, and gRPC leverages binary serialization and HTTP/2 for high-performance communication. Despite these architectural advancements, the energy implications of these APIs have not been extensively studied, with most prior research focusing on performance metrics such as latency, throughput, and CPU utilization rather than power efficiency [4], [5].

At the same time, the concept of Green and Sustainable Software Engineering has gained increasing attention in both academia and industry. Frameworks such as the GREENSOFT model emphasize that sustainability must be embedded into

software design and decision-making processes, as architectural and implementation-level choices significantly affect energy usage throughout the system lifecycle [6]. This shift toward sustainability demands that software engineers consider energy efficiency as a first-class design criterion, alongside performance, scalability, and maintainability. Within this context, the comparative analysis of API architectures represents a critical opportunity to understand how communication design choices contribute to the software energy footprint.

The motivation for this research arises from the need to bridge the gap between traditional performance optimization and sustainability-driven software engineering. By empirically evaluating and comparing the energy consumption behavior of REST, GraphQL, and gRPC architectures, this study aims to generate data-driven insights into how software communication mechanisms influence power usage under real-world workloads. The findings are expected to inform both researchers and practitioners on how to design APIs that not only perform efficiently but also contribute to the broader goal of environmentally responsible computing.

1.2 Problem Statement

The rapid expansion of digital services and distributed applications has significantly increased the energy demands of software systems. Information and Communication Technology (ICT) faces significant and growing greenhouse gas emissions, with worst-case projections suggesting ICT could contribute up to 23% of global emissions by 2030, driven by the continued growth of cloud computing and service-oriented architectures [1]. While hardware efficiency and data center optimization have been widely studied, the energy implications of software architectural decisions, particularly in API-driven systems, remain underexplored.

APIs serve as the communication backbone of modern distributed applications, with REST, GraphQL, and gRPC emerging as dominant architectural paradigms.

Existing comparative studies largely focus on performance metrics such as latency and throughput [2], [3], [4], yet performance efficiency does not necessarily imply energy efficiency. Architectural characteristics such as serialization format, transport protocol behavior, and request-processing mechanisms may significantly influence energy consumption. Although sustainable software engineering research emphasizes energy-aware design [5], [6], there is limited empirical evidence comparing the energy efficiency of these API architectures in a systematic manner.

Furthermore, prior research rarely examines how implementation-level variations within the same architecture influence energy behavior. Different libraries and configurations may produce measurable differences in energy consumption, even when adhering to identical architectural principles. Additionally, controlled laboratory experiments do not directly address how energy usage scales in distributed environments with multiple intermediate network nodes.

To address these gaps, this research conducts a controlled experimental comparison of REST, GraphQL, and gRPC architectures and their respective implementations. The availability of the hardware-based measurement laboratory at the University of Turku, together with the PowerGoblin measurement framework developed for that laboratory as part of the recently concluded Visiiri project, provided the practical opportunity to carry out these empirical experiments. Using the PowerGoblin measurement framework [7] for hardware-based energy measurement (described in detail in Chapter 3), the study analyzes architectural and operational factors affecting energy consumption, investigates how payload size and concurrent client load influence energy consumption across the tested architectures, and conducts an empirical multi-hop network topology experiment to derive an analytical framework for estimating total energy usage in distributed environments with varying numbers and types of intermediate network nodes. This integrated experimental and analytical approach contributes to advancing energy-aware API design within

sustainable software engineering.

1.3 Research Objectives and Questions

The objective of this research is to systematically analyze and compare the energy efficiency of three widely used API architectures - REST, GraphQL, and gRPC - and their respective implementations in distributed software systems. As APIs serve as the core communication mechanism in modern applications, their architectural design choices directly influence the overall energy footprint of software systems. This study establishes a controlled experimental framework using hardware-based measurement to evaluate energy consumption across the frontend, backend, and network components of a full-stack API interaction. Beyond cross-architectural comparison, the research investigates which architectural and operational factors contribute most to the observed energy differences, examines how payload size and concurrent client load affect the energy consumption of each architectural paradigm, and extends the experimental findings through an empirical multi-hop network topology experiment and a resulting analytical model that estimates total energy consumption in distributed environments with varying numbers and types of intermediate network nodes.

In alignment with this objective, the study is guided by the following research questions:

- **RQ1:** How do REST, GraphQL, and gRPC architectures differ in baseline energy consumption?
- **RQ2:** How does payload size affect energy consumption across REST, GraphQL, and gRPC API architectures?
- **RQ3:** How does concurrent client load affect energy consumption across REST, GraphQL, and gRPC API architectures?

- **RQ4:** How can measured energy consumption be extended to estimate energy usage in distributed environments?

1.4 Scope

The scope of this research is defined by two complementary components: a theoretical component grounded in a review of existing literature, and an empirical component consisting of controlled hardware-based measurements and analytical modelling. Together, these two components define what this thesis covers and, equally importantly, what it does not.

The theoretical component is concerned with establishing the conceptual and technical foundations required to reason about energy efficiency in API communication. It covers the principles of sustainable software engineering and Green ICT, the architectural characteristics of REST, GraphQL, and gRPC, and the state of the art in comparative studies of API performance and energy behaviour. The review is bounded to publications directly relevant to these three topics and to the communication layer of API-driven systems. It does not attempt to survey the broader landscape of green software engineering practices unrelated to API design, nor does it engage with programming-language-level or hardware-level energy research except where directly relevant to the research questions.

The empirical component is concerned with quantifying and comparing the energy consumption behaviour of the three API architectures under controlled conditions. It encompasses a comparative analysis of REST, GraphQL, and gRPC, with multiple implementations developed for each architecture to capture implementation-level variations. All measurements are obtained using the PowerGoblin measurement framework [7], which provides hardware-based energy instrumentation and is described in detail in Chapter 3. The measurements are taken across the frontend, backend, and network components of a full-stack API interac-

tion to provide a comprehensive picture of where energy is consumed during API communication. Each implementation is designed to provide functionally equivalent behaviour, ensuring that any observed differences in energy usage stem from architectural and protocol variations rather than differences in business logic or computational complexity. The empirical component further includes the analysis of architectural and operational factors such as serialisation formats, request handling mechanisms, and transport protocols, as well as the investigation of how payload size and concurrent client load influence energy consumption across the tested architectures.

The empirical scope is intentionally limited to the API communication layer of a full-stack interaction, encompassing the frontend client, the backend server, and the intermediate network device(s) that connect them. For RQ1–RQ3, this is a single network switch; for RQ4, the scope is extended to multiple intermediate network devices, additional wired switches and a wireless bridge, arranged in three physical topologies, in order to empirically examine how additional network infrastructure affects total energy consumption. The empirical scope does not extend to other components such as databases, message queues, or persistent storage systems. The results are therefore focused on providing insights that are specifically relevant to API design and implementation, contributing to the broader objective of developing energy-efficient and sustainable software systems.

In addition to the experimental measurements, the empirical component incorporates the RQ4 multi-hop topology experiment and a resulting analytical estimation model derived from its measured per-hop energy costs, used to approximate total energy consumption in larger distributed environments with varying numbers and types of intermediate network nodes. This modelling element is grounded directly in physical measurements of multiple network topologies rather than in extrapolation alone.

1.5 Methodological Overview and Practical Usage

This section provides a high-level overview of the research approach adopted in this thesis and briefly contextualises the practical relevance of the three API architectures under study. The intention is to clarify, before the detailed chapters that follow, which research method is applied to which part of the investigation and where REST, GraphQL, and gRPC are typically encountered in real-world software systems.

The research presented in this thesis combines two distinct but complementary methodological approaches. The theoretical and background aspects of the study - including the discussion of sustainable software engineering, the technical characteristics of each API architecture, and the synthesis of prior work on performance and energy efficiency - are addressed through a focused literature review. This review draws on established academic databases and is conducted following a documented search and selection process, which is described in detail in Chapter 2. The purpose of this part of the research is to establish the conceptual foundation on which the empirical work rests and to identify the specific gaps that the empirical study is designed to address.

The empirical aspects of the study, which form the core contribution of the thesis, are addressed through controlled hardware-based energy measurement experiments. These experiments are conducted in a dedicated laboratory environment using the PowerGoblin measurement framework [7], which enables direct instrumentation of the electrical power consumed by the devices under test. The experimental setup, implementation details, measurement procedure, and analysis approach are presented in Chapter 3. In addition to the RQ1–RQ3 experiments, the thesis conducts an empirical multi-hop network topology experiment (RQ4), measuring energy consumption across three physical network configurations with increasing numbers and types of intermediate nodes. The per-hop energy costs derived from this experiment are used to refine an analytical model that extends the measured results

to estimate energy consumption in larger distributed environments, informed by established network-level energy estimation work.

Beyond their academic significance, the three API architectures examined in this thesis are all widely deployed in modern software systems, and understanding their practical contexts helps clarify why their comparative energy behaviour matters. REST remains the most broadly adopted API style and forms the backbone of the majority of public-facing web services and open data APIs. Its simplicity, compatibility with standard HTTP tooling, and human-readable payloads have made it the default choice for open web platforms, content management systems, and integration interfaces across virtually every industry that exposes data over the internet.

GraphQL has gained significant traction in applications where client-side data requirements are highly variable or where the same backend must serve multiple clients with different data needs. It was originally developed at Facebook to address the inefficiencies of REST in mobile applications and has since been adopted by organisations such as GitHub, Shopify, and Netflix for developer-facing APIs and frontend-driven data access. GraphQL is particularly common in modern single-page applications and mobile applications, where minimising the number of network round trips and the volume of over-fetched data is important for user experience and bandwidth consumption.

gRPC occupies a different niche, primarily in backend and internal microservice communication rather than public-facing APIs. Its binary serialisation, contract-first design, and HTTP/2 transport make it well suited to high-throughput, low-latency communication between services within a controlled environment. It is widely used in large-scale backend systems at companies such as Google, Netflix, and Square, and has become a common choice for service mesh architectures, data pipelines, and inter-service communication in cloud-native deployments. Because of its browser compatibility constraints, gRPC is rarely exposed directly to end-user

clients but instead operates within the server-side ecosystem of modern distributed applications.

Together, these three architectures cover the dominant communication patterns in contemporary software systems - from public web APIs, to flexible client-driven data access, to high-performance internal service communication. Any meaningful analysis of API energy efficiency must therefore account for all three, which is the approach taken throughout this thesis.

1.6 Structure of the Thesis

The structure of this thesis follows a logical progression from establishing the theoretical foundation to presenting the experimental results and their implications. Chapter 2 explores the concepts of sustainable software engineering and Green ICT, providing the background necessary to understand energy efficiency in software systems. The chapter also introduces REST, GraphQL, and gRPC architectures and discusses their technical characteristics and previous comparative studies. In doing so, it lays the groundwork for addressing Research Question 1 (RQ1), which focuses on how these API architectures differ in baseline energy consumption.

Chapter 3 details the empirical setup and implementation developed for this study. It introduces the PowerGoblin measurement framework [7], describes the experimental setup and the physical measurement chain, and explains how the API implementations were designed across REST, GraphQL, and gRPC. The chapter also defines the workload scenario, the measurement metrics, the data collection procedure, and the data analysis approach. The threats to validity and broader limitations of the experimental design are discussed in Section 5.5. Rather than directly answering a specific research question, this chapter establishes the empirical foundation required to systematically investigate all four research questions (RQ1 through RQ4) in the subsequent analysis.

Chapter 4 presents the experimental results and analysis. It provides a baseline comparative evaluation of all six implementations addressing RQ1, examines how energy consumption scales with payload size across three representative implementations addressing RQ2, investigates the effect of concurrent client load on energy consumption across the same three implementations addressing RQ3, and presents an empirical multi-hop network topology experiment addressing RQ4. A consolidated statistical evaluation and discussion of the cross-RQ patterns concludes the chapter.

Chapter 5 discusses and interprets the experimental findings in depth, contextualises them within the existing literature on API performance and energy efficiency, examines the practical implications for API design, and acknowledges the limitations of the study.

Finally, Chapter 6 synthesises the conclusions drawn from the study. It summarises the main findings, revisits the research questions, highlights the contributions of the work, and outlines directions for future research. This chapter also presents a refined analytical energy estimation framework for distributed environments, grounded in the empirical per-hop energy costs derived from the RQ4 multi-hop topology experiment in Chapter 4, and identifies generalising this framework to other implementations, payloads, and network configurations as a direction for future work. The thesis closes by connecting the empirical findings to concrete real-world application domains, clarifying where the workload-dependent results of this study are most directly applicable.

1.7 Declaration of Generative AI

Generative AI tools were used during the preparation of this thesis. The academic search engine Consensus (consensus.app) was used on a limited scale during the initial literature search phase to generate a preliminary set of candidate publica-

tions, as described in Section 2.1. The AI assistant Claude (Anthropic) was used to support the empirical setup and measurement infrastructure configuration, code review and debugging of the .NET 8 API implementations and load client, refining of thesis text across all chapters, and assistance with Latex formatting. All empirical measurements, data collection, and analysis decisions were performed by the author. All AI-generated text was reviewed, verified, and edited by the author to ensure accuracy and consistency with the experimental work. The research questions, experimental design, and conclusions are the author's own.

2 Background and Related Work

2.1 Research Methods and Sources

The research presented in this thesis comprises two complementary components: a literature review covering sustainable software engineering, energy efficiency in software systems, and comparative evaluations of API architectures; and an empirical study measuring the energy consumption of REST, GraphQL, and gRPC using hardware-based instrumentation through the PowerGoblin framework, presented in Chapters 3 through 5. The literature component is best described as a structured but non-systematic narrative review. Sources were gathered through a combination of an AI-assisted tool and manual searching, screened against a consistent set of relevance criteria, and grouped into thematic categories, but without a pre-registered protocol, exhaustive database coverage, or independent second screening. Its aim is to establish the conceptual foundation for the empirical study and to identify the research gap it addresses, rather than to map the field exhaustively.

The search began with the AI-powered academic search engine Consensus (consensus.app), which returned approximately ten candidate publications. On manual verification, however, roughly six proved unreliable, containing incorrect bibliographic details or referring to publications that could not be located through any indexed source, and were discarded. Owing to these reliability concerns, Consensus was not used further, and the remaining searching was carried out manually through

Google Scholar. The publications themselves were accessed primarily through ResearchGate and IEEE Xplore, and additionally through the ACM Digital Library, Wiley Online Library, MDPI, and ScienceDirect, as well as the university thesis repositories Theseus, DiVA, and UTUPub. Foundational and definitional sources, notably the REST, GraphQL, and gRPC specifications and documentation, were obtained directly from their official sources rather than through search.

A consistent set of criteria guided inclusion and exclusion. A publication was included if its title, abstract, and keywords indicated a clear connection to at least one of three areas: comparative analysis of API architectures; direct measurement of energy or power consumption in software systems; or the conceptual and methodological foundations of sustainable software engineering. A temporal boundary of 2010 was applied, with an exception for foundational architectural references that define the technologies under study, specifically Fielding's dissertation on REST (2000), the GraphQL specification, and the gRPC documentation. Publications were excluded when closer reading placed their focus outside this scope, for example mobile-application energy profiling, container-monitoring overhead, or platform-specific energy behaviour unrelated to API communication. The retained literature was organised into three categories used throughout this thesis: Performance-Focused (benchmarking without direct energy measurement), Energy Efficiency (direct hardware or profiling-based energy measurement), and Sustainability and Modelling (conceptual frameworks, surveys, and analytical or network-level energy models).

Applying these criteria yielded 33 included and 10 excluded publications, listed by category in Appendix A. It should be emphasised that this process, while structured, was not fully systematic: it was carried out by a single author without a pre-registered protocol or an independent second screener, and the choice of databases and search terms was pragmatic rather than exhaustive. The review should therefore be read as a deliberately scoped narrative synthesis, sufficient to situate the empiri-

cal study and motivate its research questions, rather than a reproducible systematic mapping of the literature on API energy efficiency.

2.2 Sustainable Software Engineering and Green ICT

The continuous growth of digital infrastructure has led to increasing concern about the environmental impact of Information and Communication Technology (ICT). Data centres, cloud platforms, and networked software systems collectively account for a significant and growing share of global electricity consumption. Andrae and Edler [1] projected that, in a worst-case scenario, Communication Technology could account for up to 51% of global electricity demand and up to 23% of global greenhouse gas emissions by 2030, underscoring the urgency of energy-aware design at every layer of the technology stack. As a result, the concept of Green ICT has emerged as a multidisciplinary effort to reduce the ecological footprint of computing systems through both hardware and software innovation. Green ICT encompasses two complementary perspectives: *Green in ICT* and *ICT for Green*, the first focusing on minimising the environmental impact of ICT itself, and the second emphasising the use of ICT as a tool to enable sustainability in other domains such as smart grid management or energy-efficient logistics [8]. Within the first perspective, software plays a crucial yet frequently underestimated role. Inefficient code, excessive computation, and redundant communication processes can substantially increase a system's overall energy footprint, even when running on modern, energy-efficient hardware.

The field of Sustainable Software Engineering (SSE) extends these principles by embedding environmental sustainability directly into software design, development, and maintenance processes. Naumann et al. [5] introduced the GREENSOFT

model, one of the foundational frameworks in this area, which defines sustainability as a lifecycle attribute encompassing not only technical efficiency but also long-term maintainability, usability, and resource consumption. Critically, the model argues that sustainability cannot be retrofitted after deployment, it must be considered from the earliest stages of architectural design. Jagroep [6] builds on this foundation by advancing the concept of *Green Software Products*, encouraging practitioners to evaluate software quality not only in terms of performance and reliability but also energy and carbon efficiency. Importantly, Jagroep's work challenges the implicit assumption that performance optimisation is a proxy for energy efficiency, a conflation that is common in practice but empirically unsupported. O'Dea et al. [8] reinforce this further by proposing a measurement-driven approach to software sustainability in microservice architectures, where empirical energy data guides architectural decision-making rather than relying on performance metrics as indirect indicators. Complementing this, Bajrami [9] conducted a broader survey of industry practices and research trends in software sustainability, finding that while organisational awareness of energy efficiency is growing, systematic energy measurement remains inconsistently applied in practice, revealing a persistent gap between sustainability ambitions and engineering reality.

From a broader ICT perspective, Hilty and Aebischer [10] contextualise these software-level concerns within the larger landscape of ICT infrastructure, emphasising that communication layers and distributed architectures contribute significantly to overall energy demand. Their work highlights that sustainability in ICT cannot be addressed at the hardware level alone, software architectural decisions, including communication protocols and data exchange mechanisms, play an equally important role in determining the overall energy footprint of digital systems. Contrary to earlier expectations that hardware efficiency gains would indefinitely offset rising demand, Masanet et al. [11] demonstrated that global data centre energy use

remained relatively stable between 2010 and 2018 despite a more than tenfold increase in global IP traffic and a sixfold increase in compute instances, an outcome driven largely by improvements in server utilisation and cooling efficiency rather than software-level optimisation. This finding carries an important implication: as hardware efficiency gains plateau, software-level contributions to energy consumption will become increasingly significant and cannot be ignored in sustainable system design.

Recognising this, Lago et al. [12] argue that energy efficiency must be treated as a *first-class quality attribute* in software engineering, comparable to performance, security, or maintainability. This repositioning has direct implications for how API architectures are designed and evaluated. It is worth noting that API architecture, as discussed throughout this thesis, refers to a local aspect of a software system, the design of its communication interfaces, rather than the global architecture of the software or system as a whole; nonetheless, this local design choice carries measurable energy consequences, as the remainder of this section discusses. Serialisation format selection, transport protocol choices, and communication patterns are not merely performance concerns, they are energy decisions with measurable environmental consequences. Traditional software optimisation has concentrated on performance and scalability, treating energy consumption as an emergent byproduct rather than a deliberate design target. The growing body of research in Green ICT and SSE collectively demands a fundamental shift in both measurement culture and design methodology [5], [6], [9].

In the context of this study, these principles form the theoretical foundation for evaluating API architectures through a sustainability lens. By understanding sustainable software engineering as a holistic practice, one that accounts for the interplay between code, architecture, and resource usage, this research aligns itself with the broader objectives of Green ICT. The following sections build upon this

foundation by examining how web and API systems contribute to overall software energy consumption, and how architectural choices in REST, GraphQL, and gRPC can influence the sustainability of modern distributed systems.

2.3 Energy Efficiency in Web and API Systems

As web-based and distributed systems have become the backbone of modern computing, their energy consumption has emerged as a key sustainability concern. Unlike traditional desktop software, web applications operate in a client–server environment where energy usage is distributed across multiple layers, client devices, network infrastructure, and backend servers. Each layer contributes to the total energy footprint, and research indicates that data centres and the networked systems they support account for a significant and growing share of global electricity consumption [11]. In particular, APIs serving as the communication interface between distributed components play a critical role in determining both the performance and energy characteristics of web-based ecosystems. Yet despite their central role, the energy behaviour of API systems remains comparatively underexplored relative to their performance characteristics.

Energy efficiency in web systems can be understood as the ratio between useful computational output and total energy consumed, encompassing factors such as data transfer efficiency, response latency, and resource utilisation. Lago et al. [12] establish that framing energy efficiency as a software quality attribute is essential, without treating it as a measurable design target, it remains an afterthought rather than an engineering concern. This framing is particularly relevant in API systems, where architectural decisions such as serialisation format, transport protocol, and request handling mechanisms directly shape energy consumption patterns. Hindle [13] reinforces this by highlighting the methodological challenges of green software engineering, cautioning that performance-oriented optimisation does not reliably

translate into energy savings, a distinction that is frequently overlooked in both research and practice. This is a critical observation: assuming that a faster API is necessarily a more energy-efficient one is a methodological shortcut that lacks empirical grounding.

Empirical investigations into software energy consumption at the developer and architectural level provide further insight. Pinto et al. [14] analysed questions posted by developers on Stack Overflow related to software energy consumption, finding that architectural and implementation-level concerns, including data serialisation, caching strategies, and communication patterns, frequently surface as energy-related issues in practice. This is significant because it demonstrates that energy awareness is not purely an academic concern; it manifests in real engineering decisions made by practitioners daily. However, the study also reveals that developer understanding of energy implications remains fragmented and inconsistent, with many practitioners relying on intuition rather than measurement. This gap between awareness and systematic practice underscores the need for empirical, measurement-driven research into API energy behaviour.

Building on this, Capra et al. [15] conducted empirical measurements of application-level energy consumption and demonstrated that software design patterns, including caching, batching, and asynchronous processing, produce measurable and reproducible differences in power usage. Critically, their findings show that improvements in execution time do not always correspond to proportional reductions in energy consumption. This decoupling of performance and energy efficiency is an important result: it means that optimising an API for speed does not guarantee a reduction in its energy footprint, and that direct hardware-based energy measurement is necessary to draw reliable conclusions about sustainability.

At the data exchange layer, the choice of serialisation format represents a particularly important architectural decision with potential energy implications. The

use of compact binary formats such as Protocol Buffers, as documented by Google [16], reduces both payload size and the computational overhead associated with encoding and decoding operations compared to text-based formats such as JSON. Smaller payloads reduce transmission time and network load, while lower encoding complexity reduces CPU cycles, both of which have direct energy consequences. However, while these efficiency characteristics are well established in terms of bandwidth and latency, direct hardware-based energy measurement comparing binary and text serialisation under realistic API workloads remains limited in the literature. This represents a meaningful gap: the energy implications of serialisation format selection are theoretically grounded but empirically undervalidated.

From the perspective of sustainable software engineering, web and API systems represent a critical intersection of architectural decision-making and environmental impact. They are among the most continuously active components in cloud ecosystems, handling high volumes of requests around the clock. Yet the existing body of research treats energy efficiency in these systems inconsistently, some studies measure energy directly [15], others infer it from performance proxies [13], and many do not measure it at all. This methodological inconsistency makes cross-study comparison difficult and highlights the need for controlled, hardware-based energy measurement specifically targeting API communication patterns. Understanding the energy behaviour of API architectures is therefore not only a technical challenge but a necessary step toward designing software systems that are genuinely sustainable.

2.4 REST, GraphQL, and gRPC: Architectural Overview

The selection of an API architectural style is not merely a technical preference, it is a design decision with measurable consequences for performance, maintainability, and increasingly, energy consumption. Three architectures have come to dominate modern distributed systems: REST, GraphQL, and gRPC. Each embodies a distinct communication philosophy, serialisation mechanism, and transport strategy, and each carries different implications for how efficiently resources are consumed during API interactions. Understanding these architectural differences at a technical level is a prerequisite for any meaningful comparative energy analysis.

2.4.1 REST Architecture

Representational State Transfer (REST), formalised by Roy Fielding in his doctoral dissertation in 2000 [17], is an architectural style that defines a set of constraints for building scalable and loosely coupled distributed systems. REST APIs operate primarily over HTTP/1.1, using Uniform Resource Identifiers (URIs) to represent resources and standard HTTP methods, GET, POST, PUT, DELETE, and PATCH, to manipulate them. A RESTful system is governed by six architectural constraints: client–server separation, statelessness, cacheability, uniform interface, layered system, and optional code-on-demand. These constraints collectively promote scalability and simplicity, making REST highly accessible and straightforward to implement across diverse technology stacks.

However, REST's architectural simplicity comes with inherent limitations that carry energy implications. The statelessness constraint requires each request to carry all necessary context independently, which increases repetitive data exchange across requests. REST typically employs text-based serialisation formats such as JSON or

XML, which prioritise human readability and interoperability at the cost of larger payload sizes and increased parsing overhead. As Nadareishvili et al. [18] discuss, the combination of verbose text-based serialisation and HTTP header overhead introduces additional bandwidth consumption compared to binary communication mechanisms, a trade-off that becomes increasingly significant at scale. Furthermore, REST's resource-oriented design means that retrieving complex, interrelated data structures often requires multiple sequential requests, each incurring independent network and processing overhead. While REST remains dominant due to its simplicity, broad compatibility, and long-term ecosystem stability, these structural characteristics present challenges from an energy efficiency perspective that warrant careful empirical examination.

2.4.2 GraphQL Architecture

GraphQL was introduced by Facebook in 2015 and subsequently open-sourced and standardised through the GraphQL Foundation [19]. It represents a query-based API paradigm that fundamentally shifts data-fetching control from the server to the client. Unlike REST, which exposes multiple endpoints each mapped to specific resources, GraphQL operates through a single endpoint and uses a strongly typed schema to describe all available data and operations. Clients construct queries specifying precisely which fields they require, and the server responds with exactly that data, no more, no less.

The GraphQL architecture is built around three core components. First, the schema definition describes all data types, fields, and their relationships in a graph-like structure, serving as a contract between client and server. Second, the query language enables clients to express complex, nested data requirements in a single request. Third, resolver functions map each query field to its underlying data source, whether a database, microservice, or external API. This architecture directly

addresses two well-known inefficiencies of REST, over-fetching, where responses contain more data than the client needs, and under-fetching, where multiple requests are required to retrieve related data. Brito and Valente [20] confirmed empirically through a controlled experiment that GraphQL reduces the effort required to implement remote service queries compared to REST, with the advantage becoming more pronounced for complex, multi-parameter data retrieval scenarios.

However, GraphQL's flexibility introduces its own overhead. Each incoming query must be parsed, validated against the schema, and resolved recursively through resolver functions at runtime. This dynamic execution model introduces server-side computational complexity that is absent in REST's static endpoint approach. Resolver execution and query validation processes increase CPU utilisation relative to traditional REST endpoints, and this overhead scales with query complexity. While GraphQL's efficiency gains on the network side are well documented, the server-side computational cost of its execution model is less frequently measured, and its energy implications remain largely unquantified. Serialisation in GraphQL is typically performed using JSON, which retains the text-based inefficiencies shared with REST, limiting potential gains from reduced payload size.

2.4.3 gRPC Architecture

gRPC, developed by Google and initially released as an open-source framework in 2015 before reaching version 1.0 in 2016 [21], is a high-performance Remote Procedure Call (RPC) framework designed for efficient communication between distributed services. Built on HTTP/2 and using Protocol Buffers (Protobuf) as its default serialisation mechanism, gRPC represents a fundamentally different approach to API communication compared to REST and GraphQL, prioritising efficiency, strong typing, and cross-language interoperability over human readability and simplicity.

The gRPC communication model is contract-first. Developers define service interfaces and message structures in a .proto file, from which gRPC automatically generates type-safe client and server code across multiple programming languages. Kumar et al. [22] demonstrated that this contract-first approach, combined with Protobuf binary serialisation, produces measurably lower latency and better CPU efficiency in microservice-to-microservice communication compared to REST, attributing these gains to the compactness of binary encoding and the elimination of runtime type negotiation overhead.

The choice of HTTP/2 as the transport layer provides several structural advantages over HTTP/1.1 that directly benefit API communication efficiency. Multiplexed streams allow multiple requests and responses to be handled concurrently over a single TCP connection, eliminating the head-of-line blocking that affects HTTP/1.1. Header compression through HPACK reduces the bandwidth overhead introduced by repetitive HTTP headers across requests. Persistent connections minimise the latency and energy cost associated with repeated TCP handshakes. Oda and Yamaguchi [23] evaluated these HTTP/2 characteristics empirically and demonstrated that HTTP/2 can achieve throughput comparable to multiple parallel HTTP/1.1 connections while operating over a single connection, a result with direct implications for both performance and resource consumption efficiency.

Architecturally, gRPC supports four interaction patterns: unary RPC, server streaming, client streaming, and bidirectional streaming. This flexibility allows gRPC to serve a wide range of communication scenarios, from simple synchronous request-response interactions to real-time continuous data exchange. However, the gRPC binary protocol and HTTP/2 dependency introduce a significant compatibility constraint. Brandhorst [24] explains that it is currently impossible to implement the gRPC HTTP/2 specification directly in a browser, as no browser API exposes sufficient control over HTTP/2 framing, requiring a dedicated proxy layer such as

gRPC-Web for any browser-based deployment. This limitation restricts gRPC's applicability in public-facing web contexts, confining its primary use to backend microservice communication and internal service meshes where clients can be fully controlled.

Taken together, these three architectures represent distinct points on a spectrum between accessibility and efficiency. REST prioritises simplicity and broad compatibility at the cost of verbose data exchange. GraphQL offers flexible, precise data retrieval but introduces server-side execution complexity. gRPC maximises transport and serialisation efficiency but sacrifices compatibility and human readability. These architectural trade-offs are well understood from a performance perspective, yet their implications for energy consumption remain insufficiently studied, a gap that motivates the comparative empirical analysis conducted in this research.

2.5 Prior Comparative Studies on API Performance and Energy Use

2.5.1 Performance-Focused Comparative Studies: Summary and Limitations

A significant body of comparative research on REST, GraphQL, and gRPC has concentrated on performance evaluation rather than sustainability metrics. These studies typically assess architectural efficiency using indicators such as response time, throughput, scalability under concurrency, and bandwidth utilisation, providing a well-established empirical foundation for understanding the behavioural differences among these three architectures, while simultaneously revealing the boundaries of what performance-oriented methodologies can tell us about energy efficiency.

Śliwa and Pańczyk [2] conducted one of the earlier systematic comparisons

of REST, GraphQL, and gRPC under controlled workloads, finding that REST achieved the best performance in terms of transactions per second and server response time under simple workloads, while gRPC produced the smallest data volume owing to its binary serialisation through Protocol Buffers. This result is notable because it suggests that architectural advantage is not uniform across workload types, REST can outperform gRPC under simple, low-concurrency conditions, a nuance that is frequently overlooked in studies that only test under high load. Chandra and Farisi [3] extended this evaluation through load and stress testing, reporting that gRPC demonstrated superior performance in high-concurrency environments, while REST maintained predictable behaviour under moderate loads. GraphQL, although flexible in handling complex data queries, exhibited additional server-side processing overhead due to query parsing and resolver execution mechanisms, overhead that grows proportionally with query complexity and nested data depth.

Niswar et al. [4] extended comparative evaluation into microservice-based communication scenarios, specifically examining both flat data and nested data retrieval under concurrent request conditions. Their results reinforced the performance advantages of gRPC in terms of throughput and request handling efficiency, particularly in microservice-to-microservice communication, while also revealing that GraphQL places a progressively heavier load on CPU as the number of requests grows compared to REST. Critically, the study emphasised that HTTP/2 features such as multiplexed streams and persistent connections contribute significantly to improved performance characteristics compared to traditional HTTP/1.1-based REST implementations. However, as with earlier studies, the evaluation remained confined to temporal and resource utilisation metrics, without examining whether these protocol-level optimisations translate into reductions in overall energy consumption.

Reichersdörfer [25] contributed a practical benchmarking study using a Trello-like

task management system, demonstrating that under concurrent load gRPC significantly outperformed both REST and GraphQL in latency and throughput, achieving over 44 requests per second compared to 30 for GraphQL and 22 for REST. Reichersdörfer attributed this advantage to HTTP/2 multiplexing and binary framing, which minimise transport overhead under high concurrency. Johansson and Isabella [26] complemented this by evaluating REST and gRPC across three distinct workload scenarios, short, typical, and large payload, finding that gRPC outperformed REST by 77% in response time under short payloads, while the performance gap narrowed considerably under large payload conditions. This workload-dependent variation is a methodologically important finding: it suggests that architectural performance characteristics are not static but shift depending on the nature and size of the data being exchanged, a dimension that most studies do not systematically investigate.

Beyond direct three-way comparisons, research examining underlying communication protocols provides further performance-oriented insight. Oda and Yamaguchi [23] evaluated HTTP/2 performance under varying network conditions and demonstrated that HTTP/2 can achieve throughput comparable to multiple parallel HTTP/1.1 connections while operating over a single connection, a result attributable to header compression and stream multiplexing. While such protocol-level findings clarify the transport efficiency of gRPC's underlying infrastructure, they do not incorporate hardware-level energy measurements or sustainability metrics, leaving the energy implications of these protocol advantages unquantified.

Comparative analyses between REST and GraphQL have similarly focused on efficiency from a data retrieval perspective. Studies consistently highlight that GraphQL reduces over-fetching and under-fetching by allowing clients to request precisely defined data structures [3], [4], yet this flexibility introduces additional computational overhead at the server level. Resolver execution and query validation processes increase CPU utilisation relative to traditional REST endpoints, and this

overhead scales with query complexity. Brito and Valente [20] confirmed empirically through a controlled experiment that GraphQL reduces implementation effort for complex multi-parameter queries compared to REST. The server-side computational cost of GraphQL's execution model, including query parsing, validation, and resolver execution, has been documented in performance-focused studies [3], [4]. Although such overhead is discussed extensively in terms of performance trade-offs, it is not evaluated in terms of energy implications, leaving open the question of whether GraphQL's network efficiency gains are offset by its server-side computational cost when measured in joules rather than milliseconds.

A common methodological pattern across these performance-focused studies is the reliance on benchmark-driven experimentation in controlled environments. Workloads are typically generated using standardised tools to simulate concurrent requests, and evaluation metrics include latency distributions, average response time, requests per second, and CPU utilisation percentages. While CPU and memory usage are sometimes reported, they are rarely translated into direct energy metrics such as joules or watts.[1] Consequently, performance improvements are often implicitly assumed to indicate higher efficiency, without empirical validation through hardware-based power measurement. This assumption is methodologically problematic, as Pereira et al. [27] demonstrated across a large-scale study of programming language energy efficiency that while faster languages tend to consume less energy, this relationship is not universal and does not hold consistently across all contexts, meaning faster execution cannot be assumed to guarantee lower energy use. Applying this insight to API architecture comparison, it becomes clear that the performance rankings consistently reported in the literature cannot be directly

[1]Power and energy are related but distinct quantities: power, measured in watts, is the rate of energy use, while energy, measured in joules, is power integrated over time (one joule equals one watt sustained for one second). Throughout this thesis, watts are used to describe instantaneous or steady state power draw (for example, idle power), while joules are used to describe the total energy consumed by a measurement run.

extrapolated to energy rankings without independent measurement.

Another notable limitation is the representation of each architectural style through a single framework or runtime configuration. In the experiments conducted by Śliwa and Pańczyk [2], Chandra and Farisi [3], and Niswar et al. [4], each protocol is implemented using one primary library or framework stack. This approach assumes that architectural behaviour is independent of implementation-specific factors such as serialisation library selection, middleware configuration, or runtime optimisations. As a result, reported performance characteristics may partially reflect framework-level optimisations rather than purely architectural properties, a conflation that limits the generalisability of findings.

To summarise the landscape of performance-focused comparative research, Table 2.1 presents the key characteristics and limitations of the primary studies reviewed in this subsection.

Table 2.1: Performance-Focused Comparative Studies: Summary and Limitations

Study	**Arch.**	**Workload**	**Key Metric(s)**	**Conc.**	**Energy**	**Limitation**
Śliwa & Pańczyk [2]	All	Simple	Throughput, data volume	✓	×	Single impl. per arch.
Chandra & Farisi [3]	All	Load & stress	Response time, CPU, mem.	✓	×	No nested/bulk distinction
Niswar et al. [4]	All	Flat & nested	Response time, CPU	✓	×	No energy measurement
Reichersdörfer [25]	All	Concurrent	Latency, throughput	✓	×	Single framework per arch.
Johansson & Isabella [26]	REST, gRPC	Short, typical, large	Response time	✓	×	No GraphQL; no energy
Oda & Yamaguchi [23]	HTTP/1.1, HTTP/2	Variable network	Throughput, latency	✓	×	Protocol only; no arch. comparison
Brito & Valente [20]	REST, GraphQL	Complex queries	Implementation effort	○	×	No gRPC; no energy

Legend: All = REST, GraphQL, gRPC; Arch. = Architectures; Conc. = Concurrency Tested; Energy = Energy Measured; ✓= Yes; ×= No; ○ = Limited/Partial

2.5.2 Sustainability-Oriented Studies

While performance-oriented comparative studies dominate the literature, a growing body of research within sustainable software engineering and Green ICT has begun to establish that software architectural decisions directly and measurably influence energy consumption. Unlike performance studies that measure time-based efficiency, sustainability-oriented research focuses on quantifying energy usage in joules or watts, identifying architectural and implementation-level factors that contribute to computational intensity, and developing methodological frameworks for reliable energy measurement. Taken together, these studies provide the conceptual and empirical foundation upon which energy-aware API comparison must be built, while simultaneously revealing how far the field still has to go.

Naumann et al. [5] introduced the GREENSOFT reference model, positioning sustainability as a first-class quality attribute in software engineering. The model highlights that software design decisions, including architectural style, data handling mechanisms, and communication protocols, can significantly affect environmental impact across the software lifecycle. Although the GREENSOFT model does not specifically address API architectures, it establishes a conceptual foundation for evaluating software systems beyond performance metrics, and its lifecycle perspective implies that energy consequences of architectural choices compound over time rather than appearing only at peak load. Verdecchia et al. [28] build on this foundation in a more recent survey of green IT and green software practices, finding that while sustainability awareness in the software industry has grown considerably, the translation of sustainability principles into concrete engineering decisions, particularly at the architectural level, remains inconsistent. Their work reinforces Bajrami's [9] observation that a gap persists between sustainability ambitions and engineering reality, and situates this gap specifically within the context of software architectural decision-making.

Procaccianti et al. [29] conducted an empirical evaluation of best practices for energy-efficient software development, demonstrating that specific architectural and implementation-level patterns produce measurable and reproducible reductions in energy consumption. Critically, their study reinforces that systematic, hardware-based measurement is necessary to validate energy efficiency claims, intuition-driven assumptions about which practices reduce energy consumption are frequently incorrect when subjected to empirical scrutiny. This finding is directly relevant to API energy comparison: without controlled, repeated measurement under equivalent conditions, observed differences in energy consumption between REST, GraphQL, and gRPC cannot be attributed reliably to architectural factors rather than implementation or environmental noise.

Pinto et al. [14] investigated developer perceptions of energy consumption in software systems through mining of Stack Overflow discussions, finding that architectural and implementation-level concerns, including data serialisation, caching strategies, and communication patterns, frequently surface as energy-related issues in practice. Their analysis reveals that energy awareness among developers is fragmented and largely intuition-driven, with practitioners rarely applying systematic measurement to validate their assumptions. This finding has direct implications for API design: if developers cannot reliably predict the energy consequences of their architectural choices without measurement, then empirical studies comparing the energy behaviour of REST, GraphQL, and gRPC under controlled conditions are not merely academically interesting, they are practically necessary.

Capra et al. [15] provided empirical evidence that software design patterns, including caching, batching, and asynchronous processing, produce measurable and reproducible differences in power usage at the application level. Importantly, their study demonstrated that improvements in execution time do not always correspond to proportional reductions in energy consumption, a finding that directly undermines

the common assumption that faster APIs are necessarily more energy-efficient. This decoupling of performance and energy efficiency establishes a critical methodological requirement: direct hardware-based energy measurement cannot be substituted by performance proxies, regardless of how detailed those proxies are.

At the communication and serialisation layer, the use of compact binary formats such as Protocol Buffers, as documented by Google [16], reduces both payload size and the computational overhead associated with encoding and decoding operations compared to text-based formats such as JSON. Smaller payloads reduce transmission time and network load, while lower encoding complexity reduces CPU cycles, both of which carry direct energy implications. However, while these efficiency characteristics are well established in terms of bandwidth and latency, Herwig and Fischer [30] demonstrated in an empirical study of REST and WebSocket energy consumption on mobile devices that communication protocol choice produces measurable differences in hardware-level energy draw, and that these differences are not always predictable from performance characteristics alone. Their work represents one of the few studies that directly measures the energy consumption of web communication protocols using hardware instrumentation, making it a methodologically important reference point for API energy research despite its focus on mobile rather than server-side contexts.

The question of how serialisation format and transport protocol choices translate into energy consumption is further complicated by the relationship between payload size and network transmission energy. Aslan et al. [31] provided empirically grounded estimates of electricity intensity for internet data transmission, establishing that transmitting one gigabyte of data over fixed-line networks consumed approximately 0.06 kWh in 2015, with this figure halving roughly every two years as network infrastructure improves. While this macro-level estimate is not directly applicable to individual API request energy measurement, it provides an impor-

tant theoretical anchor for understanding why payload size reduction, as achieved by Protocol Buffers relative to JSON, carries genuine energy consequences at scale. Coroama et al. [32] extended this line of inquiry by modelling the direct energy demand of internet data flows at a more granular level, demonstrating that energy consumption scales with the volume of data transmitted across network hops and that each intermediate node in a distributed system contributes an additive energy cost. This finding is particularly relevant to RQ4: it provides theoretical grounding for an analytical model that estimates total energy consumption across distributed environments with varying numbers of intermediate network nodes.

O'Dea et al. [8] proposed a measurement-driven framework for assessing sustainability in microservice architectures, emphasising the importance of empirical data collection, workload isolation, and fine-grained component-level energy attribution. Although the study does not specifically compare REST, GraphQL, and gRPC, it demonstrates that fine-grained energy analysis in service-oriented systems is both feasible and necessary, and that the methodological infrastructure for such analysis exists and is mature enough to support cross-architectural comparison. Salonen [33] applied similar methodological rigour specifically to REST API systems, implementing hardware-based power measurement in a controlled laboratory environment using the PowerGoblin tool [7]. Herwig and Fischer [30] provide further evidence that hardware-based energy measurement of communication protocols is both feasible and produces meaningful results, though their focus on mobile devices limits direct comparability to server-side API contexts. The study demonstrated that different workload patterns, varying request intensity and data size, produce measurable and statistically distinguishable variations in energy consumption, and highlighted the necessity of isolating workload categories and controlling environmental variables to obtain reliable energy data. However, Salonen's analysis was limited to a single architectural paradigm and did not incorporate cross-architectural

comparison or implementation-level variability, leaving the relative energy behaviour of REST, GraphQL, and gRPC under equivalent workloads unaddressed. Table 2.2 summarises the key characteristics and methodological approaches of the empirical energy measurement studies reviewed in this subsection.

Table 2.2: Empirical Energy Measurement Studies

Study	**Focus Area**	**Energy**	**Hardware**	**Cross-Arch.**	**Impl.**
Capra et al. [15]	Application-level energy	✓	○	×	○
Herwig & Fischer [30]	REST/WebSocket energy	✓	✓	○	×
O'Dea et al. [8]	Microservice sustainability	✓	✓	×	×
Salonen [33]	REST API energy	✓	✓	×	×
Procaccianti et al. [29]	Best practices / energy-efficient software	✓	○	×	○

Legend: Energy = Direct Energy Measurement; Hardware = Hardware Instrumentation; Cross-Arch. = Cross-Architectural API Comparison; Impl. = Implementation-Level Analysis; ✓= Yes; ×= No; ○ = Limited/Partial/Experimental

The question of whether optimisation strategies can reduce API energy consumption remains largely unexplored in the empirical literature. Zafari et al. [34] developed an analytical optimisation framework demonstrating that jointly optimising caching, compression, and communication can improve energy efficiency by up to 88% compared to optimising any two of these dimensions in isolation. However, this work was conducted in the context of wireless sensor networks rather than web API systems, and its applicability to REST, GraphQL, and gRPC contexts has not

been empirically validated. At the application layer, Capra et al. [15] showed that caching and batching reduce energy consumption in software systems, but did not evaluate these strategies across different API architectural styles or measure their differential impact on binary versus text-based serialisation formats.

Beyond application-layer studies, Hilty and Aebischer [10] discussed the systemic impact of ICT infrastructure on sustainability, emphasising that communication layers and distributed architectures contribute significantly to overall energy demand. While such macro-level work provides important contextual grounding, it does not offer architecture-specific analytical expressions applicable to API communication patterns. Similarly, distributed systems research proposes generalised energy-per-bit estimation approaches, as evidenced by both Aslan et al. [31] and Coroama et al. [32], but these models are rarely grounded in experimentally derived architectural measurements specific to API workloads. Table 2.3 summarises the conceptual, survey, and modelling studies that form the broader sustainability foundation of this review.

Table 2.3: Conceptual, Survey and Modelling Studies

Study	**Focus Area**	**Energy**	**Modelling**	**Cross-Arch.**	**Scope**
Naumann et al. [5]	Sustainable software model	×	×	×	Lifecycle
Verdecchia et al. [28]	Green IT and software survey	×	×	×	Industry-wide
Pinto et al. [14]	Developer energy concerns	×	×	×	Developer practices
Google LLC [16]	Serialisation efficiency	×	×	○	Protocol-level

Study	Focus Area	Energy	Modelling	Cross-Arch.	Scope
Aslan et al. [31]	Network transmission energy	○	✓	×	Network-level
Coroama et al. [32]	Internet data flow energy	○	✓	×	Network-level
Zafari et al. [34]	Caching / compression optimisation	✓	✓	×	System-level
Hilty & Aebischer [10]	ICT sustainability	×	○	×	Macro-level

Legend: Energy = Direct Energy Measurement; Modelling = Analytical Modelling; Cross-Arch. = Cross-Architectural API Comparison; ✓= Yes; ×= No; ○ = Partial/Empirical model/Conceptual

2.6 Research Gap and Rationale

The synthesis of performance-focused and sustainability-oriented research reveals a consistent and consequential pattern: the two bodies of literature have developed largely in isolation from one another, each addressing a different dimension of API system evaluation without bridging the gap between them. Performance-focused studies provide a well-established empirical understanding of latency, throughput, and scalability trade-offs among REST, GraphQL, and gRPC [2], [3], [4], [23], [25], [26], yet remain silent on the energy consequences of the architectural and protocol-level differences they document. Sustainability-oriented research demonstrates that software architectural decisions measurably influence energy consumption [5], [14],

[15], [28], [29] and that hardware-based energy measurement in API contexts is both feasible and necessary [30], [33], yet stops short of applying these insights to cross-architectural comparison. Network-level modelling work provides energy-per-bit and energy-per-hop estimation frameworks [31], [32] that could underpin scalability analysis, yet these models are not grounded in experimentally derived API-specific measurements. The result is a three-way methodological disconnect that leaves the energy implications of API architectural choice fundamentally unresolved.

Three specific gaps emerge from this analysis. First, no existing study has conducted a direct, hardware-based energy comparison of REST, GraphQL, and gRPC under equivalent and systematically varied workload conditions. The studies closest to this, Salonen [33] and Oda and Yamaguchi [23], measure energy consumption of individual API paradigms or communication protocols in isolation, but do not place them in direct comparison under identical experimental conditions. Herwig and Fischer [30] provide further evidence that hardware-based energy measurement of communication protocols is both feasible and produces meaningful results, though their focus on mobile devices limits direct comparability to server-side API contexts. Second, existing comparative studies represent each architecture through a single implementation or framework, assuming that architectural behaviour is independent of implementation-level factors. This assumption has not been empirically tested, it is entirely possible that framework-level differences within the same architectural paradigm produce energy variations comparable in magnitude to cross-architectural differences. Third, while network-level energy models exist [31], [32] and optimisation frameworks have demonstrated the theoretical potential of caching and compression strategies [34], neither has been applied to API architecture comparison in a way that connects experimentally measured energy behaviour to scalable distributed environment estimation.

Beyond these empirical gaps, a methodological gap also persists. As Pereira et al. [27] demonstrated, performance metrics cannot serve as reliable proxies for energy consumption, yet this is precisely the assumption on which the existing comparative API literature implicitly rests. Until energy is measured directly and independently of performance, rankings of architectural efficiency based on latency and throughput cannot be translated into rankings of architectural sustainability. This methodological shortcut has practical consequences: developers and architects making API design decisions on sustainability grounds currently have no reliable empirical basis for those decisions.

Collectively, these gaps define the research space that this study occupies. By conducting a controlled, hardware-based energy comparison of REST, GraphQL, and gRPC across multiple implementations spanning the frontend, backend, and network components of a full-stack API interaction, this research directly addresses the cross-architectural measurement gap. By systematically varying payload size across six levels and concurrent client load across eleven levels, it addresses the gap in understanding how workload characteristics amplify or invert architectural energy differences. And by conducting an empirical multi-hop network topology experiment and deriving an analytical framework grounded in its measured per-hop energy costs, it bridges the disconnect between empirical measurement and scalability modelling for distributed environments with varying numbers and types of intermediate network nodes.

Figure 2.1 illustrates how these four contributions position this research at the intersection of the three bodies of literature identified above. The three circles correspond to the performance-focused comparative studies, sustainability-oriented energy research, and network-level energy modelling literature reviewed in this chapter, each annotated with its associated references. The four contribution boxes are placed in the central region where all three circles overlap, reflecting that this re-

search draws simultaneously on all three bodies of literature rather than extending any single one in isolation.

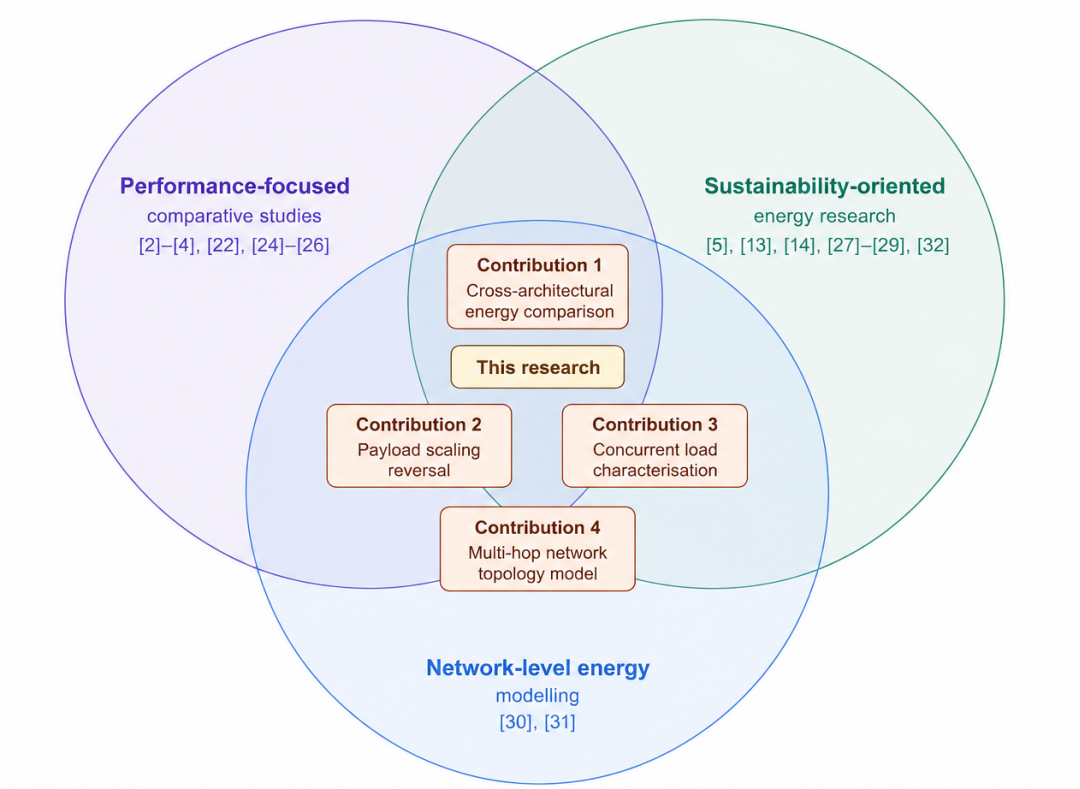


Figure 2.1: Positioning of the research contributions at the intersection of the three literature streams.

3 Empirical Setup and Implementation

3.1 Overview of the Empirical Study

This chapter presents the empirical component of the research, detailing the experimental design, measurement infrastructure, API implementations, workload scenarios, and data collection procedures used to investigate the energy consumption behaviour of REST, GraphQL, and gRPC API architectures. The empirical study is structured around four research questions. RQ1 establishes a baseline energy consumption comparison across all six API implementations under identical workload conditions. RQ2 investigates how payload size affects energy consumption across the three selected representative implementations, one per architectural paradigm. RQ3 examines how concurrent client load affects energy consumption across the same three implementations. RQ4 extends the measurement methodology to a multi-hop physical network topology, empirically examining how additional intermediate network nodes, including wired switches and a wireless bridge, affect total energy consumption.

All measurements were conducted in a controlled laboratory environment using hardware-based power meters connected to the PowerGoblin measurement framework, which enables direct instrumentation of electrical energy consumed by the

devices under test. The study follows a controlled experimental design in which a single independent variable is varied at a time while all other conditions are held constant, ensuring that observed differences in energy consumption can be attributed to the factor under investigation rather than environmental variation.

3.2 PowerGoblin Measurement Framework

The energy measurements in this study were conducted using PowerGoblin, a hardware-based energy measurement framework developed at the University of Turku as part of the Visiiri Green ICT project [7]. PowerGoblin provides a coordinated infrastructure for performing repeatable, hardware-instrumented energy measurements across distributed multi-node systems. It integrates physical power meters, software-based resource collectors, and a centralised measurement controller into a unified platform that manages measurement sessions, synchronises timestamps across nodes, and generates structured energy reports.

The framework operates through a controller instance running on a dedicated host machine, which communicates with lightweight agents deployed on each node under measurement. Each agent manages the execution of measurement scripts on its respective node, initialises resource collectors, and coordinates with the controller to ensure that energy data is captured within well-defined measurement boundaries [35]. Measurement sessions are described using structured plan files, which specify the active nodes, resource collection parameters, scripts to execute, and the number of measurement runs. Measurements are triggered through the PowerGoblin web interface, which enqueues commands to the agents [35]. The agents then execute the defined scripts while the controller records energy data from the connected hardware meters.

Hardware energy readings are obtained through HardKernel SmartPower 3 me-

ters[1], which sample electrical power at a configurable rate and transmits measurements to the PowerGoblin controller over the local network. Resource utilisation data, including CPU usage, memory consumption, and network traffic, is collected in parallel through collectd[2], a system statistics daemon configured and managed by the PowerGoblin agent on each node. The combination of hardware meter readings and software resource data provides a comprehensive view of energy consumption at both the device and component level.

For this study, PowerGoblin was configured to measure energy across two nodes simultaneously: the frontend client node and the backend server node. Each measurement run was explicitly delimited by API calls to the PowerGoblin controller, signalling the start and end of each run so that energy readings could be attributed precisely to the workload under test. The resulting data was analysed using the PowerGoblin report interface, which provides per-run energy totals, statistical summaries, and visualisations including box plots and scatter plots.

3.3 Experimental Setup

The experimental environment consists of three physical machines connected over a local area network. The first machine, referred to as the controller, is a Dell OptiPlex Micro 7010 running the PowerGoblin instance and serving as the central measurement coordinator. The second machine, referred to as odroid, is a HARDKERNEL ODROID-H3+ equipped with an Intel N6005 processor and 16 GB of RAM, running Arch Linux. This node serves as the frontend client, executing the load generation software that sends API requests to the backend. The third machine, referred to as odroid2, is a HARDKERNEL ODROID-H4 equipped with an Intel N97 processor and 8 GB of RAM, also running Arch Linux. This node serves as the backend server,

[1] `https://wiki.odroid.com/accessory/power_supply_battery/smartpower3`

[2] `https://collectd.org/`

hosting the API implementations under test.

Power consumption is measured using HardKernel SmartPower 3 hardware meters. Each SmartPower 3 unit provides two independently switchable output channels, allowing a single meter to measure two separate devices simultaneously, and samples voltage, current, and power at a configurable rate of up to 200 Hz (5 ms interval), with a typical measurement tolerance of 3%. The device accepts an input voltage of 9–21 V DC and supplies up to 3 A per output channel, and measurements are transmitted to the host machine over a USB-C connection. A single meter unit, identified as SP3-ea, is connected to both odroid and odroid2 through separate output channels, out1 for odroid and out2 for odroid2, enabling simultaneous energy measurement of both nodes within a single measurement session. The meter-to-channel mapping was established through calibration in the PowerGoblin user interface prior to measurements, and the `initialization-strategy: IncludeActive` setting was used to automatically include the calibrated channels in each session.

In addition to the SP3-ea meter, a second SmartPower 3 meter, identified as SP3-96, powers the network switch (referred to as switch1) that connects odroid and odroid2, with its energy reported on output channel out1 (referred to as `odroid/network`). This switch is the network device nearest to odroid and is present in the measurement path for RQ1, RQ2, and RQ3. For RQ4, the experimental setup is extended with additional network infrastructure, a second switch (switch2) and a WiFi bridge, each instrumented with its own SmartPower 3 meter channel; this extended setup is described separately in Section 3.5.4.

All API backend implementations are deployed as Docker [36] containers on odroid2, with each container running on host networking mode and listening on port 5000. Docker images are built prior to measurement sessions so that image build time is excluded from energy readings. The load client is similarly containerised and

executed on odroid during measurements. Backend containers are started manually before each measurement session and remain running throughout the measurement, ensuring that no startup overhead is included in the recorded energy data.

Table 3.1: Hardware specifications of the measurement nodes

Node	**Hardware**	**Role**	**Specifications**
controller	Dell OptiPlex Micro 7010	PowerGoblin host	–
odroid	HARDKERNEL ODROID-H3+	Frontend / Client	Intel N6005, 16 GB RAM
odroid2	HARDKERNEL ODROID-H4	Backend / Server	Intel N97, 8 GB RAM

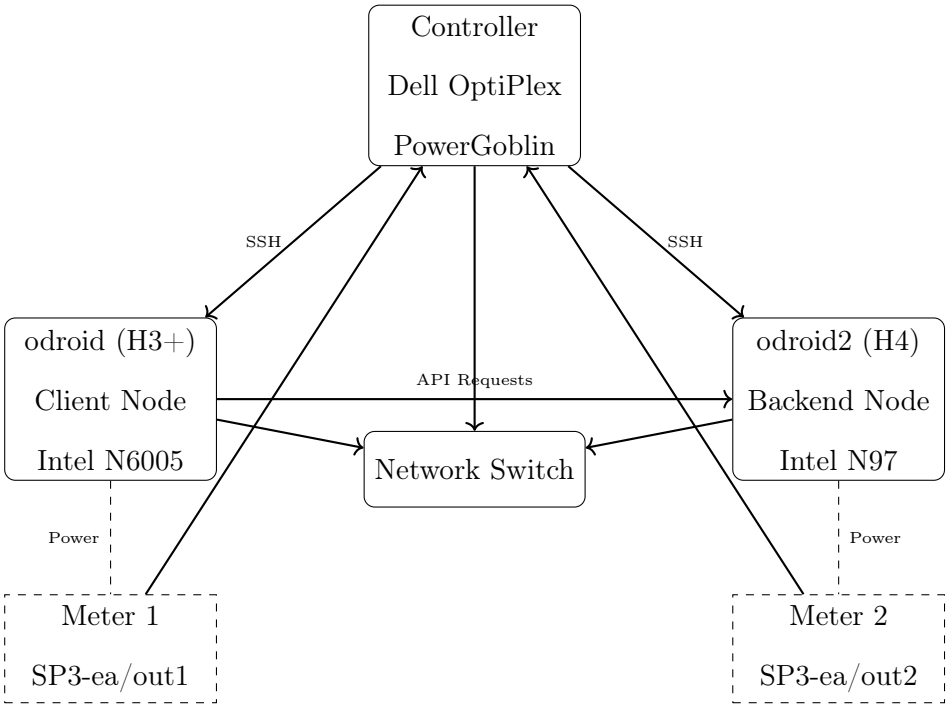


Figure 3.1: Physical measurement setup showing the controller, client node (odroid), backend node (odroid2), SmartPower 3 power meters, and local area network connections.

The two nodes communicate over a gigabit local area network via a dedicated network switch. The physical measurement setup is illustrated in Figure 3.1 and

the logical software architecture of the measurement infrastructure is shown in Figure 3.2. The controller accesses odroid and odroid2 via SSH, and the PowerGoblin web interface is accessed through a PuTTY SSH tunnel from an external workstation.

Table 3.1 summarises the hardware specifications of the three measurement nodes. The physical connections between the nodes, meters, and network infrastructure are illustrated in Figure 3.1.

3.3.1 Idle Power Baseline

For RQ4, the per-hop energy estimation model (Section 4.4.1) requires isolating the workload-attributable energy of each network component from its background power draw. For this purpose, an idle power baseline was established for each measured channel by recording the steady-state power draw of each device while no workload was running.

The idle power values obtained were: odroid2/cpu = 5.979 W, odroid/cpu = 6.442 W, and odroid/network (switch1) = 4.298 W. For the additional RQ4 channels, switch2, wifi1, and wifi2, idle power was independently cross-validated using Use Case 1 (Section 3.5.4), in which these devices were physically connected but not part of the active request path; the resulting idle power values (switch2 = 2.460 W, wifi1 = 5.602 W, wifi2 = 5.243 W) were consistent with separately obtained calibration measurements for the same channels.

For each RQ4 measurement run, the net (idle-corrected) energy attributable to the workload is calculated as:

$$E_{net} = E_{raw} - P_{idle} \times t_{run} \tag{3.1}$$

where E_{raw} is the total energy reported by PowerGoblin for the run, P_{idle} is the idle power of the corresponding channel, and t_{run} is the wall-clock duration of the

run. Because idle power is a steady-state rate rather than an accumulated quantity, this correction is valid regardless of run duration, provided the device remains in steady idle state when not under load.

For RQ1 and RQ2, energy values are reported as raw totals as recorded by PowerGoblin, without idle correction. This is consistent with the comparative nature of these research questions: since the idle power contribution is a property of the measurement hardware rather than of the API implementation under test, it is common to all implementations measured within a given session and does not affect the validity of relative comparisons between implementations, which is the basis of the RQ1 and RQ2 analyses. Idle correction is applied only for RQ4 (Section 4.4.1), where isolating the network-attributable energy of each hop is necessary for the additive per-hop estimation model.

3.4 API Implementation Details

All six API implementations were developed as functionally equivalent .NET 8 [37] applications, each exposing a single endpoint that retrieves a user record together with a configurable number of associated posts and comments from a shared SQLite database [38]. The implementations share a common data model and database schema, defined in a shared project, ensuring that any observed differences in energy consumption reflect architectural and framework-level characteristics rather than differences in business logic or data access complexity.

The database is seeded with a single user record, 640 posts, and 640 comments per post, yielding a maximum dataset of 409,600 comment records. Data retrieval is performed using raw SQL queries via Microsoft.Data.Sqlite [39], bypassing the Entity Framework Core ORM entirely. This design decision was motivated by the need to ensure that only the requested number of records is fetched from the database, with `LIMIT` clauses applied directly in SQL. Using an ORM with eager loading would

load the entire dataset into memory before applying any filtering, which would invalidate the payload scaling experiment by introducing a constant and architecture-independent data access overhead regardless of the requested payload size.

Each implementation accepts two parameters, `postsLimit` and `commentsLimit`, that control the number of posts and comments returned per request. These parameters are passed by the load client through the appropriate mechanism for each protocol, as described in the subsections below.

3.4.1 REST Implementations

The REST architecture is represented by two implementations built on ASP.NET Core [40]. The first, *Rest.Controllers*, uses the conventional MVC controller pattern, where a dedicated `UsersController` class handles routing and request processing through attribute-based routing. The second, *Rest.MinimalApi*, uses the ASP.NET Core Minimal API pattern introduced in .NET 6, where endpoints are registered directly in the application startup code without a dedicated controller class. Both implementations expose the endpoint `GET /users/{id}/with-posts-and-comments` and accept `posts` and `comments` as query parameters. Responses are serialised as JSON using the built-in `System.Text.Json` serialiser. Both implementations operate over HTTP/1.1.

3.4.2 GraphQL Implementations

The GraphQL architecture is represented by two implementations using different .NET GraphQL libraries. The first, *GraphQL.GraphQLNet*, is built on the GraphQL.NET library [41], which follows a schema-first approach where types and queries are defined programmatically using C# classes. The query accepts `posts` and `comments` as integer arguments, which are passed through to the data access layer to limit the records returned. The second, *GraphQL.HotChocolate*, is built

on the Hot Chocolate library [42] developed by ChilliCream, which supports both annotation-based and code-first schema definition. Both implementations expose a single `POST /graphql` endpoint and accept queries conforming to the GraphQL specification [19]. Responses are serialised as JSON. Both implementations operate over HTTP/1.1.

3.4.3 gRPC Implementations

The gRPC architecture is represented by two implementations using different approaches to service definition. The first, *Grpc.AspNetCore*, uses the contract-first approach provided by the official Google gRPC library for ASP.NET Core [43]. The service contract is defined in a Protocol Buffers [16] `.proto` file, from which client and server code is automatically generated at build time. The `UserRequest` message includes `posts_limit` and `comments_limit` fields that control the number of records returned. The second, *Grpc.ProtobufNet*, uses the code-first approach provided by the protobuf-net.Grpc library [44], where the service contract is defined as a C# interface decorated with attributes, eliminating the need for a separate `.proto` file. Both implementations operate over HTTP/2 with unencrypted connections, as TLS is not required in the controlled local area network environment used for this study.

Table 3.2 provides a summary of all six implementations together with their corresponding libraries and frameworks.

Table 3.2: Six API implementations used in the study

Architecture	**Implementation**	**Library / Framework**
REST	Rest.Controllers	ASP.NET Core Controllers [40]
REST	Rest.MinimalApi	ASP.NET Core Minimal APIs [40]
GraphQL	GraphQL.GraphQLNet	GraphQL.NET [41]
GraphQL	GraphQL.HotChocolate	Hot Chocolate [42]
gRPC	Grpc.AspNetCore	Grpc.AspNetCore [43]
gRPC	Grpc.ProtobufNet	protobuf-net.Grpc [44]

3.5 Workload Scenario

The workload scenario used across all measurements is modelled on a social feed retrieval use case, in which a client application requests a user profile together with a list of posts and their associated comments. This scenario was selected because it represents a realistic and commonly occurring data retrieval pattern in web and mobile applications, and because it exercises the core architectural characteristics of each protocol, including serialisation, request routing, and nested data resolution, in a meaningful and comparable way.

Each request targets a single user with identifier 1 and retrieves a configurable number of posts and their associated comments. The load client is configured with two parameters, `postsLimit` and `commentsLimit`, which are passed to the backend through the appropriate protocol-specific mechanism. For REST implementations, these are passed as HTTP query parameters. For GraphQL implementations, they are passed as query variables in the request body. For gRPC implementations, they are included as fields in the `UserRequest` protobuf message.

The load client is implemented as a .NET 8 console application and executed as a Docker container on the odroid frontend node. It communicates with the PowerGob-

lin controller to signal the start and end of each measurement run, ensuring that energy readings are attributed precisely to the request processing window. Prior to each measurement session, a configurable number of warmup requests are sent without PowerGoblin signalling to allow the backend to reach a stable operating state [45], including JIT compilation, connection pool initialisation, and database file caching, before energy recording begins.

For RQ1 and RQ2, the load client sends requests sequentially, one request at a time, waiting for each response before issuing the next. Each measured run consists of 100 such sequential requests, with the PowerGoblin run-start and run-stop signals bracketing all 100 requests; the reported per-run energy and time values therefore correspond to the cumulative cost of 100 sequential requests rather than a single request. Aggregating 100 requests into a single run serves to average out transient sources of measurement noise, such as operating system scheduling jitter, garbage collection pauses, and momentary fluctuations in background system activity, that would otherwise dominate the energy and timing signal of an individual request; the energy cost of a single request is small relative to these transient effects, whereas their influence is proportionally reduced when amortised over 100 consecutive requests. For RQ3, the load generation approach differs and is described separately in Section 3.5.3.

3.5.1 RQ1 Workload

For the baseline comparison (RQ1), all six implementations were measured under identical conditions. Each measurement session consisted of 10 runs with 5 warmup runs, using a fixed payload of 10 posts and 10 comments per post (`postsLimit=10`, `commentsLimit=10`). This payload level represents a realistic small-to-medium response size and serves as the common baseline for RQ1, RQ2, and RQ3. The logical software architecture of the measurement flow, which applies equally to RQ1 and

RQ2, is illustrated in Figure 3.2.

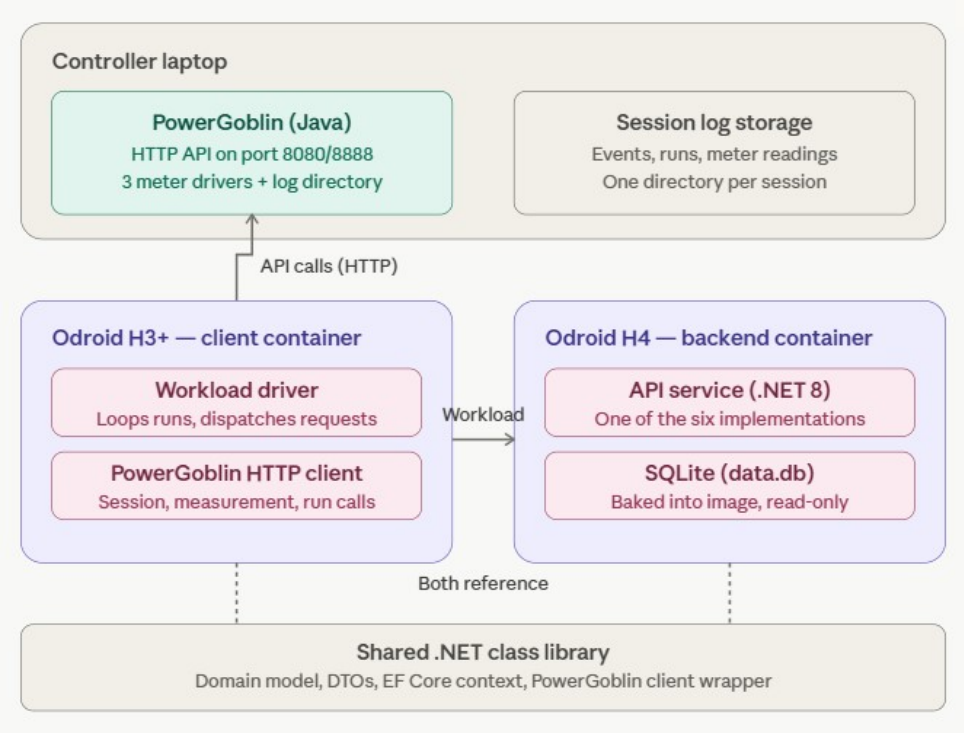


Figure 3.2: Logical software architecture of the sequential measurement flow used for RQ1 and RQ2. The .NET load client on odroid dispatches requests to the backend on odroid2 and signals run boundaries to PowerGoblin on the controller.

3.5.2 RQ2 Workload

For the payload size experiment (RQ2), three representative implementations were selected, one per architectural paradigm: *Rest.Controllers*, *GraphQL.GraphQLNet*, and *Grpc.AspNetCore*. These were chosen as the primary representative of each architecture, as they are the most widely studied variants in the literature [3], [4], [25]. Each implementation was measured across six payload levels, following a doubling sequence: 10, 20, 40, 80, 160, and 320 posts and comments per post. This sequence was chosen to reveal whether energy consumption scales linearly or exhibits non-linear behaviour as payload size increases. Each payload level was measured with 10 runs and 5 warmup runs. The payload level of 10 posts and 10 comments corresponds to the RQ1 baseline and serves as the reference point for the scaling analysis.

A seventh level (640 posts and comments per post) was originally planned but was excluded due to the time cost of measurement: each run consists of 100 sequential requests (Section 3.5), and at 320 posts and comments per post a single run already takes between 45 and 95 seconds depending on the implementation, making the 640-level condition impractical within the time available for this study.

To characterise the actual data volume associated with each payload level, the response size in bytes was measured for each of the three RQ2 implementations at each payload level. For Rest.Controllers and GraphQL.GraphQLNet, the response size corresponds to the raw HTTP response body, measured using `curl`. For Grpc.AspNetCore, the response size corresponds to the serialised Protobuf message size plus the 5-byte gRPC framing header, obtained via `CalculateSize()` on the response message. Table 3.3 presents the resulting payload sizes.

Table 3.3: Response payload size in bytes for each RQ2 implementation across the six payload levels

Payload	**Rest.Controllers**	**GraphQL.GraphQLNet**	**Grpc.AspNetCore**
P10	12,903	12,921	8,705
P20	48,606	48,624	32,765
P40	189,054	189,072	127,645
P80	748,899	748,917	504,845
P160	3,019,813	3,019,831	2,033,033
P320	12,176,384	12,176,402	8,213,993

The REST and GraphQL payload sizes are nearly identical at every level, differing by a constant 18 bytes corresponding to the GraphQL response envelope (`{"data":{"user":{...}}}`). The gRPC Protobuf payload is consistently approximately 67.4% the size of the equivalent REST/GraphQL JSON payload at every level, a 33% reduction rather than the substantially larger reductions sometimes reported for Protobuf relative to JSON, likely because the response is dominated

by variable-length text fields (post and comment bodies, timestamps), where Protobuf's compact integer encoding and absence of field names yield comparatively smaller savings.

3.5.3 RQ3 Workload

For the concurrency experiment (RQ3), the same three representative implementations were used as in RQ2. The payload was fixed at the baseline level (`postsLimit=10`, `commentsLimit=10`) to isolate the effect of concurrency from payload size. Rather than the sequential load client used in RQ1 and RQ2, concurrent load was generated using k6 [46], an open-source load testing tool developed by Grafana Labs. This approach is consistent with the methodology used in prior comparative API performance studies [3], which similarly employed k6 to simulate concurrent virtual users.

Each k6 test script spawned a configurable number of virtual users (VUs), each independently and continuously sending requests to the backend for a fixed duration of 30 seconds. The concurrency levels tested were 5, 10, 15, 20, 25, 30, 100, 500, 1000, 2000, and 4000 virtual users. PowerGoblin measurement boundaries were signalled from within the k6 script using the PowerGoblin HTTP API, with a single measurement and run boundary wrapping the entire 30-second test window. The session identifier required by the PowerGoblin API was passed to the k6 script as an environment variable by the PowerGoblin agent at runtime. Unlike RQ1 and RQ2, which were each measured over 10 runs, every RQ3 concurrency level was measured in a single 30-second window; each reported RQ3 value is therefore a single point estimate rather than a mean over repeated runs, and within-level measurement variability is not quantified for this experiment.

The concurrent measurement flow, including the role of k6, the virtual users, the network, and the PowerGoblin energy recording, is illustrated in Figure 3.3.

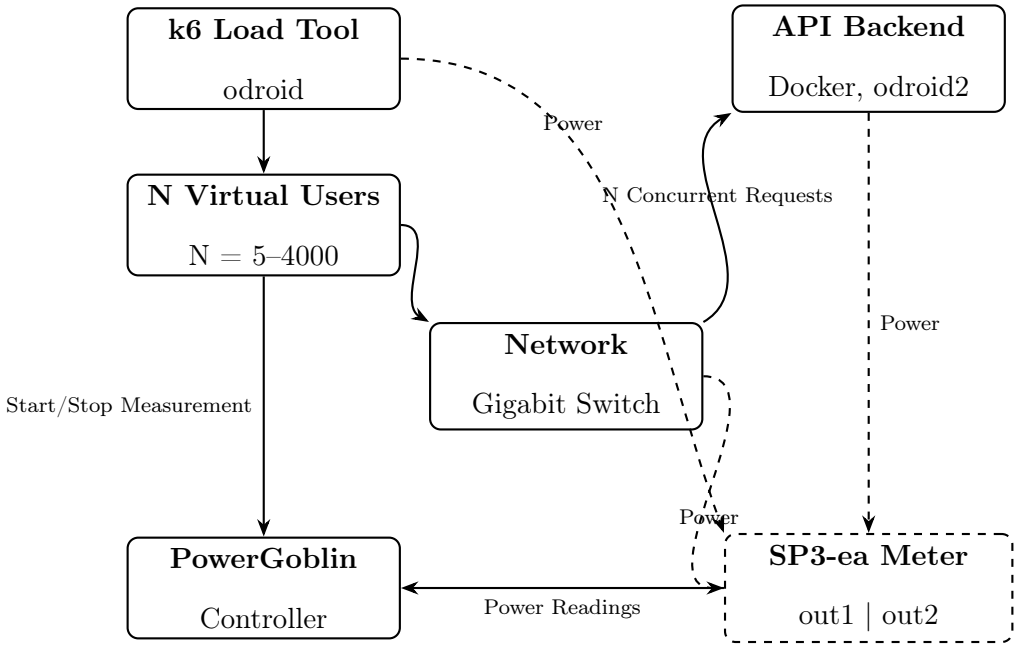


Figure 3.3: Concurrent measurement flow for RQ3. k6 spawns N virtual users that send concurrent requests to the backend via the network switch for 30 seconds. PowerGoblin records total energy consumption from both compute nodes and the network infrastructure simultaneously.

3.5.4 RQ4 Workload

For RQ4, the measurement methodology is extended from the single-switch topology used in RQ1–RQ3 to a set of multi-hop physical network topologies, in order to empirically examine how additional intermediate network nodes affect total energy consumption. A single representative implementation, Rest.Controllers, was used with a fixed payload of 320 posts and 320 comments per post (P320). Unlike RQ1 and RQ2, each measurement run consists of 5 sequential requests rather than 100, following the same overall session structure (10 runs, 5 warmup runs) as RQ1 and RQ2.

Three network topologies (use cases) were measured:

- **UC1:** odroid – switch1 – odroid2 (one switch hop, the same configuration used in RQ1–RQ3).

- **UC2:** odroid – switch1 – switch2 – odroid2 (two switch hops).
- **UC3:** odroid – switch1 – WiFi bridge – WiFi bridge – switch2 – odroid2 (one switch hop plus a wireless bridge hop).

Both switches are Linksys LGS116 unmanaged switches. switch1 is the same switch used throughout RQ1–RQ3, positioned nearest to odroid, with approximately 10 ports actively connected as part of the wider laboratory network. switch2 is a second LGS116 switch with only one or two ports connected, introduced specifically for the RQ4 topologies. For UC3, the wireless hop is provided by an Asus RT-AX53U (AX1800) router running OpenWrt 24.10.5, configured as a WDS (Wireless Distribution System) bridge over 802.11ac (2x2 MIMO, 40 MHz channel width, WPA2 encryption) at a separation distance of approximately 10 cm between the two bridge endpoints.

The other devices connected to switch1 as part of the wider laboratory network are not running any active services nor actively sending requests during measurement; occasional DHCP traffic can occur, but the lease times involved are long and the footprint per packet is small, and the practice of measuring multiple runs per condition (Section 3.6) is expected to mitigate any resulting fluctuations in switch1's readings. As shown by the idle power baseline established in Section 3.3.1, these background connections do increase the static power consumption of switch1 relative to an unconnected switch, which is accounted for by calibrating the idle baseline directly from switch1 in its normal, connected state.

Each additional network device is instrumented with its own SmartPower 3 meter channel: switch2's energy is recorded on a dedicated channel, and the two endpoints of the WDS wireless bridge are recorded as separate channels, referred to as wifi1 and wifi2. In UC1, switch2, wifi1, and wifi2 are physically connected but not part of the active request path, allowing their idle power to be cross-validated as described in Section 3.3.1.

3.6 Measurement Metrics and Data Collection

The primary metric collected in this study is total energy consumption in joules, recorded separately for each node over the duration of each measurement run. PowerGoblin derives total energy by integrating the instantaneous power readings obtained from the SmartPower 3 hardware meters over the run duration. For each measurement session, the following energy metrics are recorded:

- **Total energy (odroid/cpu) [J]:** The total electrical energy consumed by the frontend client node (odroid) during the measurement run, as recorded by the SP3-ea meter on output channel out1. This metric captures the energy associated with load generation, HTTP request construction, serialisation of outgoing requests, and deserialisation of incoming responses on the client side.

- **Total energy (odroid/network) [J]:** The total electrical energy consumed by switch1, the network switch nearest to odroid, as recorded by the SP3-96 meter on output channel out1 (Section 3.3). This metric reflects the energy associated with network switching activity along the path between the frontend and backend nodes.

- **Total energy (odroid2/cpu) [J]:** The total electrical energy consumed by the backend server node (odroid2) during the measurement run, as recorded by the SP3-ea meter on output channel out2. This metric is most directly reflective of the server-side processing cost of each API architecture, including request routing, deserialisation, query execution, data retrieval, and response serialisation.

- **Time per run [s]:** The wall-clock duration of each measurement run, from the PowerGoblin run start signal to the run stop signal.

- **Total energy (switch2, wifi1, wifi2) [J]:** For RQ4 only, the total elec-

trical energy consumed by the second switch and the two endpoints of the WiFi bridge, as recorded by their respective SmartPower 3 meter channels (Section 3.5.4).

Throughout this thesis, the `odroid/cpu` and `odroid2/cpu` channels are referred to as *frontend* and *backend CPU energy*, respectively. Each SP3-ea channel is connected inline on the corresponding node's DC power input and therefore records the node's total board-level energy; `cpu` denotes the PowerGoblin channel label rather than an isolated processor power rail. Because each node runs no other significant workload during a measurement session, differences in this node-level energy across implementations are attributable primarily to the cost of request processing, which is why it serves as the basis for the relative comparisons in RQ1–RQ3.

Each measurement session produces one data point per run, yielding 10 data points per implementation per experimental condition. Statistical summaries, including the mean, standard deviation, minimum, maximum, and interquartile range, are computed by PowerGoblin across the runs within each session and presented in the measurement reports. These statistics are used as the basis for cross-architectural and cross-condition comparisons in Chapter 4.

In addition to hardware meter readings, PowerGoblin collects software-based resource utilisation metrics through collectd on each node, including CPU user time, CPU system time, CPU idle time, memory usage, and network interface octets. These metrics are recorded at the collectd sampling interval and provide supporting context for interpreting the hardware energy readings, though the primary analysis is based on the hardware meter data.

All raw measurement data is stored in PowerGoblin session logs on the controller node. Results are accessed and reviewed through the PowerGoblin report interface, which renders measurement reports from structured Markdown plan files and presents energy statistics alongside box plot and scatter plot visualisations. Due

to a known issue with the `comparison.csv` export in the version of PowerGoblin used in this study, cross-session comparisons were performed by manually extracting the summary statistics from individual `report.html` files generated for each measurement session.

3.6.1 Sources of Measurement Error and Reliability Assessment

Several sources of error affect the measurement chain used in this study. The SmartPower 3 meters carry a manufacturer-specified tolerance of approximately 3% (Section 3.3), which applies uniformly across implementations and sets a practical lower bound on what differences can be considered meaningful. The idle power baseline (Section 3.3.1) was calibrated at a single point in time, and minor drift in ambient conditions or background system activity is a plausible contributor to the small positive and negative net values observed at low payload levels in RQ1 and RQ2, and at switch1 in RQ4 (Section 4.4), where background laboratory traffic adds further noise (Section 3.5.4). On the measured nodes, .NET JIT compilation [45], garbage collection, and OS scheduling jitter introduce transient timing and energy effects that are not specific to any one architecture; these are mitigated by warmup requests and, for RQ1 and RQ2, by aggregating 100 sequential requests per run (Section 3.5). PowerGoblin's run-boundary signalling also introduces a small, fixed latency common to all sessions. Finally, RQ3 concurrency levels are each measured in a single 30-second window rather than 10 repeated runs (Section 3.5.3), so within-condition variability cannot be quantified for that experiment; this is the most significant reliability limitation identified in this study and is discussed further in Section 5.5.

Despite these error sources, RQ1 backend CPU energy shows a coefficient of variation of approximately 5–12% across implementations, falling below 1% at P320 in RQ2 as 100-request aggregation averages out transient noise. One exception is

GraphQL.GraphQLNet's frontend energy at P80 (Table 4.3), with a coefficient of variation of approximately 18%; this is treated as isolated run-to-run variability rather than a systematic issue, as it does not affect the ordering observed at that payload level. Given this consistently low variability and the largely implementation-independent nature of the identified error sources, the RQ1, RQ2, and RQ4 results are considered reliable for the relative comparisons underpinning this study's conclusions, while RQ3 warrants more cautious interpretation given the absence of repeated-run estimates.

3.7 Data Analysis Approach

The data analysis in this study follows a descriptive and comparative approach, grounded in the energy measurements collected through the PowerGoblin framework. For each experimental condition, defined by the combination of implementation, payload level, and concurrency level, the primary unit of analysis is the mean total energy consumption per run, reported in joules, together with the standard deviation across the 10 runs of that session. The standard deviation serves as an indicator of measurement stability and is used to assess the reliability of the recorded values.

For RQ1, the six implementations are compared directly on the basis of their mean energy consumption across the three measured metrics, frontend CPU energy, backend network energy, and backend CPU energy, under identical baseline conditions. The comparison is presented in tabular form and supported by box plot visualisations generated by PowerGoblin, which show the distribution of per-run energy values across the 10 runs of each session. Differences between implementations are discussed in terms of their magnitude relative to the standard deviation, with particular attention to whether observed differences are large enough to be considered meaningful given the measurement variability.

For RQ2, the analysis examines how energy consumption scales across the six payload levels for each of the three selected implementations. For each implementation, the mean backend CPU energy and frontend CPU energy are plotted against payload size to reveal the scaling behaviour, whether energy grows linearly, sub-linearly, or super-linearly with increasing payload. Cross-architectural comparisons are made by overlaying the scaling curves of the three implementations on the same plot, allowing direct visual comparison of how each architecture responds to increasing data volumes.

For RQ3, the analysis examines how total energy consumption changes as the number of concurrent virtual users increases across the eleven tested concurrency levels. The primary metric of interest is backend CPU energy (odroid2/cpu), as this most directly reflects the server-side cost of handling concurrent connections. For each architecture, the energy recorded in the single 30-second measurement window is plotted against concurrency level. The shapes of the resulting curves, and in particular whether any architecture exhibits a plateau, a linear increase, or a super-linear increase, are used to characterise the concurrency scaling behaviour of each architectural paradigm.

Where relevant, the analysis contextualises the observed energy differences with reference to the architectural characteristics identified in Chapter 2, including serialisation format, transport protocol, and request handling mechanisms, to provide a theoretically grounded interpretation of the empirical findings.

For RQ4, the analysis compares the net energy consumption of each component (frontend client, backend server, and intermediate network devices) across the three topologies (UC1, UC2, UC3), in order to derive an empirical per-hop energy cost for each hop type, an additional wired switch hop versus a wireless bridge hop. These empirically derived per-hop costs are then related to the additive per-hop energy model proposed by Coroama et al. [32], examining whether the assumption of a

uniform additive cost per hop holds, or whether hop type is a significant factor that the model should account for.

For RQ1 and RQ2, where 10 independent runs are available per condition, the descriptive comparisons above are supplemented by formal statistical hypothesis testing, described in Section 3.7.1. For RQ3 and RQ4, comparisons remain descriptive, on the basis of visual inspection of the measurement distributions; for RQ3 this is a direct consequence of the single-run-per-condition design (Section 3.5.3), which precludes any test of within-condition variability, while for RQ4 the limited number of measured topologies (three) is judged too small to support a meaningful formal test. This selective approach is consistent with comparable empirical energy measurement studies [29], [30], [33], which similarly rely primarily on descriptive comparison given small per-condition sample sizes.

3.7.1 Statistical Testing Procedure for RQ1 and RQ2

For RQ1 and RQ2, where each experimental condition (a given implementation, and for RQ2 a given payload level) is measured across repeated independent runs, the descriptive comparisons above are supplemented by formal statistical hypothesis testing on the underlying per-run energy values.

Normality of the per-run distributions was first assessed using the Shapiro-Wilk test, applied separately to each implementation-condition-metric combination. A substantial number of these distributions deviated significantly from normality ($p < 0.05$), which is unsurprising given the small sample size (n=10 for most conditions, n=5 for the RQ2 P320 condition, Section 3.5.2) and the presence of a brief warm-up tail in the first one or two runs of several sessions, despite the warm-up requests issued prior to each measurement session (Section 3.5). Given this, the non-parametric Kruskal-Wallis H-test [47] was selected as the primary test for comparing energy consumption across implementations, rather than a one-way ANOVA, since

it does not assume normally distributed samples and is more robust to the small, occasionally skewed samples present in this dataset.

For RQ1, a Kruskal-Wallis test was conducted across the six implementations, separately for each of the three measured metrics (frontend CPU energy, backend CPU energy, and network energy), under the fixed baseline payload. For RQ2, a Kruskal-Wallis test was conducted across the three selected implementations, separately for each metric and at each of the six payload levels, to determine whether the cross-architectural energy differences observed descriptively in Section 4.2 are statistically distinguishable from random variation at each stage of payload scaling.

Where a Kruskal-Wallis test indicated a statistically significant difference among groups ($p < 0.05$), Dunn's post-hoc test [48] was applied to identify which specific pairs of implementations differ significantly, with the resulting p-values adjusted for multiple comparisons using the Bonferroni correction. This two-stage procedure, an omnibus test followed by a corrected pairwise post-hoc test, is the standard approach for multi-group non-parametric comparison and avoids the inflated false-positive risk of running uncorrected pairwise tests directly.

In addition to the group-comparison tests, the payload-scaling relationship investigated in RQ2 (Section 4.2) was quantified using log-log linear regression for each of the three selected implementations: the natural logarithm of backend CPU energy was regressed on the natural logarithm of payload size, yielding a scaling exponent b such that $E \approx a \cdot \text{payload}^b$. A scaling exponent of $b \approx 1$ indicates linear scaling, while $b > 1$ indicates super-linear scaling, providing a quantitative complement to the qualitative scaling descriptions (linear, sub-linear, super-linear) already used in the per-implementation comparisons.

All statistical tests were conducted in Python using the SciPy [49] and scikit-posthocs [50] libraries, on the per-run energy values extracted directly from the PowerGoblin session data underlying the summary tables presented in Chapter 4,

rather than from the reported means and standard deviations alone. A significance threshold of $\alpha = 0.05$ is used throughout.

4 Results and Analysis

4.1 Baseline Energy Consumption Comparison (RQ1)

This section presents the results of the baseline energy consumption measurements conducted to address RQ1: *How do REST, GraphQL, and gRPC architectures differ in baseline energy consumption?* All six implementations were measured under identical conditions using a fixed payload of 10 posts and 10 comments per post, with 10 measurement runs and 5 warmup runs per session. Values reported are raw totals as recorded by PowerGoblin (Section 3.3.1); since the idle power contribution of the measurement hardware is common to all implementations measured within a session, it does not affect the validity of relative comparisons between implementations. The results are summarised in Table 4.1 and illustrated through per-implementation box plots in Figures 4.1 through 4.6.

The results in Table 4.1 show that energy consumption per run is modest across all implementations and follows a consistent pattern across the three measured metrics. Backend CPU energy (odroid2/cpu) ranges from 5.995 J (GraphQL.HotChocolate) to 7.338 J (Rest.Controllers), a spread of 1.343 J. Frontend client energy (odroid/cpu) ranges from 4.947 J (GraphQL.HotChocolate) to 5.433 J (Grpc.ProtobufNet), a spread of 0.486 J. Network energy (odroid/network) ranges from 2.362 J (GraphQL.HotChocolate) to 2.861 J (Rest.Controllers), a spread

of 0.499 J. Mean run duration ranges from 0.558 s (GraphQL.HotChocolate) to 0.644 s (Rest.Controllers), with the two GraphQL implementations completing the baseline workload fastest and the two REST implementations slowest.

Table 4.1: Baseline energy consumption across all six implementations (RQ1). Values are mean ± standard deviation over 10 runs.

Implementation	**odroid/cpu (J)**	**odroid/network (J)**	**odroid2/cpu (J)**	**Time (s)**
Rest.Controllers	5.406 ± 0.156	2.861 ± 0.023	7.338 ± 0.471	0.644
Rest.MinimalApi	5.209 ± 0.420	2.625 ± 0.042	6.763 ± 0.358	0.607
GraphQL.GraphQLNet	5.092 ± 0.393	2.473 ± 0.038	6.156 ± 0.580	0.575
GraphQL.HotChocolate	4.947 ± 0.355	2.362 ± 0.026	5.995 ± 0.714	0.558
Grpc.AspNetCore	5.346 ± 0.159	2.560 ± 0.028	6.602 ± 0.421	0.595
Grpc.ProtobufNet	5.433 ± 0.186	2.627 ± 0.037	6.833 ± 0.437	0.613

For comparison with other studies, where the absolute idle power draw of the measurement hardware will generally differ, Table 4.2 additionally reports the load-induced (idle-corrected) energy for each metric, calculated as $E_{net} = E_{raw} - P_{idle} \times t$ using the idle power values established in Section 3.3.1 (odroid2/cpu = 5.979 W, odroid/cpu = 6.442 W, odroid/network = 4.298 W). The table also expresses each implementation's net backend CPU energy as a percentage relative to GraphQL.HotChocolate, the implementation with the lowest value for this metric, providing a hardware-independent measure of the relative differences between implementations.

Table 4.2: Load-induced (idle-corrected) energy consumption across all six implementations (RQ1), calculated as raw energy minus idle power × duration. The rightmost column expresses net backend CPU energy (odroid2/cpu) as a percentage relative to GraphQL.HotChocolate, the lowest-consuming implementation for this metric.

Implementation	**odroid/cpu net (J)**	**odroid/ network net (J)**	**odroid2/cpu net (J)**	**odroid2/cpu rel. (%)**
Rest.Controllers	1.257	0.093	3.488	+31.2%
Rest.MinimalApi	1.299	0.016	3.134	+17.9%
GraphQL.GraphQLNet	1.388	0.002	2.718	+2.2%
GraphQL.HotChocolate	1.352	−0.036	2.659	0.0%
Grpc.AspNetCore	1.513	0.003	3.044	+14.5%
Grpc.ProtobufNet	1.484	−0.008	3.168	+19.1%

The net values in Table 4.2 confirm the same ordering as the raw values in Table 4.1, with GraphQL.HotChocolate and GraphQL.GraphQLNet recording the lowest net backend CPU energy and Rest.Controllers the highest, approximately 31% higher than GraphQL.HotChocolate. As discussed in Section 3.3.1, the net network energy values are small and, in three of the six implementations, slightly negative, consistent with net network energy at this payload level being dominated by idle-baseline measurement noise rather than a workload-attributable signal.

The backend CPU energy metric (odroid2/cpu) is the most relevant for cross-architectural comparison, as it directly reflects the server-side processing cost of each implementation. GraphQL.HotChocolate records the lowest backend CPU energy at 5.995 ± 0.714 J, followed by GraphQL.GraphQLNet at 6.156 ± 0.580 J, Grpc.AspNetCore at 6.602 ± 0.421 J, Rest.MinimalApi at 6.763 ± 0.358 J, Grpc.ProtobufNet at 6.833 ± 0.437 J, and Rest.Controllers at 7.338 ± 0.471 J. A consistent pattern is visible across all three metrics: the two GraphQL implemen-

tations record the *lowest* values, while Rest.Controllers records the *highest* backend CPU and network energy of all six implementations. Notably, as shown in RQ2 (Section 4.2), this ordering does not persist as payload size increases - GraphQL's resolver-based execution model becomes the dominant cost driver at larger payloads, while gRPC emerges as the most efficient architecture across all three metrics at the largest payload tested. As before, the standard deviations of all six implementations overlap substantially, so these baseline differences should be interpreted as a starting point for the scaling investigation in RQ2 rather than a definitive ranking. This is confirmed by formal statistical testing (Section 3.7.1): a Kruskal-Wallis test across the six implementations finds no statistically significant difference in frontend CPU energy ($H(5) = 2.44$, $p = 0.785$) or network energy ($H(5) = 8.29$, $p = 0.141$) at this baseline payload, but does find a significant difference in backend CPU energy ($H(5) = 14.20$, $p = 0.014$). A Dunn's post-hoc test with Bonferroni correction shows that, of the 15 pairwise comparisons among the six implementations, only Rest.Controllers and GraphQL.HotChocolate differ significantly in backend CPU energy ($p = 0.014$, Cohen's $d = 2.16$, a large effect), consistent with these two implementations recording the highest and lowest backend CPU energy respectively in Table 4.1. All other pairwise comparisons, including those involving the gRPC implementations, are not statistically distinguishable at this baseline payload, reinforcing that the RQ1 ordering should be read as indicative rather than definitive.

The box plot visualisations in Figures 4.1 through 4.6 further illustrate the per-run distribution of energy values, confirming that measurement variability is moderate and consistent across implementations for all three metrics.

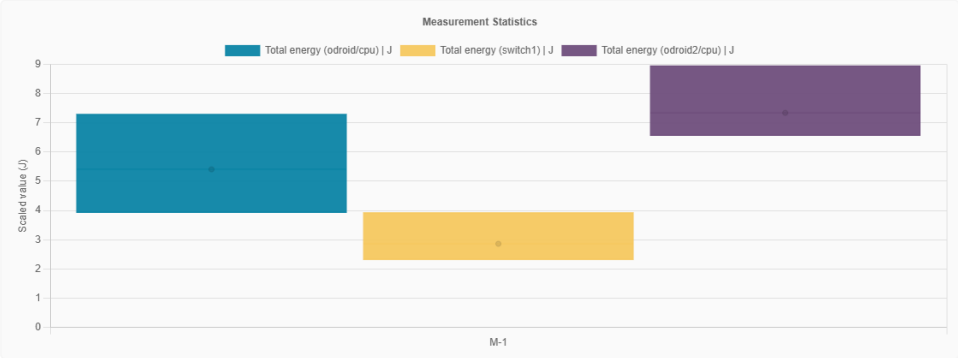


Figure 4.1: Energy consumption distribution for Rest.Controllers across 10 measurement runs. Box plots show total energy per run for odroid/cpu, odroid/network, and odroid2/cpu metrics.

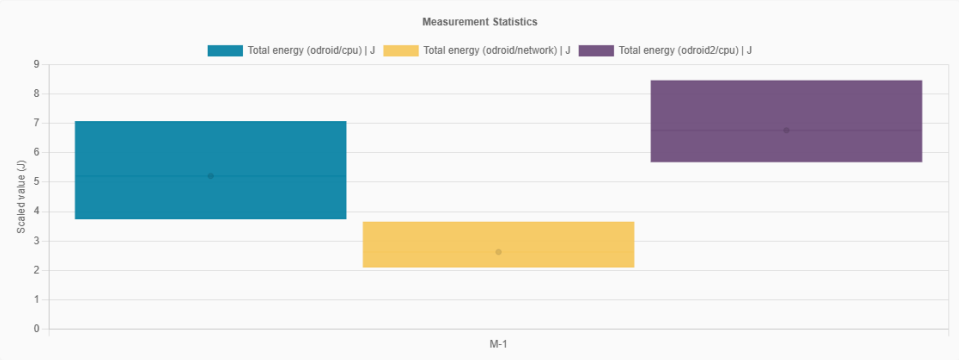


Figure 4.2: Energy consumption distribution for Rest.MinimalApi across 10 measurement runs.

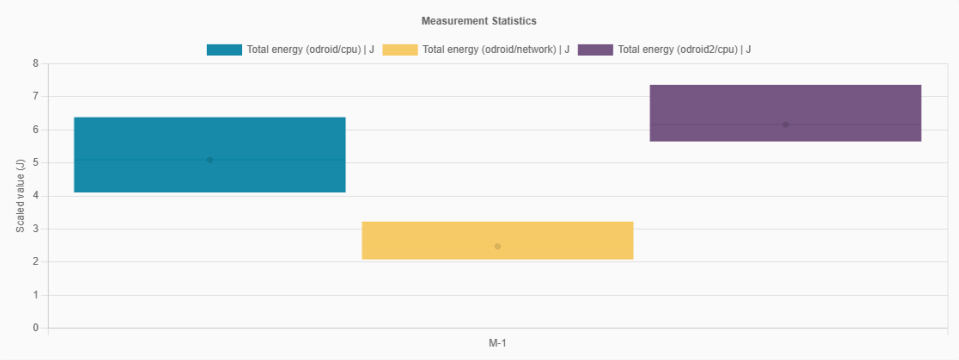


Figure 4.3: Energy consumption distribution for GraphQL.GraphQLNet across 10 measurement runs.

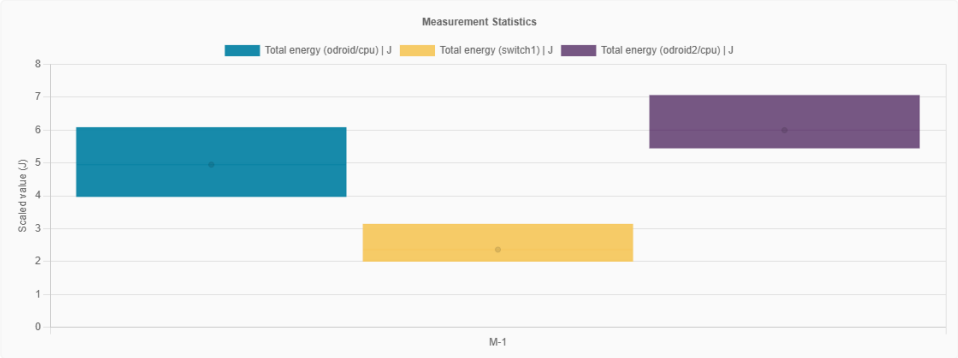


Figure 4.4: Energy consumption distribution for GraphQL.HotChocolate across 10 measurement runs.

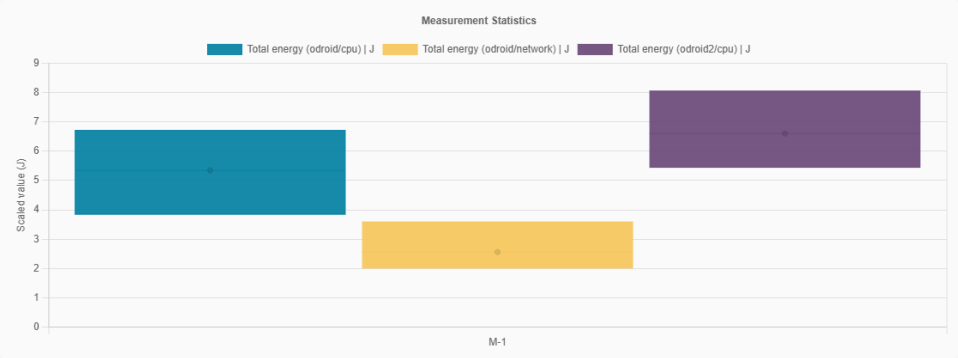


Figure 4.5: Energy consumption distribution for Grpc.AspNetCore across 10 measurement runs.

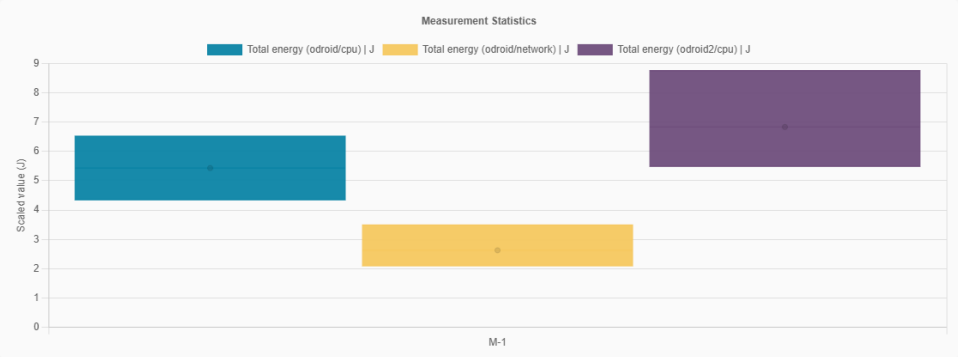


Figure 4.6: Energy consumption distribution for Grpc.ProtobufNet across 10 measurement runs.

The overall finding for RQ1 is that the six implementations show small but consistent differences across all three energy metrics at this small fixed payload.

The two GraphQL implementations record the lowest energy consumption across frontend, backend, and network metrics, while Rest.Controllers records the highest backend and network energy, though all differences remain modest relative to the measurement variability. These findings motivate the investigation of payload scaling (RQ2), which reveals that this baseline ordering does not persist as payload size increases, and of concurrent load (RQ3) as an additional factor that may amplify or alter architectural energy differences observed at baseline.

4.2 Effect of Payload Size on Energy Consumption (RQ2)

This section presents the results of the payload scaling experiment conducted to address RQ2: *How does payload size affect energy consumption across REST, GraphQL, and gRPC API architectures?* Three representative implementations were selected, Rest.Controllers, GraphQL.GraphQLNet, and Grpc.AspNetCore, one per architectural paradigm. Each was measured across six payload levels following a doubling sequence: 10, 20, 40, 80, 160, and 320 posts and comments per post (Section 3.5.2). The payload level of 10 corresponds to the RQ1 baseline. Values reported are raw totals as recorded by PowerGoblin (Section 3.3.1); as in RQ1, the idle power contribution is common to all implementations within a session and does not affect relative comparisons. The complete results are presented in Tables 4.3, 4.4, and 4.5, and the scaling behaviour is illustrated in Figures 4.7, 4.8, and 4.9.

Table 4.3: Frontend CPU energy (odroid/cpu) in joules across payload levels for RQ2. Values are mean ± standard deviation over 10 runs.

Payload	**Rest.Controllers**	**GraphQL.GraphQLNet**	**Grpc.AspNetCore**
P10	5.406 ± 0.156	5.092 ± 0.393	5.346 ± 0.159
P20	8.676 ± 0.989	8.387 ± 0.931	7.935 ± 1.607
P40	19.99 ± 1.605	23.43 ± 2.031	17.77 ± 2.804
P80	49.65 ± 2.366	65.57 ± 11.81	46.30 ± 2.759
P160	165.7 ± 1.517	210.5 ± 1.164	151.1 ± 2.650
P320	609.8 ± 5.766	800.8 ± 5.352	300.9 ± 2.904

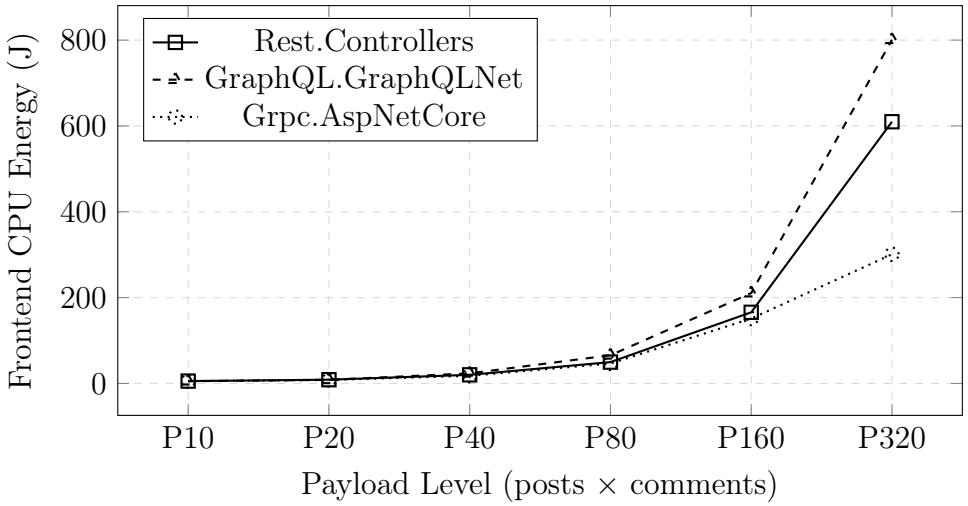


Figure 4.7: Frontend CPU energy consumption (odroid/cpu) across six payload levels for the three selected implementations.

Table 4.4: Backend CPU energy (odroid2/cpu) in joules across payload levels for RQ2. Values are mean ± standard deviation over 10 runs.

Payload	**Rest.Controllers**	**GraphQL.GraphQLNet**	**Grpc.AspNetCore**
P10	7.338 ± 0.471	6.156 ± 0.580	6.602 ± 0.421
P20	11.23 ± 0.555	12.63 ± 0.430	11.22 ± 1.968
P40	29.53 ± 0.229	36.09 ± 0.300	27.16 ± 0.747
P80	75.76 ± 0.815	103.7 ± 1.419	77.29 ± 0.421
P160	241.4 ± 2.387	349.5 ± 2.915	239.3 ± 2.231
P320	872.7 ± 6.370	1331 ± 9.577	722.5 ± 2.416

Table 4.5: Network energy (odroid/network) in joules across payload levels for RQ2. Values are mean ± standard deviation over 10 runs.

Payload	**Rest.Controllers**	**GraphQL.GraphQLNet**	**Grpc.AspNetCore**
P10	2.861 ± 0.023	2.473 ± 0.038	2.560 ± 0.028
P20	4.721 ± 0.326	4.402 ± 0.358	4.159 ± 0.716
P40	11.27 ± 0.125	13.08 ± 0.281	9.752 ± 0.545
P80	25.92 ± 0.547	33.09 ± 0.909	26.15 ± 0.254
P160	79.54 ± 0.299	107.5 ± 0.450	76.03 ± 0.646
P320	284.8 ± 1.384	407.4 ± 1.372	193.3 ± 0.348

To normalise energy consumption by the actual data volume transferred, Table 4.6 expresses backend CPU energy as joules per kilobyte of response payload, using the payload sizes reported in Table 3.3.

Table 4.6: Backend CPU energy normalised by response payload size (J/KB) across payload levels for RQ2.

Payload	**Rest.Controllers**	**GraphQL.GraphQLNet**	**Grpc.AspNetCore**
P10	0.582	0.488	0.777
P20	0.237	0.266	0.351
P40	0.160	0.196	0.218
P80	0.104	0.142	0.157
P160	0.082	0.119	0.121
P320	0.073	0.112	0.090

Energy per kilobyte decreases substantially for all three implementations as payload size increases, from approximately 0.49–0.78 J/KB at P10 to 0.07–0.11 J/KB at P320. This reflects the amortisation of fixed per-request overhead, such as connection handling and request parsing, over an increasingly large response body, so that the marginal energy cost of each additional kilobyte of data is substantially lower than the cost implied by small payloads. At small payload levels, Grpc.AspNetCore has the highest energy per kilobyte despite having the smallest payload size in bytes, consistent with the RQ1 finding that gRPC carries comparatively higher fixed per-request overhead at low data volumes; at the largest payload level, Rest.Controllers achieves the lowest energy per kilobyte, while GraphQL.GraphQLNet remains the least efficient by this measure across all payload levels, consistent with its steeper overall scaling observed in Figure 4.8.

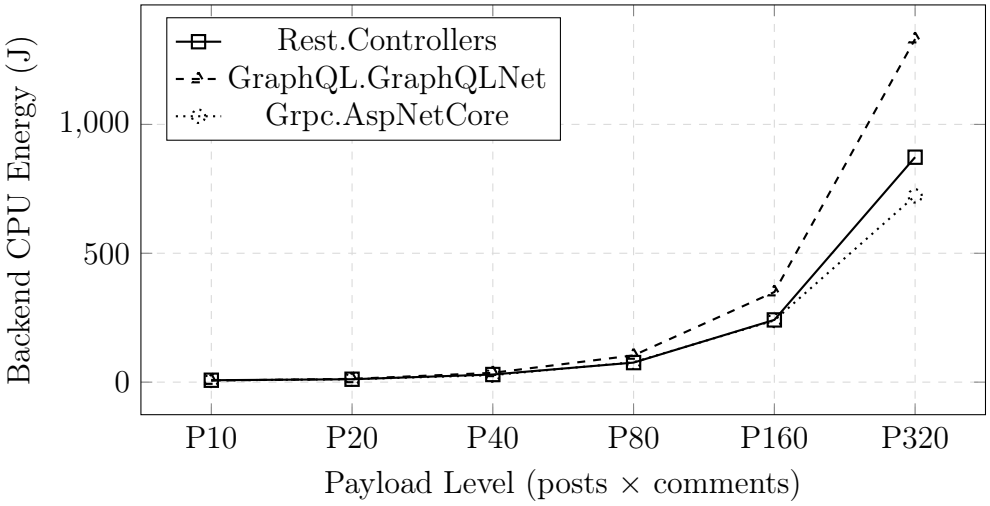


Figure 4.8: Backend CPU energy consumption (odroid2/cpu) across six payload levels for the three selected implementations. GraphQL.GraphQLNet diverges most steeply, while Rest.Controllers and Grpc.AspNetCore scale more similarly to one another.

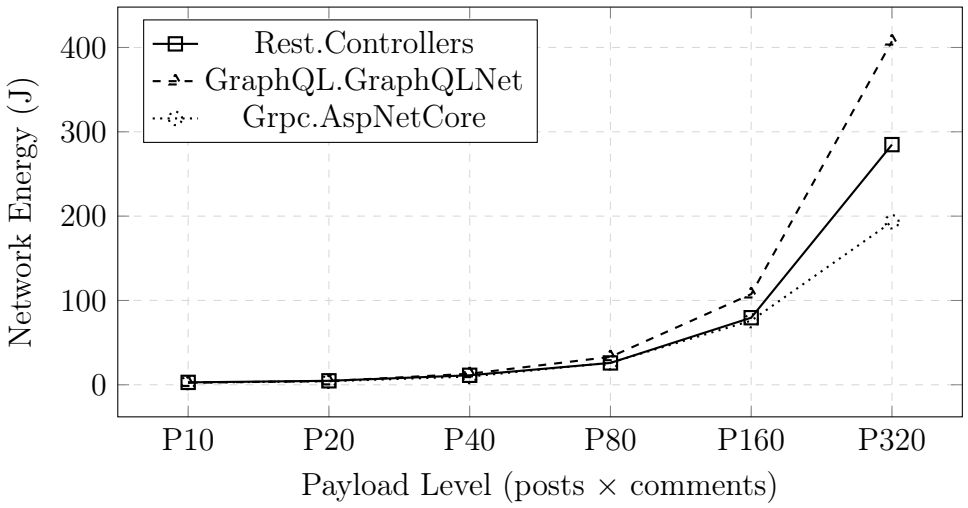


Figure 4.9: Network energy consumption (odroid/network) across six payload levels for the three selected implementations. Grpc.AspNetCore consistently records the lowest network energy from P40 onwards, reflecting the compactness of Protobuf binary serialisation.

The RQ1 baseline (P10) shows GraphQL.GraphQLNet as the most energy-efficient of the three selected implementations across all three metrics (frontend 5.092 J, backend 6.156 J, network 2.473 J), with Rest.Controllers the least efficient across all three (frontend 5.406 J, backend 7.338 J, network 2.861 J) and

Grpc.AspNetCore in an intermediate position. As payload size increases, this ordering inverts. By P20, Grpc.AspNetCore has already become the most efficient implementation on the frontend and network metrics, and is essentially tied with Rest.Controllers on the backend metric. By P40, the ordering has fully inverted relative to P10 across all three metrics: Grpc.AspNetCore is the most efficient, GraphQL.GraphQLNet the least efficient, and Rest.Controllers occupies the intermediate position, an ordering that holds, and intensifies, through P320.

As illustrated in Figure 4.8, GraphQL.GraphQLNet's backend CPU energy diverges most steeply from the other two implementations. At P320, it reaches 1331 ± 9.577 J, approximately 84.2% higher than Grpc.AspNetCore (722.5 ± 2.416 J) and approximately 52.5% higher than Rest.Controllers (872.7 ± 6.370 J). This pattern is consistent with GraphQL's dynamic resolver execution model, in which each field in the response is resolved individually at runtime: as the number of posts and comments grows, resolver execution overhead scales proportionally, resulting in disproportionately higher CPU energy consumption relative to the other architectures. Rest.Controllers and Grpc.AspNetCore scale more similarly to one another at the backend, with Rest.Controllers approximately 20.8% higher than Grpc.AspNetCore at P320, a substantially narrower gap than either implementation's gap to GraphQL.GraphQLNet.

This inversion is supported by formal statistical testing (Section 3.7.1). A Kruskal-Wallis test across the three implementations finds a statistically significant difference in backend CPU energy at every payload level tested, from P10 ($H(2) = 7.53$, $p = 0.023$) through P320 ($H(2) = 12.50$, $p = 0.002$), with the significance strengthening through the middle payload levels (P40: $H(2) = 25.81$, $p < 0.001$). A Dunn's post-hoc test with Bonferroni correction shows that the specific pair driving this result shifts as payload increases: at P10, only Rest.Controllers and GraphQL.GraphQLNet differ significantly ($p = 0.021$), consistent with the RQ1

baseline ordering; from P40 onward, GraphQL.GraphQLNet and Grpc.AspNetCore differ significantly instead ($p = 0.033$ at P40, strengthening to $p < 0.001$ at P160 and $p = 0.001$ at P320), directly supporting the claim that GraphQL.GraphQLNet and Grpc.AspNetCore exchange positions as the least and most efficient implementations as payload size grows. This same Grpc.AspNetCore-versus-GraphQL.GraphQLNet pairing is also the only one to reach statistical significance consistently across the frontend and network energy metrics from P40 onward, mirroring the pattern observed for backend CPU energy.

The super-linear scaling visible in Figure 4.8 is further quantified using log-log linear regression of backend CPU energy against payload size (Section 3.7.1). All three implementations exhibit a scaling exponent greater than 1, confirming super-linear growth: Rest.Controllers ($b = 1.366$, 95% CI $[1.311, 1.421]$, $R^2 = 0.978$), Grpc.AspNetCore ($b = 1.370$, 95% CI $[1.320, 1.420]$, $R^2 = 0.982$), and GraphQL.GraphQLNet ($b = 1.533$, 95% CI $[1.493, 1.573]$, $R^2 = 0.991$). GraphQL.GraphQLNet's scaling exponent is the largest of the three, and its confidence interval does not overlap with either Rest.Controllers' or Grpc.AspNetCore's, providing a quantitative, statistically grounded confirmation that GraphQL.GraphQLNet's backend CPU energy grows disproportionately faster with payload size than the other two implementations, consistent with the steeper divergence visible in Figure 4.8.

The frontend CPU energy (Table 4.3, Figure 4.7) and network energy (Table 4.5, Figure 4.9) follow a similar pattern, with Grpc.AspNetCore's advantage becoming most pronounced at P320. At this payload level, Grpc.AspNetCore's frontend energy (300.9 J) is approximately 50.7% lower than Rest.Controllers (609.8 J) and approximately 62.4% lower than GraphQL.GraphQLNet (800.8 J); its network energy (193.3 J) is approximately 32.1% lower than Rest.Controllers (284.8 J) and approximately 52.5% lower than GraphQL.GraphQLNet (407.4 J). The network en-

ergy advantage of Grpc.AspNetCore is consistent with the compactness of Protocol Buffers binary serialisation [16], which produces payloads approximately 67% the size of the equivalent JSON payloads used by REST and GraphQL (Table 3.3), resulting in lower data transmission volumes and correspondingly lower network energy consumption. The frontend energy advantage reflects the lower deserialisation cost on the client side when receiving Protobuf-encoded responses, as binary decoding of compact messages requires less CPU work than parsing equivalent JSON payloads.

For comparison with studies using different measurement hardware, Tables 4.7, 4.8, and 4.9 additionally report the load-induced (idle-corrected) energy for each metric, calculated as $E_{net} = E_{raw} - P_{idle} \times t$ using the idle power values established in Section 3.3.1. Because the run duration t differs between implementations at a given payload level, the idle subtraction $P_{idle} \times t$ also differs, so relative percentages computed from net values are not identical to those computed from raw values; the percentages quoted throughout this section (e.g. GraphQL.GraphQLNet consuming 84.2% more backend CPU energy than Grpc.AspNetCore at P320) are derived from the raw values in Tables 4.3, 4.4, and 4.5, and are used consistently throughout this thesis. The net values in Tables 4.7–4.9 are presented to support comparison of absolute joule figures with studies conducted on different hardware, where the idle power draw of the measurement equipment used in this study would not apply.

Table 4.7: Load-induced (idle-corrected) frontend CPU energy (odroid/cpu) in joules across payload levels for RQ2, calculated as raw energy minus idle power × duration.

Payload	**Rest.Controllers**	**GraphQL.GraphQLNet**	**Grpc.AspNetCore**
P10	1.257	1.388	1.513
P20	1.603	1.739	1.690
P40	3.086	3.724	3.166
P80	10.914	15.922	7.120
P160	47.038	49.579	37.334
P320	185.659	192.031	9.399

Table 4.8: Load-induced (idle-corrected) backend CPU energy (odroid2/cpu) in joules across payload levels for RQ2, calculated as raw energy minus idle power × duration.

Payload	**Rest.Controllers**	**GraphQL.GraphQLNet**	**Grpc.AspNetCore**
P10	3.488	2.718	3.044
P20	4.665	6.460	5.424
P40	13.841	17.800	13.606
P80	39.808	57.620	40.926
P160	131.267	200.145	133.711
P320	479.043	765.985	451.950

Table 4.9: Load-induced (idle-corrected) network energy (odroid/network) in joules across payload levels for RQ2, calculated as raw energy minus idle power × duration.

Payload	**Rest.Controllers**	**GraphQL.GraphQLNet**	**Grpc.AspNetCore**
P10	0.093	0.002	0.003
P20	0.002	−0.034	−0.007
P40	−0.008	−0.068	0.008
P80	0.076	−0.035	0.010
P160	0.371	0.136	0.127
P320	1.820	1.239	−1.184

As with RQ1, the net network energy values at P10–P80 are close to zero and occasionally negative, consistent with net network energy at these payload levels reflecting idle-baseline measurement noise rather than a workload-attributable signal (Section 3.3.1). The net backend CPU values in Table 4.8 preserve the same ordering and inflection pattern as the raw values: Grpc.AspNetCore and Rest.Controllers track closely together across all payload levels, while GraphQL.GraphQLNet diverges sharply from P80 onwards.

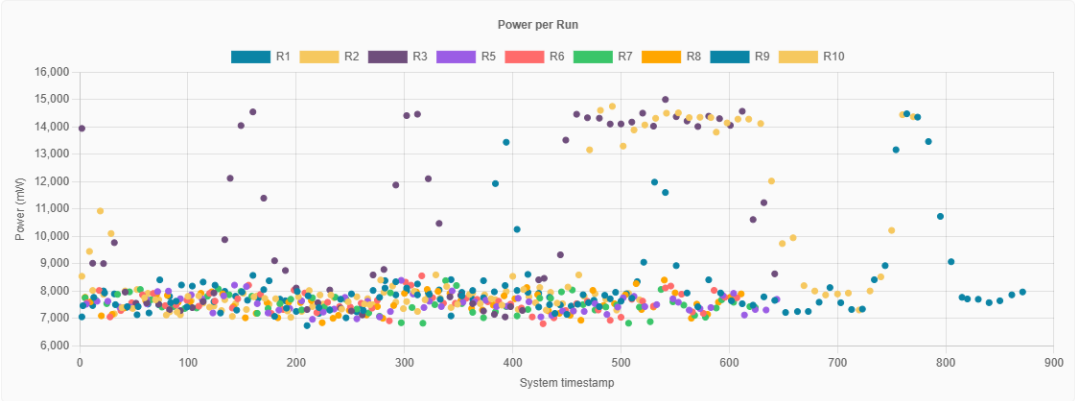
The frontend and network net values at P320 for Grpc.AspNetCore (9.399 J and −1.184 J respectively) are correct but disproportionately small compared to the other two implementations at the same payload level. This is an artefact of the idle-correction arithmetic rather than an anomaly in the underlying measurement: Grpc.AspNetCore's raw frontend energy at P320 (300.9 J) is the smallest of the three implementations, yet its run still takes 45.25 s to complete (100 sequential requests over a large Protobuf payload), so the idle-power term ($P_{idle} \times t \approx 291.5$ J) consumes approximately 96.9% of its raw total, compared to 69.6% for Rest.Controllers and 76.0% for GraphQL.GraphQLNet at the same payload level. Because Grpc.AspNetCore combines the lowest raw energy with a non-negligible run duration, it is the implementation most sensitive to this correc-

tion, leaving very little energy once the idle baseline is subtracted; the same effect drives its network net value slightly negative. This further illustrates why the raw values, rather than these net values, are used as the basis for the relative percentage comparisons in this section.

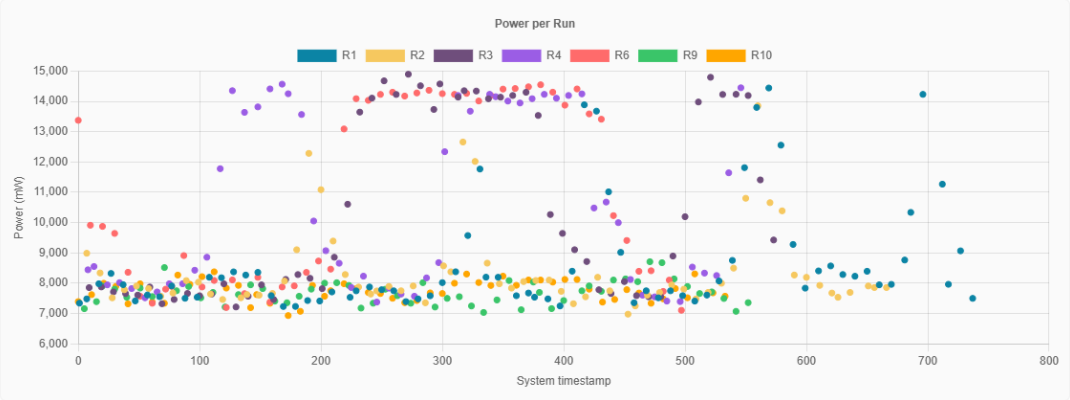
4.2.1 Per-Run Distribution of RQ2 Measurements

To complement the mean-based comparison above, scatter plots of individual per-run energy values are presented for the smallest (P10) and largest (P320) payload levels for each of the three implementations, illustrating both the magnitude of the scaling effect and the run-to-run variability at each extreme.

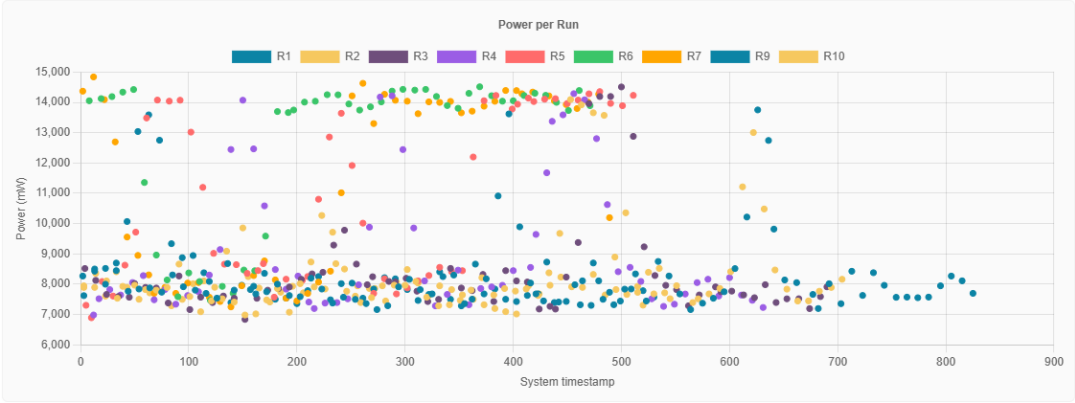
Figure 4.10 groups the three P10 scatter plots together (Rest.Controllers, GraphQL.GraphQLNet, and Grpc.AspNetCore), each covering the short run durations (approximately 0.56–0.64 s) characteristic of the smallest payload level, allowing the per-run distributions at this payload level to be compared directly across implementations on a common timescale.



(a) Rest.Controllers at P10



(b) GraphQL.GraphQLNet at P10

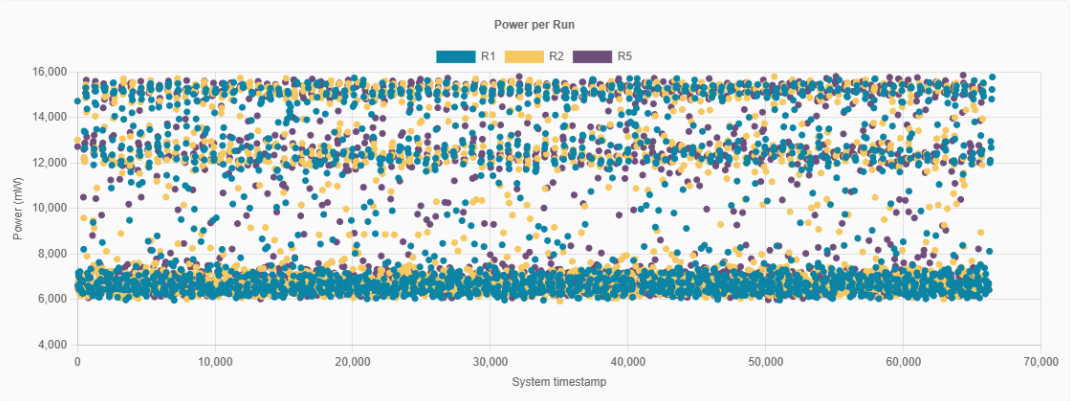


(c) Grpc.AspNetCore at P10

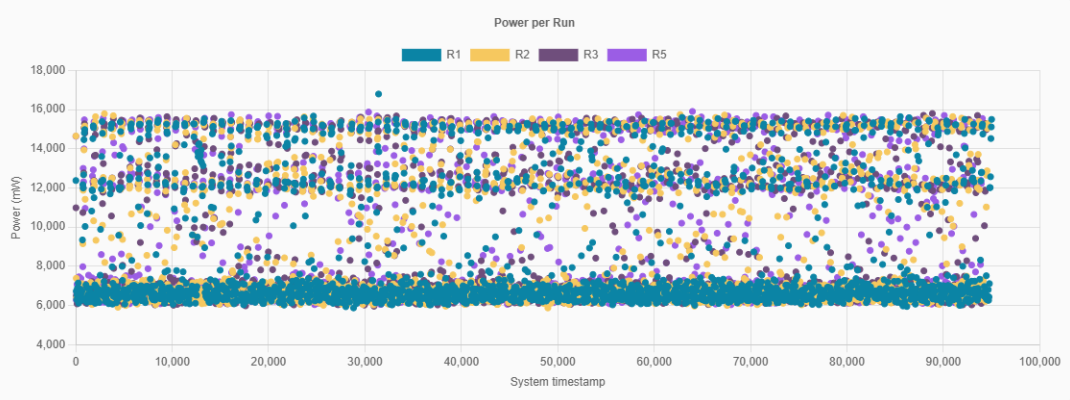
Figure 4.10: Per-run energy distributions at P10 for the three selected implementations. All three sessions share a similar run duration (approximately 0.56–0.64 s), so the plots are grouped together on a common timescale for direct comparison.

Figure 4.11 groups the three P320 scatter plots together, each covering the substantially longer run durations (approximately 45–94 s, Table 4.1 and Section 4.2)

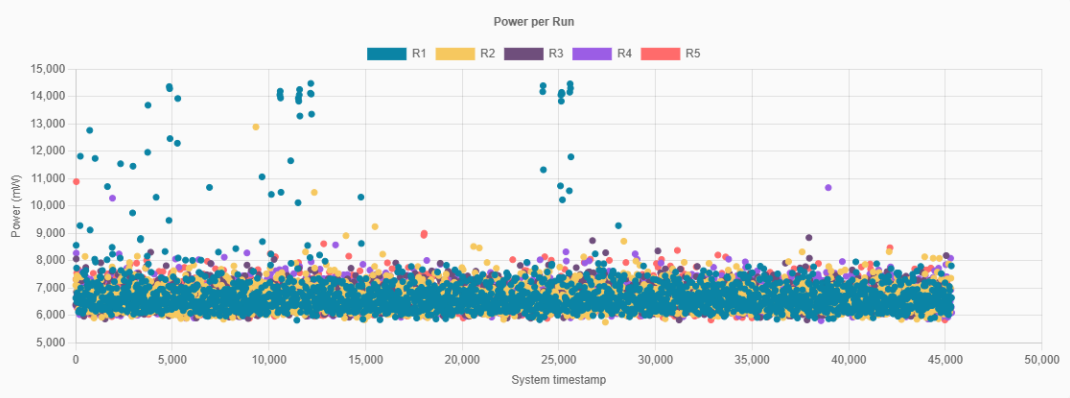
characteristic of the largest payload level. Grouping these on a shared timescale highlights both the much larger absolute energy values at this payload level and the differing run durations between implementations, with GraphQL.GraphQLNet's longer run duration visible as a wider horizontal spread compared to the other two implementations.



(a) Rest.Controllers at P320



(b) GraphQL.GraphQLNet at P320



(c) Grpc.AspNetCore at P320

Figure 4.11: Per-run energy distributions at P320 for the three selected implementations. All three sessions involve substantially longer run durations than at P10 (approximately 45–94 s), so the plots are grouped together on a common timescale for direct comparison.

The answer to RQ2 is therefore that payload size has a significant, architecture-

dependent, and direction-changing effect on energy consumption. At the smallest payload (P10), GraphQL.GraphQLNet is the most efficient of the three selected implementations and Rest.Controllers the least efficient. As payload size increases, this ordering inverts: by P40, Grpc.AspNetCore becomes the most efficient implementation across all three metrics, GraphQL.GraphQLNet becomes the least efficient, and Rest.Controllers occupies an intermediate position, an ordering that holds through P320, where GraphQL.GraphQLNet consumes 52–84% more energy than Grpc.AspNetCore depending on the metric. GraphQL's resolver-based execution model, negligible at small payloads, becomes the dominant cost driver at scale, while gRPC's combination of compact Protobuf serialisation and efficient client-side deserialisation yields an increasing advantage across all three metrics as payload size grows.

4.3 Effect of Concurrent Load on Energy Consumption (RQ3)

This section presents the results of the concurrency experiment conducted to address RQ3: *How does concurrent client load affect energy consumption across REST, GraphQL, and gRPC API architectures?* The same three representative implementations used in RQ2 were measured, Rest.Controllers, GraphQL.GraphQLNet, and Grpc.AspNetCore, with payload fixed at the baseline level (10 posts and 10 comments per post). Concurrent load was generated using k6 [46], with each measurement consisting of a single 30-second window during which N virtual users continuously sent requests to the backend. Eleven concurrency levels were tested: 5, 10, 15, 20, 25, 30, 100, 500, 1000, 2000, and 4000 virtual users. The results for backend CPU energy and network energy are presented in Tables 4.10 and 4.11 and illustrated in Figures 4.12 and 4.13.

Table 4.10: Backend CPU energy (odroid2/cpu) in joules across concurrency levels for RQ3. Each value represents a single 30-second measurement window.

VUs	**Rest.Controllers**	**GraphQL.GraphQLNet**	**Grpc.AspNetCore**
5	178.5	176.2	177.1
10	179.8	180.3	179.1
15	180.4	181.4	181.6
20	183.4	183.1	184.5
25	186.9	188.5	185.5
30	188.7	189.6	188.7
100	229.5	223.3	224.1
500	363.8	361.4	384.8
1000	502.3	511.4	464.4
2000	537.0	541.7	478.7
4000	594.4	618.0	511.2

Table 4.11: Network energy (odroid/network) in joules across concurrency levels for RQ3. Each value represents a single 30-second measurement window.

VUs	Rest.Controllers	GraphQL.GraphQLNet	Grpc.AspNetCore
5	214.6	213.9	215.1
10	214.0	214.0	215.2
15	214.4	214.1	215.8
20	214.3	214.8	215.8
25	215.0	215.3	216.0
30	215.0	215.5	216.4
100	220.2	219.2	219.0
500	223.3	221.6	226.6
1000	226.3	224.2	230.7
2000	228.0	226.4	235.9
4000	273.2	283.1	241.4

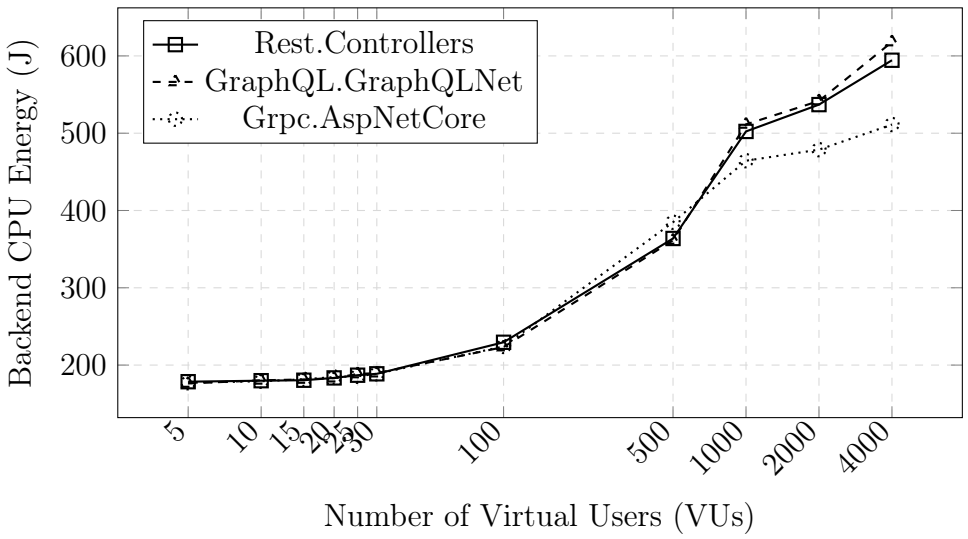


Figure 4.12: Backend CPU energy consumption (odroid2/cpu) across eleven concurrency levels. A logarithmic scale is used on the x-axis to accommodate the wide range of virtual user counts. All three architectures show stable energy consumption up to 30 VUs before increasing sharply beyond 100 VUs.

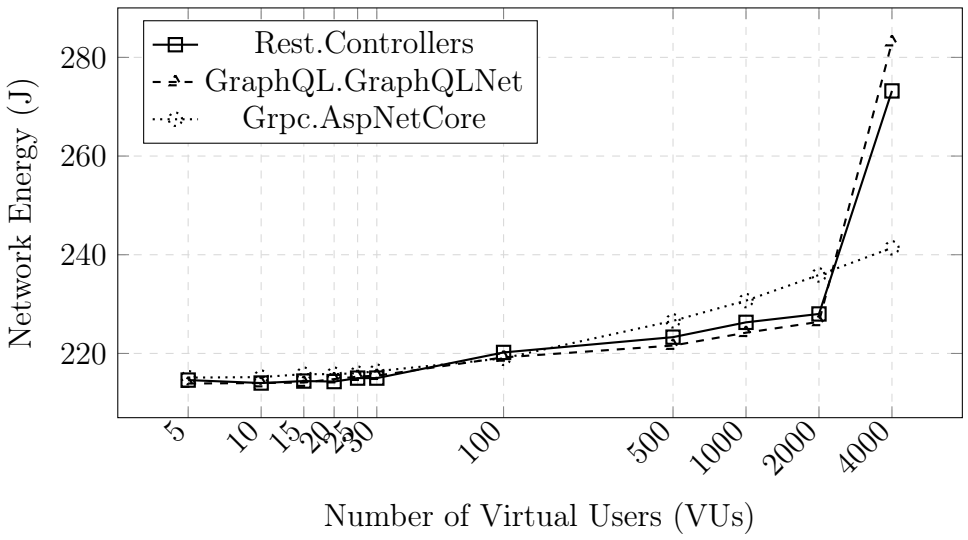


Figure 4.13: Network energy consumption (odroid/network) across eleven concurrency levels. Network energy remains remarkably stable up to 2000 VUs for all three architectures, with REST and GraphQL exhibiting a sharp increase at 4000 VUs while gRPC maintains a more gradual increase.

The results reveal three distinct phases of energy behaviour as concurrency increases, visible in both Figure 4.12 and Figure 4.13. These phase boundaries were identified through visual inspection of the concurrency scaling curves rather than through a formal change-point detection algorithm, consistent with the descriptive, visual-inspection-based analysis approach adopted throughout this study (Section 3.7).

Phase 1, Low concurrency (5 to 30 VUs): All three implementations exhibit remarkably stable and comparable energy consumption across this range. Backend CPU energy increases by less than 13 J between C5 and C30 for all three architectures, from approximately 177–179 J to 188–190 J. Network energy is similarly stable, remaining within a 3 J range across all three implementations. These results suggest that at low concurrency levels, the backend hardware is not saturated and handles the incremental load increase without significant additional energy cost. The differences between architectures are negligible in this phase, consistent with the RQ1 baseline findings.

Phase 2, Moderate concurrency (100 VUs): At 100 virtual users, all three implementations show a noticeable jump in backend CPU energy, to approximately 223–230 J, representing an increase of roughly 40 J compared to C30. This jump is consistent across all three architectures, suggesting that the transition from tens to hundreds of concurrent connections introduces a threshold effect, likely related to thread pool management, connection handling overhead, and increased context switching on the 4-core backend processor. Network energy remains largely stable at this level, increasing by only 3–5 J compared to C30.

Phase 3, High concurrency (500 to 4000 VUs): Beyond 100 VUs the three architectures begin to diverge meaningfully. At C500, all three show a large jump in backend CPU energy to approximately 360–385 J, with Grpc.AspNetCore slightly higher than REST and GraphQL at this specific level, a result that may reflect the overhead of establishing a large number of HTTP/2 connections simultaneously. It is also possible that the backend is approaching saturation at this concurrency level, with incoming requests being queued, or that messages are being split across multiple packets; distinguishing between these possible explanations would require packet-level network capture, which was not performed in this study, so this remains a tentative interpretation of the C500 anomaly rather than a confirmed cause. However, from C1000 onwards Grpc.AspNetCore demonstrates consistently lower backend CPU energy than the other two implementations, reaching 511.2 J at C4000 compared to 594.4 J for Rest.Controllers and 618.0 J for GraphQL.GraphQLNet. This advantage of approximately 83–107 J at C4000 is consistent with HTTP/2 multiplexing allowing gRPC to handle multiple concurrent requests more efficiently over persistent connections, reducing the per-connection overhead that accumulates under extreme concurrency for HTTP/1.1-based REST and GraphQL.

The network energy results in Figure 4.13 reveal an additional architectural distinction at extreme concurrency. Network energy remains remarkably flat across

all three implementations from C5 to C2000, increasing by less than 15 J over this entire range. However, at C4000, REST and GraphQL exhibit a sharp increase, to 273.2 J and 283.1 J respectively, while gRPC's network energy continues to increase gradually to 241.4 J. This divergence at extreme concurrency further supports the interpretation that gRPC's HTTP/2 transport manages high connection volumes more efficiently than the HTTP/1.1 used by REST and GraphQL, resulting in lower network-level energy under extreme load.

The absolute energy values in Table 4.10 combine two effects that increase together as concurrency rises: the energy cost of each individual request, and the number of requests completed within the fixed 30-second measurement window. Since the number of requests completed per window was not recorded separately from the energy measurement in this study, the values in Table 4.10 cannot be converted into an energy-per-request figure; however, Table 4.12 expresses backend CPU energy at each concurrency level relative to the C5 baseline, isolating how much energy consumption grows as a proportion of the lowest-concurrency condition, independent of the absolute joule values.

Table 4.12: Backend CPU energy at each concurrency level, expressed as a percentage increase relative to the C5 baseline, for RQ3.

VUs	**Rest.Controllers**	**GraphQL.GraphQLNet**	**Grpc.AspNetCore**
5	+0.0%	+0.0%	+0.0%
30	+5.7%	+7.6%	+6.5%
100	+28.6%	+26.7%	+26.5%
500	+103.8%	+105.1%	+117.3%
1000	+181.4%	+190.2%	+162.2%
2000	+200.8%	+207.4%	+170.3%
4000	+233.0%	+250.7%	+188.7%

Expressed relative to the C5 baseline, all three implementations follow a similar

growth pattern through Phase 1 and Phase 2, with energy consumption remaining within 30% of the baseline up to C100. The architectures diverge from C500 onwards, with Grpc.AspNetCore's relative increase overtaking REST and GraphQL at C500 before falling behind both at C1000 and beyond, reaching a relative increase of only +188.7% at C4000 compared to +233.0% for Rest.Controllers and +250.7% for GraphQL.GraphQLNet. This relative view reinforces the conclusion drawn from the absolute values, that Grpc.AspNetCore's energy consumption grows more slowly under extreme concurrent load than the other two implementations.

The answer to RQ3 is therefore that concurrent load has a substantial and architecture-dependent effect on energy consumption at high concurrency levels, while differences between architectures are negligible at low to moderate concurrency. Grpc.AspNetCore demonstrates the most favourable scaling behaviour under extreme concurrent load, consistent with the theoretical advantages of HTTP/2 multiplexing documented in prior performance studies [3], [23]. GraphQL.GraphQLNet scales least efficiently at high concurrency, consistent with the cumulative resolver execution overhead observed in RQ2.

4.4 Empirical Multi-Hop Network Topology Analysis (RQ4)

This section presents the results of the multi-hop network topology experiment conducted to address RQ4: *How can measured energy consumption be extended to estimate energy usage in distributed environments?* Rest.Controllers was measured at the P320 payload level across the three physical network topologies described in Section 3.5.4: UC1 (one switch hop), UC2 (two switch hops), and UC3 (one switch hop plus a wireless bridge hop). All values reported are net (idle-corrected) energy, calculated using the idle power baseline established in Section 3.3.1, with the

idle power for switch2, wifi1, and wifi2 cross-validated against UC1, in which these devices were connected but not part of the active request path.

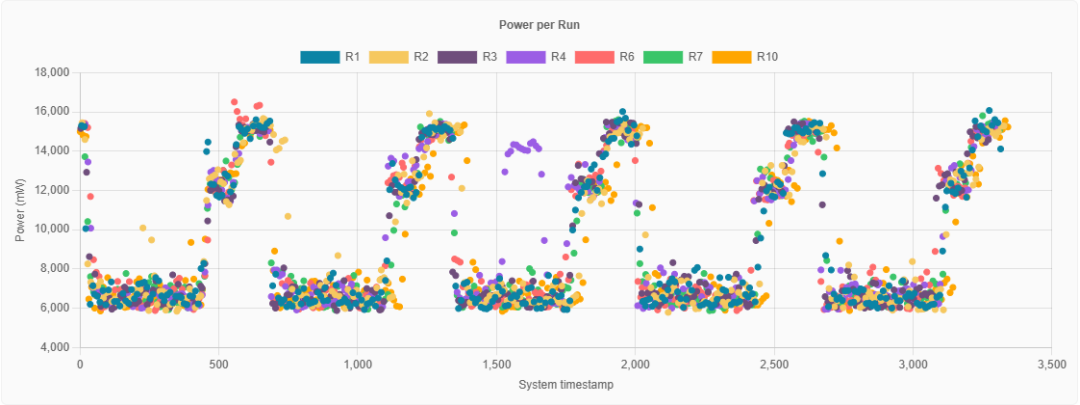
Table 4.13 presents the net energy consumption of each component across the three topologies, together with the total path energy, the sum of the net energy of all components in the active request path.

Table 4.13: Net (idle-corrected) energy consumption (J) per component across the three RQ4 network topologies, Rest.Controllers at P320.

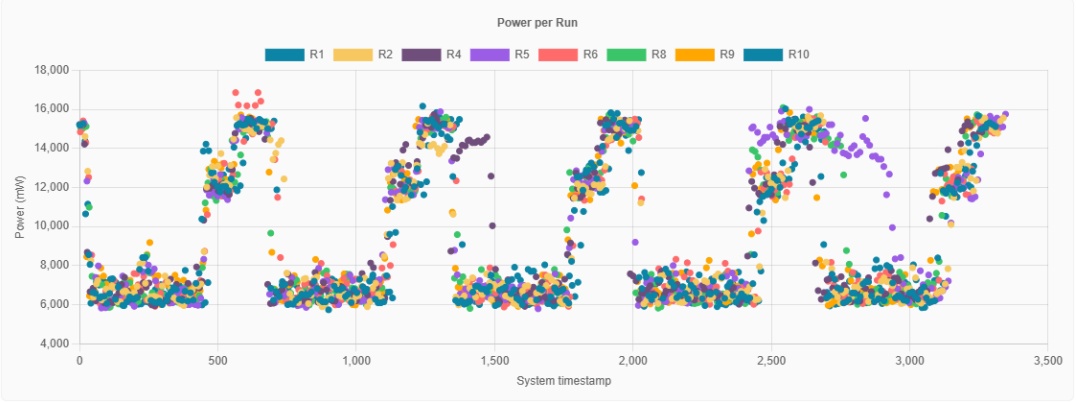
Component	**UC1 (J)**	**UC2 (J)**	**UC3 (J)**
odroid/cpu (frontend)	9.195	9.495	10.576
odroid2/cpu (backend)	23.350	23.492	23.964
switch1	0.132	−1.001	−1.385
switch2	≈ 0	1.444	2.386
wifi1	≈ 0	≈ 0	1.634
wifi2	≈ 0	≈ 0	4.282
Total path energy	**32.68**	**33.43**	**41.46**
Duration (s)	3.315	3.318	3.616

Expressing each component as a proportion of the total path energy clarifies how the energy budget shifts across topologies. In UC1, the backend accounts for approximately 71.5% of total path energy (23.350/32.68), the frontend approximately 28.1% (9.195/32.68), and switch1 approximately 0.4% (0.132/32.68). In UC2, the backend share is comparable at approximately 70.3%, while the combined network share (switch1 and switch2) rises to approximately 1.3%. In UC3, the backend share falls to approximately 57.8% and the frontend share to approximately 25.5%, while the combined network share (switch1, switch2, wifi1, and wifi2) rises to approximately 16.7%, reflecting the additional energy drawn by the wireless bridge hop. This proportional view reinforces the finding that the wireless bridge hop, rather than the wired switch hop, drives the increase in network share across the three

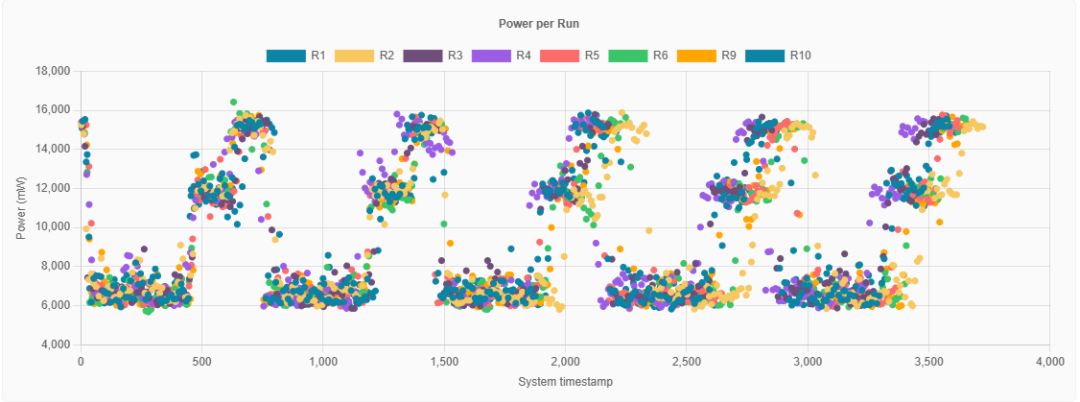
topologies.



(a) UC1 (odroid – switch1 – odroid2)



(b) UC2 (odroid – switch1 – switch2 – odroid2)



(c) UC3 (odroid – switch1 – WiFi bridge – WiFi bridge – switch2 – odroid2)

Figure 4.14: Per-run net energy distributions for the three RQ4 network topologies. All three sessions share a similar run duration (approximately 3.3–3.6 s), so the plots are grouped together on a common timescale for direct comparison.

Figure 4.14 presents scatter plots of the individual per-run net energy values

underlying Table 4.13, grouped together for UC1, UC2, and UC3 (subfigures a, b, and c). All three topologies share a similar run duration (approximately 3.3–3.6 s), allowing the per-run distributions to be compared directly on a common timescale. Each plot shows the distribution of net energy for the frontend, backend, and active network components of that topology, providing a visual indication of run-to-run variability alongside the mean values reported in the table.

The total path energy increases from 32.68 J (UC1) to 33.43 J (UC2) to 41.46 J (UC3). Decomposing these increases by hop type: adding a second wired switch hop (UC1 $\rightarrow$ UC2) increases total path energy by approximately 0.75 J, while additionally introducing a wireless bridge hop (UC2 $\rightarrow$ UC3) increases total path energy by a further 8.03 J, more than ten times the cost of the additional wired switch hop. The frontend and backend CPU energy also increase slightly with each additional hop (frontend: 9.195 $\rightarrow$ 9.495 $\rightarrow$ 10.576 J; backend: 23.350 $\rightarrow$ 23.492 $\rightarrow$ 23.964 J), and run duration increases correspondingly (from 3.315 s in UC1–UC2 to 3.616 s in UC3), reflecting the additional latency introduced by the wireless hop.

One anomaly is visible in the switch1 values: switch1's net energy is small and positive in UC1 (0.132 J) but becomes negative in UC2 (-1.001 J) and UC3 (-1.385 J), even though switch1 is the first active hop in all three topologies. This is interpreted as normal variation around switch1's idle baseline, given that switch1 carries substantial background traffic from the wider laboratory network (Section 3.5.4), its instantaneous power draw during a short measurement window may reasonably fluctuate by more than the small workload-attributable signal being measured, particularly since the workload-attributable energy at switch1 itself is expected to be small relative to switch2 and the WiFi bridge, which are dedicated to the experimental path.

4.4.1 Relation to the Additive Per-Hop Energy Model

Coroama et al. [32] proposed that energy consumption in distributed systems scales with the volume of data transmitted and that each intermediate network node contributes an additive energy cost to the total. The results of this experiment provide empirical support for the additive principle in general, total path energy increases monotonically with the number of hops (UC1 < UC2 < UC3), but reveal that the magnitude of this additive cost is strongly dependent on hop *type* rather than being uniform per hop, as a simple additive model would assume. An additional wired switch hop adds approximately 0.75 J, whereas an additional wireless bridge hop adds approximately 8.03 J under the conditions tested, a difference of roughly an order of magnitude.

This finding refines rather than contradicts the additive model: the total energy of a multi-hop path can still be expressed as the sum of per-hop contributions, but an analytical model intended for practical estimation should use hop-type-specific energy costs (e.g. distinguishing wired switch hops from wireless hops) rather than a single uniform per-hop value. This refined, empirically-grounded per-hop cost structure is used in Chapter 6 to revise the analytical energy estimation framework for distributed environments.

4.4.2 Worked Example: Estimating Larger Topologies

To illustrate how the empirically derived per-hop costs can be applied to estimate energy consumption in topologies beyond the three measured here, this section presents a simple linear extrapolation based on the UC1–UC3 measurements in Table 4.13.

UC1, with one wired switch hop, has a total path energy of 32.68 J. Treating this as the energy cost of the frontend, backend, and the first (unavoidable) switch hop combined, a base term can be defined as $E_{base} = 32.68$ J. Each *additional* wired switch hop beyond this first one then contributes approximately $E_{switch} = 0.75$ J

(the UC1 → UC2 difference), and each additional WiFi bridge pair contributes approximately $E_{wifi} = 8.03$ J (the UC2 → UC3 difference, i.e. the cost of introducing a wireless hop into an already two-switch path). The resulting linear estimate for a topology with n_{switch} additional wired switch hops and n_{wifi} additional WiFi bridge pairs is:

$$E_{total} \approx E_{base} + n_{switch} \cdot E_{switch} + n_{wifi} \cdot E_{wifi} \tag{4.1}$$

Table 4.14 applies Equation 4.1 to three illustrative topologies beyond those measured directly, alongside the two measured topologies (UC2 and UC3) for which the equation can be checked against the actual measured values.

Table 4.14: Linear extrapolation of total path energy for topologies beyond those measured, using Equation 4.1. UC2 and UC3 are included for comparison against their measured values from Table 4.13.

Topology	n_{switch}	n_{wifi}	**Estimated** E_{total} **(J)**	**Measured** E_{total} **(J)**
UC2 (2 switches)	1	0	33.43	33.43
UC3 (2 switches + WiFi bridge)	1	1	41.46	41.46
3 switches, no WiFi	2	0	34.18	–
4 switches, no WiFi	3	0	34.93	–
2 switches + 2 WiFi bridges	1	2	49.49	–

By construction, the estimates for UC2 and UC3 reproduce the measured values exactly, since Equation 4.1 is parameterised directly from these two measurements; this is not an independent validation of the model, but it confirms the equation is correctly applied. The remaining three rows extrapolate beyond the measured range and should be read as illustrative estimates rather than validated predictions.

This worked example highlights both the utility and the limitations of the current model. On one hand, it provides a simple, interpretable basis for reasoning about the energy cost of adding network infrastructure to a distributed deployment, for example, suggesting that adding two further wired switch hops to a path would increase total energy by less than 1 J, while adding a second WiFi bridge would increase it by a further 8 J. On the other hand, the model rests on only three measured data points, from which two slopes (E_{switch} and E_{wifi}) are derived directly rather than estimated through regression over multiple measurements. The model has not been tested for topologies with more than two wired switch hops or more than one WiFi bridge, and it assumes that each additional hop of a given type contributes a constant marginal cost regardless of its position in the path or the presence of other hop types, an assumption that has not been empirically verified. Establishing this model on a more rigorous footing, for example by measuring topologies with three or more switch hops, multiple WiFi bridges at varying distances, or different switch and access point hardware, and fitting the per-hop costs via regression across a larger number of measured topologies, is identified as a direction for future work in Chapter 6.

The answer to RQ4 is therefore that measured energy consumption can be extended to distributed environments using an additive per-hop model, provided that the per-hop energy cost is differentiated by hop type. Empirically, a wireless bridge hop in this setup costs approximately ten times as much additional energy as a wired switch hop, a distinction that is essential for any estimation framework aiming to reflect realistic heterogeneous network topologies rather than idealised uniform ones.

4.5 Statistical Evaluation and Discussion of Results

This section provides a consolidated statistical evaluation of the findings across the four research questions and discusses the key patterns that emerge from the

experimental data.

Measurement reliability and variability. For RQ1 and RQ2, each experimental condition was measured across 10 runs, enabling computation of mean and standard deviation values for raw energy as recorded by PowerGoblin (Section 3.3.1). The standard deviations observed are generally small relative to the mean values across all three metrics and payload levels, indicating that the measurements are stable and reproducible under the controlled laboratory conditions of this study. One exception is GraphQL.GraphQLNet's frontend CPU energy at P80 (65.57±11.81 J), where the standard deviation is notably larger relative to the mean than at other payload levels; this does not affect the overall ordering observed at P80 or the trend visible at neighbouring payload levels. For RQ4, an idle power baseline was additionally established (Section 3.3.1) to support the per-hop energy estimation in Section 4.4.1.

For RQ3, each concurrency level was measured in a single 30-second window, resulting in zero standard deviation in the reported values. This is a methodological limitation acknowledged in Section 3.5.3, the single-run approach for RQ3 means that measurement variability cannot be quantified, and the reported values should be interpreted as point estimates rather than statistical means. Despite this limitation, the consistency of the trends observed across all three architectures and across the wide range of concurrency levels tested provides confidence that the results reflect genuine architectural behaviour rather than measurement artefacts.

Cross-RQ patterns. Several consistent patterns emerge across the four research questions that together provide a coherent picture of the energy behaviour of the three architectural paradigms.

First, at the smallest workload tested (RQ1's P10 baseline), all three selected architectures are broadly comparable in absolute terms, with backend CPU energy differing by less than 1.4 J across all six implementations. However, the relative

ordering of the three architectures observed at P10, GraphQL.GraphQLNet most efficient, Rest.Controllers least efficient, is not representative of behaviour at larger workloads: RQ2 shows this ordering inverting by P40, and RQ3 shows negligible architectural differences only up to approximately C30. This finding has practical implications: for applications operating under light load with small response payloads, absolute energy differences between API architectures are small, but the relative efficiency ranking observed at this scale should not be extrapolated to higher workloads.

Second, architectural energy differences emerge, amplify, and, in the case of GraphQL and gRPC, invert direction as workload intensity increases. Both payload size (RQ2) and concurrency level (RQ3) act as amplifiers of the underlying architectural characteristics. In RQ2, the ordering observed at P10 has fully inverted by P40, after which Grpc.AspNetCore's advantage and GraphQL.GraphQLNet's disadvantage grow steadily through P320; in RQ3, a comparable inflection point appears between C30 and C100. Beyond these thresholds, the energy costs of GraphQL's resolver execution model, REST's JSON serialisation overhead, and gRPC's binary encoding efficiency become increasingly visible in the measured data.

Third, Grpc.AspNetCore demonstrates the most favourable energy efficiency under high workload conditions across both RQ2 and RQ3, and uniquely so across all three RQ2 metrics simultaneously at P320. Its backend CPU energy at P320 is approximately 17.2% lower than Rest.Controllers and approximately 45.7% lower than GraphQL.GraphQLNet; its frontend CPU energy is approximately 50.7% lower than Rest.Controllers and approximately 62.4% lower than GraphQL.GraphQLNet; and its network energy is approximately 32.1% lower than Rest.Controllers and approximately 52.5% lower than GraphQL.GraphQLNet. At C4000 (RQ3), its backend CPU energy is approximately 14% lower than Rest.Controllers and approximately 17% lower than GraphQL.GraphQLNet, a smaller margin than at P320, suggesting

that payload size is a stronger amplifier of gRPC's advantage than concurrency for this workload.

Fourth, GraphQL.GraphQLNet consistently exhibits the highest energy consumption under high workload conditions across both RQ2 and RQ3. This is consistent with the theoretical overhead of GraphQL's runtime query parsing, schema validation, and per-field resolver execution [3], [4], which impose a computational cost that scales with both the complexity and volume of the data being retrieved.

Fifth, the two implementations within each architectural paradigm, Rest.Controllers vs Rest.MinimalApi, GraphQL.GraphQLNet vs GraphQL.HotChocolate, and Grpc.AspNetCore vs Grpc.ProtobufNet, produce comparable RQ1 baseline results, with differences smaller than the measurement variability in most cases. This suggests that within-paradigm implementation choices have limited energy impact at the baseline level, contrary to the possibility, raised in the literature (Section 2.6), that framework-level differences within a paradigm might rival cross-architectural ones, and that the dominant energy determinants are the architectural and protocol-level characteristics shared by both implementations within each paradigm.

Sixth, the RQ4 results (Section 4.4) extend these architectural findings to the network topology dimension, showing that the energy cost of additional network infrastructure is not uniform but depends strongly on hop type. An additional wired switch hop adds a comparatively small amount of energy (approximately 0.75 J at P320), while an additional wireless bridge hop adds roughly ten times as much (approximately 8.03 J). This complements the RQ2 finding that serialisation format affects network energy primarily at large payload sizes: in distributed deployments, the choice and number of wireless hops along the request path may have a substantially larger impact on total network energy than the choice of API architecture itself.

Comparison with prior performance studies. The energy findings of this study are broadly consistent with the performance rankings reported in prior comparative studies [2], [3], [4], [25], which consistently find gRPC to outperform REST and GraphQL under high load and large payload conditions, and GraphQL to exhibit the highest server-side processing overhead. However, the relationship between performance and energy is not perfectly proportional, as noted by Capra et al. [15] and Pereira et al. [27], faster execution does not always imply proportionally lower energy consumption. The results of this study confirm this: at the baseline payload, GraphQL.GraphQLNet completes the workload fastest (0.575 s, Table 4.1) and also records the lowest backend CPU energy of the three selected implementations, yet this alignment between speed and energy does not persist as the payload grows, by P320, GraphQL.GraphQLNet becomes the least energy-efficient implementation despite processing the same data volume as REST and gRPC. This decoupling of relative speed from relative energy confirms that energy consumption cannot be reliably inferred from performance measurements alone.

5 Discussion

5.1 Interpretation of Findings

The empirical results presented in Chapter 4 collectively address the four research questions of this study and reveal a consistent and interpretable picture of how API architectural choices influence energy consumption under varying workload conditions.

The most significant finding is that architectural energy differences are workload-dependent, and that the relative *ranking* of architectures can invert as workload intensity increases. At the RQ1 baseline (P10), backend CPU energy differs by less than 1.4 J across all six implementations, with the two GraphQL implementations recording the *lowest* backend CPU energy and Rest.Controllers the *highest* (Section 4.1). Among the three implementations selected for RQ2, GraphQL.GraphQLNet is the most efficient across all three metrics (frontend, backend, and network energy) at P10, while Rest.Controllers is the least efficient and Grpc.AspNetCore occupies an intermediate position. As payload size increases, this ordering inverts: by P40, Grpc.AspNetCore becomes the most efficient implementation across all three metrics, GraphQL.GraphQLNet becomes the least efficient, and Rest.Controllers occupies the intermediate position, an ordering that holds, and intensifies, through P320. This indicates that GraphQL's resolver-based execution model carries comparatively little overhead at very small payloads, where the

number of fields to resolve is small, but that this overhead scales disproportionately with payload size, eventually dominating GraphQL's energy profile and outweighing the modest baseline advantage it held at P10. The architectural characteristics of each paradigm therefore do not produce a fixed ranking; rather, they interact with workload size in ways that favour different architectures at different scales.

The energy advantage of Grpc.AspNetCore at large payloads is broad-based, spanning all three measured metrics, and can be attributed to two complementary factors. First, GraphQL's resolver-based execution model is largely absent from the gRPC vs. REST comparison, so the backend CPU gap between Grpc.AspNetCore and Rest.Controllers at P320 (approximately 17.2%) is considerably smaller than the gap to GraphQL.GraphQLNet (approximately 45.7%), indicating that the presence or absence of per-field resolver execution is the dominant factor in backend CPU energy at scale. Second, Protocol Buffers binary serialisation produces substantially more compact payloads than the JSON used by REST and GraphQL (approximately 67% the size at every payload level, Table 3.3), which directly explains gRPC's larger advantage on the frontend and network metrics: at P320, Grpc.AspNetCore's frontend energy is approximately 50.7% lower than Rest.Controllers and 62.4% lower than GraphQL.GraphQLNet, and its network energy is approximately 32.1% and 52.5% lower respectively. Separately, HTTP/2 multiplexing allows gRPC to handle multiple concurrent requests over persistent connections, reducing the per-connection overhead that accumulates under high concurrency for HTTP/1.1-based REST and GraphQL [23]. The RQ3 results confirm this at extreme concurrency levels, where gRPC's backend CPU energy advantage over REST and GraphQL is approximately 14% and 17% respectively at 4000 virtual users, a smaller margin than at P320, suggesting payload size is a stronger amplifier of gRPC's advantage than concurrency for this workload.

GraphQL's consistently higher energy consumption under high workload condi-

tions is explained by its dynamic execution model. Each incoming query must be parsed, validated against the schema, and resolved field by field at runtime [3], [4]. As payload size increases, the number of resolver invocations grows proportionally with the number of posts and comments in the response, resulting in disproportionately higher CPU energy consumption. This overhead is architecture-specific, it is absent in REST, where a static endpoint handler processes the entire request in a single execution path, and in gRPC, where the service method directly maps to a strongly typed handler without dynamic field resolution.

The finding that REST occupies a middle position between gRPC and GraphQL in both RQ2 and RQ3 is consistent with its architectural characteristics. REST's JSON serialisation is less compact than Protobuf but avoids the dynamic resolver overhead of GraphQL, placing it between the two in terms of serialisation efficiency and server-side processing cost. The similarity between Rest.Controllers and Rest.MinimalApi in the RQ1 baseline suggests that the choice between controller-based and minimal API patterns has negligible energy impact, which is an important practical finding for developers choosing between these two ASP.NET Core patterns.

5.2 Comparison with Previous Research

The findings of this study can be contextualised within the broader body of comparative API research reviewed in Chapter 2. Prior performance-focused studies consistently report that gRPC outperforms REST and GraphQL under high load and large payload conditions in terms of latency and throughput [2], [3], [4], [25], and that GraphQL introduces higher server-side CPU utilisation due to its resolver execution model [3], [4]. The energy results of this study are broadly consistent with these performance rankings, providing empirical validation that the architectural factors driving performance differences also drive energy differences under equivalent conditions.

However, the relationship between performance and energy is not perfectly proportional, as demonstrated by the RQ1 baseline results. At small payload sizes, run duration varies only modestly across implementations (0.558–0.644 s), yet measurable energy differences exist in the backend CPU metric, and the implementations with the shortest run durations (the two GraphQL implementations) are also those with the lowest backend CPU energy, a relationship that does not hold at larger payload sizes, where GraphQL's backend CPU energy becomes the highest despite RQ2 showing GraphQL.GraphQLNet's frontend energy increasing alongside, rather than inversely with, its backend cost. This is consistent with the finding of Capra et al. [15] that execution time improvements do not always correspond to proportional reductions in energy consumption, and with Pereira et al. [27] who demonstrated that the relationship between speed and energy efficiency is context-dependent. The present study extends these insights to the specific context of API architecture comparison, confirming that energy measurement cannot be substituted by performance proxies, and that the relationship between speed and energy can itself vary with workload size.

The study most directly comparable to this work is Salonen [33], which measured the energy consumption of REST API implementations using the same PowerGoblin framework. Salonen's study focused on a single architectural paradigm and examined the effect of optimisation strategies such as caching and compression, finding measurable energy reductions from these techniques. The present study complements Salonen's work by extending hardware-based energy measurement to cross-architectural comparison and by investigating payload size and concurrency as workload factors, rather than optimisation strategies. While the absolute energy values are not directly comparable, Salonen's measurements aggregate substantially larger workloads per run and do not apply idle-power correction, both studies demonstrate that the PowerGoblin framework produces stable, low-variance measurements

suitable for reliable cross-condition comparison.

Herwig and Fischer [30] demonstrated that communication protocol choice produces measurable differences in hardware-level energy consumption on mobile devices, with REST and WebSocket exhibiting distinct energy profiles. The present study extends this finding to server-side contexts, confirming that protocol-level differences, specifically between HTTP/1.1 and HTTP/2, and between JSON and Protobuf serialisation, produce measurable energy differences that grow with workload intensity.

5.3 Practical Implications for API Design

The findings of this study have several practical implications for software architects and developers making API design decisions with energy efficiency as a consideration.

First, for applications operating under light load with small response payloads, such as low-traffic internal services, IoT device management APIs, or development and testing environments, the choice of API architecture is unlikely to have a meaningful impact on energy consumption. All three paradigms perform comparably under these conditions, and other factors such as developer productivity, ecosystem maturity, and tooling support should take precedence.

Second, for applications that regularly transfer large response payloads, such as data-intensive APIs serving complex nested data structures, reporting endpoints, or bulk data export services, gRPC offers a measurable and broad-based energy advantage over both REST and GraphQL that grows with payload size. At P320, Grpc.AspNetCore's backend CPU energy is approximately 17.2% lower than Rest.Controllers and approximately 45.7% lower than GraphQL.GraphQLNet, with comparable or larger advantages on the frontend and network metrics (Section 5.1). The substantially smaller gap between gRPC and REST (compared to

the gap between either and GraphQL) suggests that, for backend processing cost specifically, avoiding GraphQL's resolver overhead matters considerably more than the choice between JSON and Protobuf serialisation alone, though gRPC's compact serialisation provides an additional, compounding advantage on the frontend and network metrics. REST therefore offers a reasonable middle-ground alternative where gRPC's browser compatibility constraints [24] make it unsuitable for direct client-facing deployment, while gRPC remains the most energy-efficient choice overall for large-payload, backend-to-backend communication.

Third, for applications subject to high concurrent load, such as public-facing APIs serving many simultaneous users during peak traffic periods, gRPC's HTTP/2 multiplexing provides an energy efficiency advantage that becomes increasingly significant at extreme concurrency levels. However, this advantage only manifests beyond approximately 100 concurrent connections; below this threshold, all three architectures are comparable.

Fourth, GraphQL's energy efficiency disadvantage under high workload conditions does not necessarily disqualify it as an architectural choice. GraphQL's ability to retrieve precisely the data required by each client, eliminating over-fetching, may result in lower total system energy in scenarios where clients have highly variable data requirements and would otherwise receive unnecessarily large REST responses. The present study used a fixed query that retrieved all available fields, which represents a worst-case scenario for GraphQL's resolver overhead. In real-world deployments where clients leverage GraphQL's selective fetching capability, the effective payload size and corresponding energy cost may be substantially lower than the fixed-query results suggest.

5.4 Applicability of the Results to Real-World Use Cases

The empirical findings of this study, summarised in Section 6.1 and detailed in Chapter 4, are workload-dependent: the relative energy efficiency of REST, GraphQL, and gRPC changes with payload size and concurrent load rather than following a fixed ranking. This section briefly connects these workload-dependent results to concrete application domains, to clarify where the findings of this study are most directly applicable and where their relevance is limited.

Low-traffic and lightweight services. Internal tooling, administrative dashboards, IoT device management endpoints, and development or staging environments typically exchange small payloads, comparable to the P10 condition tested in this study (10 posts and 10 comments, 8.7–12.9 KB depending on implementation, Table 3.3), under light, often single-client load. For these use cases, RQ1 and the low-concurrency phase of RQ3 (up to approximately 30 virtual users, Section 4.3) both show that energy differences between REST, GraphQL, and gRPC are small and, for most pairwise comparisons, not statistically distinguishable (Section 4.1). The architectural choice for this class of application can therefore be made primarily on grounds other than energy efficiency, such as development speed, team familiarity, or ecosystem tooling, without a meaningful energy efficiency trade-off.

Data-intensive APIs and bulk data services. Reporting endpoints, data export services, content management backends, and APIs serving large nested resources (for example, a social media feed with many posts and comments, or a product catalogue with detailed nested attributes) regularly exchange payloads at the scale of the P160 and P320 conditions tested in this study (3–12 MB depending on implementation, Table 3.3). RQ2 shows that this is precisely the regime in which architectural energy differences become large and statistically significant

(Section 4.2): at P320, GraphQL.GraphQLNet consumes 52–84% more energy than Grpc.AspNetCore depending on the metric, while Rest.Controllers occupies an intermediate position. For backend services in this category, particularly internal or backend-to-backend communication where gRPC's browser compatibility constraints [24] do not apply, the results of this study indicate a measurable energy efficiency advantage in adopting gRPC with Protobuf serialisation over REST or GraphQL with JSON.

High-concurrency, public-facing APIs. Public web APIs and backend services that serve many simultaneous users, such as e-commerce platforms during peak shopping periods, ticketing systems during on-sale events, or social media APIs during viral traffic spikes, routinely operate at concurrency levels comparable to the higher end of the range tested in RQ3 (1000–4000 virtual users, Section 4.3). At these concurrency levels, gRPC's HTTP/2 multiplexing provides a measurable energy advantage over the HTTP/1.1-based REST and GraphQL implementations tested, though this advantage (approximately 14–17% at 4000 virtual users) is smaller in relative terms than the payload-driven advantage observed in RQ2. For public-facing endpoints where gRPC cannot be used directly due to browser compatibility, REST remains the more practical choice at high concurrency, as it tracks more closely with gRPC's energy profile than GraphQL does under sustained concurrent load (Section 4.3).

Client-driven, variable-shape data access. Mobile and single-page applications that require precise, client-specified data shapes, where the alternative is multiple REST round trips or substantially over-fetched responses, represent a use case in which GraphQL's architectural strengths are not directly captured by this study's measurements. As discussed in Section 5.3, the fixed, full-field query used throughout this study represents a worst-case scenario for GraphQL's resolver overhead; in deployments where GraphQL's selective field querying meaningfully reduces

the effective payload size compared to an equivalent REST response, the energy cost of GraphQL's resolver execution may be offset, partially or fully, by the avoided cost of transmitting and processing unused data. The results of this study therefore most directly inform the case where GraphQL is queried for a fixed, complete data shape; they do not directly quantify the energy trade-off of GraphQL's selective fetching, which remains a direction for future work (Section 6.4).

Across all four of these domains, the results of this study should be applied with the limitations discussed in Section 5.5 in mind, particularly the use of low-power ODROID hardware rather than server-grade infrastructure and the focus on a single, read-heavy workload scenario. The relative architectural patterns identified here, payload size and concurrency as amplifiers of architectural energy differences, gRPC's growing advantage at scale, and GraphQL's resolver-driven overhead under large or complex queries, are expected to generalise qualitatively across deployment environments, even where the absolute energy values reported in this thesis do not.

5.5 Limitations of the Study

Several limitations of this study should be acknowledged when interpreting the findings.

The hardware used in this study, ODROID single-board computers with low-power Intel processors, is not representative of server-grade hardware used in production cloud deployments. The absolute energy values reported are specific to this hardware configuration and should not be directly extrapolated to cloud or data centre environments. The relative differences between architectures and the scaling trends are expected to reflect genuine architectural characteristics, but their magnitude may differ on different hardware.

The study uses a single workload scenario, a nested user-post-comment retrieval, which represents one class of read-heavy API interaction. Write-heavy operations,

streaming responses, or highly selective GraphQL queries may produce different energy profiles and different relative rankings between architectures.

More broadly, the measurement chain used in this study is subject to several identifiable sources of error, including the 3% tolerance of the SmartPower 3 meters, minor idle-baseline drift, background laboratory network traffic affecting switch1, and software-level transient effects such as JIT compilation, garbage collection, and OS scheduling jitter; these are catalogued, together with an assessment of the resulting reliability of the RQ1, RQ2, RQ3, and RQ4 results, in Section 3.6.1. None of these sources is expected to bias comparisons between implementations differentially, but they do set a practical floor on the smallest energy difference that can be considered meaningful.

The RQ3 concurrency levels were generated by k6 spawning multiple virtual users on a single physical client machine (odroid), rather than by distributing requests across multiple independent client systems. While this approach is consistent with common load testing practice and allowed concurrency to be varied systematically within the available hardware, a single client machine sharing CPU, memory, and network interface resources across all virtual users does not behave identically to genuinely independent client machines issuing requests concurrently; in particular, contention for these shared client-side resources at high virtual user counts may itself contribute to the energy and performance effects observed at extreme concurrency levels, rather than these effects being attributable solely to the backend's handling of concurrent connections. Future work using a distributed load generation setup with multiple physical or virtual client machines would help isolate client-side contention effects from genuine backend concurrency-handling behaviour.

A related sample-size limitation applies to the statistical testing introduced in Section 3.7.1. While most RQ1 and RQ2 conditions are measured across the full 10 runs, the RQ2 P320 condition for all three selected implementations is measured

across only 5 runs rather than 10 (Section 3.5.2), owing to the substantially longer duration of each run at this payload level. The Kruskal-Wallis and Dunn's post-hoc results reported for P320 (Section 4.2) are therefore based on a smaller sample than the other five payload levels, which reduces statistical power and increases the risk that a true difference between implementations fails to reach significance at this specific payload level, as illustrated by Rest.Controllers not being statistically distinguishable from either GraphQL.GraphQLNet or Grpc.AspNetCore at P320 despite a similar pattern of separation being significant at P160. The P320 results should accordingly be interpreted with this reduced sample size in mind; the underlying mean values remain consistent with the trend observed at the other five payload levels, but the corresponding significance tests carry comparatively less statistical power.

The three implementations selected for RQ2 and RQ3 represent one implementation choice per architectural paradigm. The RQ1 results suggest that within-paradigm implementation differences are small at baseline conditions, but this may not hold under high workload conditions where framework-level differences in connection pooling, thread management, and serialisation optimisation could become significant.

For RQ1 and RQ2, energy values are reported as raw totals recorded by PowerGoblin, without idle power correction (Section 3.3.1). The idle power contribution of the measurement hardware is constant across implementations measured within a session and therefore does not affect the validity of relative comparisons, which form the basis of the RQ1 and RQ2 analyses. However, this means that the absolute values reported for RQ1 and RQ2 include background system energy in addition to the energy attributable to the workload itself, and should not be interpreted as the energy cost of a single API request in isolation. For RQ4, an idle power baseline was established and subtracted (Section 3.3.1), as isolating the network-attributable

energy of each hop is necessary for the additive per-hop estimation model in Section 4.4.1; consequently, RQ4's values are not directly comparable in magnitude to the raw values reported for RQ1–RQ3. For RQ3, the concurrency measurements use 30-second windows with continuously varying numbers of concurrent connections, for which a single steady-state idle power value would not meaningfully represent the "idle" condition of the system during the measurement window; RQ3's values are therefore also reported as raw totals.

The RQ4 multi-hop topology experiment used a single implementation (Rest.Controllers) and a single payload level (P320), measured across three topologies. While this provides a useful first empirical estimate of per-hop energy costs for wired switch and wireless bridge hops, it does not establish whether these per-hop costs are independent of the API architecture or payload size in use; the per-hop costs derived in Section 4.4.1 should be treated as estimates specific to the tested configuration rather than universal constants.

All six implementations in this study were developed using a single programming language and runtime, .NET 8. Implementations of the same architectural paradigm in a different language or runtime could plausibly exhibit different absolute and relative energy profiles, since garbage collection behaviour, just-in-time compilation, and serialisation library implementations vary across runtimes. The architectural and protocol-level differences identified in this study are expected to generalise across languages, but the magnitude of these differences, and potentially their direction at extreme workload levels, has not been verified outside the .NET ecosystem.

6 Conclusion and Future Work

6.1 Summary of Key Findings

This study conducted a controlled empirical comparison of the energy consumption of six .NET 8 API implementations across three architectural paradigms, REST, GraphQL, and gRPC, using hardware-based power measurement through the PowerGoblin framework [7]. Measurements were performed across four experimental dimensions: a baseline comparison of all six implementations under identical conditions (RQ1), a payload scaling experiment across six payload levels for three representative implementations (RQ2), a concurrency scaling experiment across eleven concurrency levels for the same three implementations (RQ3), and an empirical multi-hop network topology experiment across three physical network configurations (RQ4).

The key findings can be summarised as follows. Under baseline conditions with small payloads and sequential single-client requests, all six implementations consume comparable energy, with backend CPU energy differences of less than 1.4 J. Among the three implementations selected for RQ2, GraphQL.GraphQLNet is the most efficient across all three measured metrics (frontend, backend, and network energy) at this baseline payload, while Rest.Controllers is the least efficient and Grpc.AspNetCore occupies an intermediate position, an ordering that inverts completely as payload size increases (RQ2).

As payload size increases, architectural energy differences emerge, amplify, and invert relative to the baseline ordering. By P40, the ordering observed at P10 has fully inverted across all three metrics: Grpc.AspNetCore becomes the most efficient implementation, GraphQL.GraphQLNet the least efficient, and Rest.Controllers occupies the intermediate position, an ordering that holds, and intensifies, through P320. At the largest tested payload of 320 posts and comments, GraphQL.GraphQLNet's backend CPU energy is approximately 84.2% higher than Grpc.AspNetCore's and approximately 52.5% higher than Rest.Controllers', while Rest.Controllers' backend CPU energy is itself approximately 20.8% higher than Grpc.AspNetCore's. GraphQL's reversal, from the most energy-efficient implementation at baseline to the least efficient at scale, reflects the cumulative cost of its per-field resolver execution model, which is negligible for small responses but scales disproportionately with the number of fields resolved.

Under concurrent load, all three architectures perform comparably up to approximately 30 virtual users, after which energy consumption increases with concurrency for all implementations. At extreme concurrency levels of 1000 VUs and above, Grpc.AspNetCore demonstrates consistently lower backend CPU energy than REST and GraphQL, with a difference of approximately 83–107 J at 4000 VUs. This advantage is consistent with HTTP/2 multiplexing allowing gRPC to manage high numbers of concurrent connections more efficiently than the HTTP/1.1 used by REST and GraphQL.

Finally, the empirical multi-hop network topology experiment (RQ4) shows that total path energy increases monotonically with the number of intermediate network hops, but that the magnitude of this increase depends strongly on hop type: an additional wired switch hop adds approximately 0.75 J, while an additional wireless bridge hop adds approximately 8.03 J, roughly ten times more. This supports the additive per-hop principle proposed in prior network-level energy modelling work

[32], while showing that hop type, not merely hop count, is essential for accurate energy estimation in distributed environments.

6.2 Answers to Research Questions

RQ1: How do REST, GraphQL, and gRPC architectures differ in baseline energy consumption? Under baseline conditions, the six implementations are broadly comparable in energy consumption, though small but consistent differences exist across all three measured metrics. The two GraphQL implementations record the lowest backend CPU energy, with GraphQL.HotChocolate at 5.995 ± 0.714 J per run, while Rest.Controllers records the highest at 7.338 ± 0.471 J. A consistent pattern is visible across all three metrics: among the three implementations selected for RQ2, GraphQL.GraphQLNet is the most efficient and Rest.Controllers the least efficient at this payload level. No architectural paradigm demonstrates a clear advantage in absolute terms at this small payload, and, as RQ2 shows, this baseline ordering does not predict relative behaviour at larger payload sizes; it in fact inverts completely by P40.

RQ2: How does payload size affect energy consumption across REST, GraphQL, and gRPC API architectures? Payload size has a significant, architecture-dependent, and direction-changing effect on energy consumption. At the smallest payload (P10), GraphQL.GraphQLNet is the most efficient of the three selected implementations and Rest.Controllers the least efficient. By P40, this ordering has fully inverted across all three metrics: Grpc.AspNetCore becomes the most efficient, GraphQL.GraphQLNet the least efficient, and Rest.Controllers occupies an intermediate position. At the maximum tested payload (320 posts and comments), GraphQL.GraphQLNet's backend CPU energy is approximately 84.2% higher than Grpc.AspNetCore's and approximately 52.5% higher than Rest.Controllers', while Rest.Controllers is itself approximately 20.8% higher than

Grpc.AspNetCore. Grpc.AspNetCore's advantage is broad-based, extending to frontend CPU energy (50.7%/62.4% lower than REST/GraphQL) and network energy (32.1%/52.5% lower) at P320, indicating that gRPC's combination of compact Protobuf serialisation and the absence of resolver overhead makes it the most energy-efficient architecture at scale across all three metrics.

RQ3: How does concurrent client load affect energy consumption across REST, GraphQL, and gRPC API architectures? Concurrent load has negligible energy impact up to approximately 30 virtual users. Beyond this threshold, energy increases with concurrency for all three architectures. At extreme concurrency levels, gRPC demonstrates a meaningful energy advantage over REST and GraphQL, consistent with the efficiency of HTTP/2 connection multiplexing under high concurrent load. GraphQL exhibits the highest energy consumption at extreme concurrency, consistent with the cumulative overhead of its resolver execution model.

RQ4: How can measured energy consumption be extended to estimate energy usage in distributed environments? RQ4 is addressed empirically in Section 4.4, through measurements of Rest.Controllers at P320 across three physical network topologies with increasing numbers of intermediate hops. Total path energy increases monotonically with the number of hops, supporting the additive per-hop principle proposed by Coroama et al. [32]. However, the per-hop energy cost is strongly dependent on hop type: an additional wired switch hop adds approximately 0.75 J, while an additional wireless bridge hop adds approximately 8.03 J, roughly ten times more. Measured energy consumption can therefore be extended to distributed environments using an additive model, provided that per-hop costs are differentiated by hop type rather than treated as uniform (Section 4.4.1). A worked example applying this model to topologies beyond those measured is presented in Section 4.4.2.

6.3 Contributions

This study makes four primary contributions to the field of sustainable software engineering and energy-aware API design.

The first contribution is a direct, hardware-based cross-architectural energy comparison of REST, GraphQL, and gRPC under equivalent and systematically varied workload conditions. No prior study has conducted such a comparison using hardware power meters across all three paradigms simultaneously, addressing a gap explicitly identified in the literature [30], [33].

The second contribution is the empirical quantification of how payload size affects the relative energy efficiency of the three architectures. The finding that the relative efficiency ordering observed at the smallest payload inverts completely by P40, and that this inversion intensifies through P320, with GraphQL.GraphQLNet's energy consumption growing 52–84% higher than the other implementations depending on the metric, provides actionable guidance for API design decisions in data-intensive applications: efficiency rankings observed at small payloads should not be assumed to hold as payload size grows.

The third contribution is the empirical characterisation of how concurrent load affects the energy consumption of REST, GraphQL, and gRPC. The finding that gRPC's energy advantage under high concurrency only emerges beyond approximately 100 concurrent connections provides a practical threshold for architects evaluating the energy implications of protocol choice in high-traffic deployments.

The fourth contribution is an empirical, hardware-measured characterisation of how additional network infrastructure affects total energy consumption in a multi-hop topology, refining the additive per-hop energy model of Coroama et al. [32] with hop-type-specific empirical costs, distinguishing wired switch hops from wireless bridge hops. This provides a concrete, measurement-grounded basis for estimating distributed system energy consumption, addressing a gap identified in Chapter

2 where existing network-level energy models were not grounded in API-specific empirical measurements.

6.4 Future Research Directions

Several directions for future research emerge from the findings and limitations of this study.

Generalising the empirical per-hop energy model (RQ4). The RQ4 results (Section 4.4) establish hop-type-specific empirical energy costs for a single implementation (Rest.Controllers) at a single payload level (P320): approximately 0.75 J for an additional wired switch hop and approximately 8.03 J for an additional wireless bridge hop. The additive energy estimation model can be refined to incorporate these hop-type-specific costs:

$$E_{total} = r \cdot (E_{client} + n_{switch} \cdot E_{switch} + n_{wifi} \cdot E_{wifi} + E_{backend}) \tag{6.1}$$

where n_{switch} and n_{wifi} denote the number of additional wired switch hops and wireless bridge hops respectively, and E_{switch} and E_{wifi} are the corresponding empirically derived per-hop costs from Table 4.13. The worked example in Section 4.4.2 applies this model to several illustrative topologies beyond UC1–UC3, demonstrating its use while also highlighting that it is parameterised from only three measured data points, with two per-hop slopes derived directly rather than estimated via regression. Future work should validate whether these per-hop costs generalise across other implementations, payload sizes, and physical network hardware, including different switch models, WiFi standards, and bridge distances, and should measure additional topologies (three or more wired switch hops, multiple WiFi bridges) so that the per-hop costs can be estimated via regression rather than read directly from a minimal set of measurements, in order to establish Equation 6.1 as a reliable

estimation tool for heterogeneous distributed topologies.

Finer-grained energy decomposition. The energy model used in this study, and the worked example in Section 4.4.2, operate at the granularity of board-level energy for the frontend and backend nodes (`odroid/cpu` and `odroid2/cpu`) and per-device energy for intermediate network components. A more granular model could decompose the client- and server-side energy terms further, for example into application-level, runtime (.NET runtime), operating system kernel, network stack, and network interface controller (NIC) contributions. Such a decomposition would require process- or component-level energy attribution tools, such as RAPL-based per-process profiling or eBPF-based energy measurement, that go beyond the board-level SmartPower 3 measurements used in this study. Pursuing this direction would allow future work to attribute the energy differences observed between architectures more precisely, for example distinguishing how much of GraphQL.GraphQLNet's elevated backend energy at large payloads is attributable to application-level resolver execution versus runtime serialisation versus kernel-level I/O handling.

Optimisation strategies. Future work should investigate the energy impact of configuration-based optimisation techniques applied to the tested architectures, including response compression, HTTP/2 for REST, connection pooling, and caching. Zafari et al. [34] demonstrated that jointly optimising communication, caching, and compression can produce substantial energy reductions, and Salonen [33] showed that compression and caching reduce REST API energy consumption measurably. Extending these findings to cross-architectural comparison would provide a more complete picture of the energy optimisation potential available to each paradigm.

Extended hardware platforms. Replicating this study on server-grade hardware, including multi-core processors, larger memory configurations, and cloud virtual machine instances, would establish whether the relative energy rankings observed on the ODROID hardware generalise to production deployment environ-

ments. The absolute energy values would differ, but the relative architectural differences are expected to persist.

Write-heavy and streaming workloads. This study focused exclusively on read-heavy nested data retrieval. Future work should investigate write operations, mutations, and streaming responses, including gRPC server streaming and GraphQL subscriptions, which may produce different relative energy profiles between architectures.

Repeated concurrency measurements. The single-run approach used for RQ3 precludes statistical analysis of measurement variability at each concurrency level. Future work should repeat concurrency measurements across multiple sessions to establish confidence intervals and enable more rigorous statistical comparison between architectures under concurrent load.

CO_2 equivalent estimation. The energy measurements from this study can be connected to environmental impact estimates through CO_2 equivalent conversion factors that account for national energy mixes, data centre power usage effectiveness (PUE), and infrastructure scaling factors. Such estimates would bridge the gap between laboratory energy measurements and the broader sustainability impact of API architectural choices at scale, connecting the findings of this study to the Green ICT objectives that motivate it [1], [10].

References

[1] A. S. G. Andrae and T. Edler, "On global electricity usage of communication technology: Trends to 2030", *Challenges*, vol. 6, no. 1, pp. 117–157, 2015. DOI: 10.3390/challe6010117.

[2] M. Śliwa and B. Pańczyk, "Performance comparison of programming interfaces on the example of REST API, GraphQL and gRPC", *Journal of Computer Sciences Institute*, vol. 21, pp. 356–361, 2021.

[3] S. Chandra and A. Farisi, "Comparative analysis of RESTful, GraphQL, and gRPC APIs: Performance insight from load and stress testing", *Jurnal Sisfokom (Sistem Informasi dan Komputer)*, vol. 14, no. 1, pp. 81–85, Jan. 2025.

[4] M. Niswar, R. A. Safruddin, A. Bustamin, and I. Aswad, "Performance evaluation of microservices communication with REST, GraphQL, and gRPC", *International Journal of Electronics and Telecommunications*, vol. 70, no. 2, pp. 429–436, 2024.

[5] S. Naumann, M. Hirsch-Dick, E. Kern, and T. Johann, "The GREENSOFT model: A reference model for green and sustainable software and its engineering", *Sustainable Computing: Informatics and Systems*, vol. 1, no. 4, pp. 294–304, 2011.

[6] E. A. Jagroep, "Green software products", ISBN: 978-90-393-6840-4, Ph.D. dissertation, Utrecht University, Utrecht, Netherlands, 2017.

[7] PowerGoblin, *PowerGoblin — hardware-based energy measurement framework*, https://visiiri.tt.utu.fi/powergoblin/, University of Turku, Visiiri Project. Accessed: Jun. 2026.

[8] E. O'Dea, T. Vogel, and P. Lago, "A measurement-driven approach to enhancing sustainability in microservice architectures", in *Proc. IEEE 22nd Int. Conf. Softw. Archit. Companion (ICSA-C)*, 2025. DOI: 10.1109/ICSA-C63560.2025.11014885.

[9] E. Bajrami, "Assessing the role of software in sustainability: A survey of industry practices and research trends", *Sakarya University Journal of Computer and Information Sciences*, vol. 8, no. 2, pp. 273–285, Jun. 2025. DOI: 10.35377/saucis.1589506.

[10] L. M. Hilty and B. Aebischer, Eds., *ICT Innovations for Sustainability* (Advances in Intelligent Systems and Computing). Cham, Switzerland: Springer, 2015, vol. 310.

[11] E. Masanet, A. Shehabi, N. Lei, S. Smith, and J. Koomey, "Recalibrating global data center energy-use estimates", *Science*, vol. 367, no. 6481, pp. 984–986, Feb. 2020. DOI: 10.1126/science.aba3758.

[12] P. Lago, S. A. Koçak, I. Crnkovic, and B. Penzenstadler, "Framing sustainability as a software quality attribute", *Communications of the ACM*, vol. 58, no. 10, pp. 70–78, 2015.

[13] A. Hindle, "Green software engineering: The curse of methodology", in *Proc. 23rd IEEE Int. Conf. Software Analysis, Evolution, and Reengineering (SANER)*, Osaka, Japan, Mar. 2016, pp. 529–540. DOI: 10.1109/SANER.2016.60.

[14] G. Pinto, F. Castor, and Y. D. Liu, "Mining questions about software energy consumption", in *Proc. 11th Working Conf. Mining Software Reposito-*

ries (MSR), Hyderabad, India, May 2014, pp. 22–31. DOI: 10.1145/2597073.2597110.

[15] E. Capra, C. Francalanci, and S. A. Slaughter, "Measuring application software energy efficiency", *IT Professional*, vol. 14, no. 2, pp. 54–61, 2012.

[16] Google LLC, *Protocol buffers: Developer guide*, https://protobuf.dev/overview/, Accessed Mar. 10, 2026.

[17] R. T. Fielding, "Architectural styles and the design of network-based software architectures", Ph.D. dissertation, University of California, Irvine, CA, USA, 2000.

[18] I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen, *Microservice Architecture: Aligning Principles, Practices, and Culture*. Sebastopol, CA, USA: O'Reilly Media, 2016.

[19] GraphQL Foundation, *GraphQL specification*, https://spec.graphql.org/October2021/, Oct. 2021.

[20] G. Brito and M. T. Valente, "REST vs GraphQL: A controlled experiment", in *Proc. IEEE Int. Conf. Software Architecture (ICSA)*, Salvador, Brazil: IEEE, Mar. 2020, pp. 81–91. DOI: 10.1109/ICSA47634.2020.00016.

[21] Google, *gRPC: A high performance, open source universal RPC framework*, https://grpc.io/, Accessed Mar. 10, 2026.

[22] P. K. Kumar, R. Agarwal, R. Shivaprasad, D. Tiwari, and S. Kalambur, "Performance characterization of communication protocols in microservice applications", in *Proc. IEEE Int. Conf. Smart Applications, Communications and Networking (SmartNets)*, 2021. DOI: 10.1109/SmartNets52353.2021.9555434.

[23] N. Oda and S. Yamaguchi, "HTTP/2 performance evaluation with latency and packet losses", in *Proc. 15th IEEE Annual Consumer Communications & Networking Conference (CCNC)*, Las Vegas, NV, USA, Jan. 2018. DOI: 10.1109/CCNC.2018.8319285.

[24] J. Brandhorst, *The state of gRPC in the browser*, https://grpc.io/blog/state-of-grpc-web/, Jan. 2019.

[25] S. Reichersdörfer, "A comparative study of REST, GraphQL, and gRPC for API development: Practical benchmarking with a Trello-like task management system", B.S. thesis, Jamk University of Applied Sciences, Jyväskylä, Finland, Dec. 2025.

[26] J. Johansson and E. Isabella, "A comparative study of REST and gRPC for microservice communication", B.S. thesis, Dept. of Computer Science, Linköping University, Linköping, Sweden, 2022.

[27] R. Pereira et al., "Energy efficiency across programming languages: How do energy, time, and memory relate?", in *Proc. 10th ACM SIGPLAN Int. Conf. Software Language Engineering (SLE)*, Vancouver, Canada, Oct. 2017, pp. 256–267. DOI: 10.1145/3136014.3136031.

[28] R. Verdecchia, P. Lago, C. Ebert, and C. de Vries, "Green IT and green software", *IEEE Software*, vol. 38, no. 6, pp. 7–15, Nov. 2021. DOI: 10.1109/MS.2021.3102254.

[29] G. Procaccianti, H. Fernández, and P. Lago, "Empirical evaluation of two best practices for energy-efficient software development", in *Proc. 38th IEEE/ACM Int. Conf. Software Engineering (ICSE)*, Austin, TX, USA, May 2016, pp. 237–246. DOI: 10.1145/2884781.2884803.

[30] S. Herwig and C. Fischer, "Assessment of REST and WebSocket in regards to their energy consumption for mobile applications", in *Proc. IEEE 8th*

Int. Conf. Intelligent Data Acquisition and Advanced Computing Systems (IDAACS), Warsaw, Poland, Sep. 2015. DOI: 10.1109/IDAACS.2015.7340755.

[31] J. Aslan, K. Mayers, J. G. Koomey, and C. France, "Electricity intensity of internet data transmission: Untangling the estimates", *Journal of Industrial Ecology*, vol. 22, no. 4, pp. 785–798, 2018. DOI: 10.1111/jiec.12630.

[32] V. C. Coroama, L. M. Hilty, and M. Heiss, "The direct energy demand of internet data flows", *Journal of Industrial Ecology*, vol. 17, no. 5, pp. 680–688, 2013. DOI: 10.1111/jiec.12038.

[33] T. Salonen, "Green coding and energy efficiency in REST APIs: Empirical evaluation", M.S. thesis, Dept. of Computing, University of Turku, Turku, Finland, Jun. 2025. [Online]. Available: https://www.utupub.fi/bitstream/handle/10024/182395/Salonen_Teemu_opinnayte.pdf.

[34] F. Zafari, J. Li, K. K. Leung, C. Karakus, and A. Raghavan, "Optimal energy consumption for communication, computation, caching, and quality of service guarantee", *IEEE Transactions on Wireless Communications*, vol. 18, no. 8, pp. 3844–3857, Aug. 2019. DOI: 10.1109/TWC.2019.2913800.

[35] PowerGoblin, *PowerGoblin agent documentation*, https://visiiri.tt.utu.fi/powergoblin/manual/usage/agent/, University of Turku, Visiiri Project. Accessed: Jun. 2026.

[36] Docker Inc., *Docker*, https://www.docker.com, Accessed Jun. 2026.

[37] Microsoft, *.NET 8*, https://dotnet.microsoft.com/en-us/download/dotnet/8.0, Accessed Jun. 2026.

[38] SQLite Consortium, *SQLite*, https://www.sqlite.org, Accessed Jun. 2026.

[39] Microsoft, *Microsoft.data.sqlite*, https://learn.microsoft.com/en-us/dotnet/standard/data/sqlite/, Accessed Jun. 2026.

[40] Microsoft, *ASP.NET core*, https://learn.microsoft.com/en-us/aspnet/core/, Accessed Jun. 2026.

[41] graphql-dotnet contributors, *GraphQL.NET*, https://github.com/graphql-dotnet/graphql-dotnet, Accessed Jun. 2026.

[42] ChilliCream Inc., *Hot chocolate*, https://chillicream.com/docs/hotchocolate, Accessed Jun. 2026.

[43] Microsoft, *gRPC for ASP.NET core*, https://learn.microsoft.com/en-us/aspnet/core/grpc/, Accessed Jun. 2026.

[44] M. Gravell, *Protobuf-net.Grpc*, https://github.com/protobuf-net/protobuf-net.Grpc, Accessed Jun. 2026.

[45] Microsoft, *Runtime configuration options for compilation*, https://learn.microsoft.com/en-us/dotnet/core/runtime-config/compilation, Accessed Jun. 2026.

[46] Grafana Labs, *K6*, https://k6.io, Accessed Jun. 2026.

[47] W. H. Kruskal and W. A. Wallis, "Use of ranks in one-criterion variance analysis", *Journal of the American Statistical Association*, vol. 47, no. 260, pp. 583–621, 1952. DOI: 10.1080/01621459.1952.10483441.

[48] O. J. Dunn, "Multiple comparisons using rank sums", *Technometrics*, vol. 6, no. 3, pp. 241–252, 1964. DOI: 10.1080/00401706.1964.10490181.

[49] P. Virtanen et al., "SciPy 1.0: Fundamental algorithms for scientific computing in Python", *Nature Methods*, vol. 17, no. 3, pp. 261–272, 2020. DOI: 10.1038/s41592-019-0686-2.

[50] M. A. Terpilowski, "Scikit-posthocs: Pairwise multiple comparison tests in Python", *Journal of Open Source Software*, vol. 4, no. 36, p. 1169, 2019. DOI: 10.21105/joss.01169.

Appendix A Literature Review References

This appendix provides a complete listing of the references considered during the literature review described in Section 2.1. The first table lists the 33 publications that were included in the review, organised by category. The second table lists the 10 publications that were identified during the search process but excluded from the review, with their categories indicated for reference.

Included References

No.	Paper Title & Authors	Link	Category
[1]	On global electricity usage of communication technology: Trends to 2030 — Andrae & Edler (2015)	Link	Sustainability & Modelling
[2]	Performance comparison of REST API, GraphQL and gRPC — Sliwa & Panczyk (2021)	Link	Performance
[3]	Comparative analysis of RESTful, GraphQL, and gRPC APIs — Chandra & Farisi (2025)	Link	Performance

No.	Paper Title & Authors	Link	Category
[4]	Performance evaluation of microservices communication with REST, GraphQL, and gRPC — Niswar et al. (2024)	Link	Performance
[5]	The GREENSOFT model — Naumann et al. (2011)	Link	Sustainability & Modelling
[6]	Green Software Products — Jagroep (2017)	Link	Sustainability & Modelling
[7]	A measurement-driven approach to enhancing sustainability in microservice architectures — O'Dea et al. (2025)	Link	Energy Efficiency
[8]	Assessing the role of software in sustainability — Bajrami (2025)	Link	Sustainability & Modelling
[9]	ICT Innovations for Sustainability — Hilty & Aebischer (2015)	Link	Sustainability & Modelling
[10]	Recalibrating global data center energy-use estimates — Masanet et al. (2020)	Link	Sustainability & Modelling
[11]	Framing sustainability as a software quality attribute — Lago et al. (2015)	Link	Sustainability & Modelling
[12]	Green software engineering: The curse of methodology — Hindle (2016)	Link	Energy Efficiency
[13]	Mining questions about software energy consumption — Pinto et al. (2014)	Link	Energy Efficiency
[14]	Measuring application software energy efficiency — Capra et al. (2012)	Link	Energy Efficiency
[15]	Protocol Buffers: Developer Guide — Google LLC (2024)	Link	Performance

No.	Paper Title & Authors	Link	Category
[16]	Architectural styles and the design of network-based software architectures — Fielding (2000)	Link	Performance
[17]	Microservice Architecture — Nadareishvili et al. (2016)	Link	Performance
[18]	GraphQL Specification — GraphQL Foundation (2021)	Link	Performance
[19]	REST vs GraphQL: A controlled experiment — Brito & Valente (2020)	Link	Performance
[20]	gRPC: A high performance, open source universal RPC framework — Google (2016)	Link	Performance
[21]	Performance characterization of communication protocols in microservice applications — Kumar et al. (2021)	Link	Performance
[22]	HTTP/2 performance evaluation with latency and packet losses — Oda & Yamaguchi (2018)	Link	Performance
[23]	The state of gRPC in the browser — Brandhorst (2019)	Link	Performance
[24]	A comparative study of REST, GraphQL, and gRPC for API development — Reichersdorfer (2025)	Link	Performance
[25]	A comparative study of REST and gRPC for microservice communication — Johansson & Isabella (2022)	Link	Performance

No.	Paper Title & Authors	Link	Category
[26]	Energy efficiency across programming languages — Pereira et al. (2017)	Link	Energy Efficiency
[27]	Green IT and green software — Verdecchia et al. (2021)	Link	Sustainability & Modelling
[28]	Empirical evaluation of two best practices for energy-efficient software development — Procaccianti et al. (2016)	Link	Energy Efficiency
[29]	Assessment of REST and WebSocket energy consumption for mobile applications — Herwig & Fischer (2015)	Link	Energy Efficiency
[30]	Electricity intensity of internet data transmission — Aslan et al. (2018)	Link	Sustainability & Modelling
[31]	The direct energy demand of internet data flows — Coroama et al. (2013)	Link	Sustainability & Modelling
[32]	Green coding and energy efficiency in REST APIs — Salonen (2025)	Link	Energy Efficiency
[33]	Optimal energy consumption for communication, computation, caching — Zafari et al. (2019)	Link	Sustainability & Modelling

Table A.1: Included References in the Literature Review

Excluded References

No.	Paper Title & Authors	Link	Category
[P1]	A systematic review of Green AI — Verdecchia et al. (2023)	Link	Sustainability & Modelling
[P2]	Empirical study of energy consumption of Android applications — Li et al. (2014)	Link	Energy Efficiency
[P3]	Development and evaluation of a reference measurement model (GSMM) — Guldner et al. (2024)	Link	Energy Efficiency
[P4]	An empirical study of practitioners' perspectives on green software engineering — Manotas et al. (2016)	Link	Sustainability & Modelling
[P5]	Green mining: Relating software change to power consumption — Hindle (2015)	Link	Energy Efficiency
[P6]	Energy Consumption in Microservices Architectures: A Systematic Literature Review — Araujo et al. (2024)	Link	Energy Efficiency
[P7]	An Empirical Evaluation of the Energy and Performance Overhead of Monitoring Tools on Docker-Based Systems — Dinga et al. (2023)	Link	Energy Efficiency
[P8]	On the Effectiveness of Microservices Tactics and Patterns to Reduce Energy Consumption — Bogner et al. (2024)	Link	Energy Efficiency
[P9]	Catalog of Energy Patterns for Mobile Applications — Cruz & Abreu (2019)	Link	Energy Efficiency

No.	Paper Title & Authors	Link	Category
[P10]	Characterizing Energy Consumption of Third-Party API Libraries — Chowdhury et al. (2020)	Link	Energy Efficiency

Table A.2: Excluded References from the Literature Review