Vrije Universiteit Amsterdam

Bachelor Thesis

A Cross-Language Energy Benchmark Using CPU-, Memory-, and I/O-Bound Workloads in Data Processing Pipelines

---

Author: Jonathan Davidson (jdn439)

1st supervisor:	dr. Francesc Verdugo Rojano
daily supervisor:	Ronald Vlaar (Info Support)
2nd reader:	mr. Mikhail Cassar

A thesis submitted in fulfillment of the requirements for the VU Bachelor of Science degree in Computer Science

July 8, 2026

Abstract

The increasing reliance on software-driven systems contributes to rising energy demand, placing pressure on energy grids and computing infrastructure. Previous studies have estimated that information and communication technology could account for up to 14% of global greenhouse gas emissions by 2040 if current trends continue, highlighting the importance of improving the energy efficiency of software systems.

Existing studies on programming language energy efficiency often rely on small synthetic benchmarks or isolated algorithmic tasks. Such benchmarks are useful for controlled comparison, but may not fully capture how applications behave when CPU-oriented, memory-oriented, and I/O-oriented stages are combined. This limits how directly their conclusions can be applied to realistic data-processing workloads.

This thesis addresses this gap by designing and evaluating a reproducible benchmark suite for C, Go, Julia, and Python. The benchmark suite is based on recurring stages in ETL-inspired data-processing pipelines: JSON parsing, byte-buffer processing, sequential file output, and composed pipeline execution. The workloads are evaluated in isolation and as composed end-to-end pipelines using runtime, CPU package energy, DRAM energy, and peak resident set size as metrics.

The results show that no single language is best for all workloads and metrics. C achieves the lowest runtime and CPU package energy in most workload configurations, while Go remains within a small margin of C across workloads. Python frequently achieves low peak memory usage, but incurs high runtime and energy costs in workloads where large byte-wise loops execute at interpreter level rather than through native library code. Julia carries a larger fixed runtime and memory overhead in simple workloads due to JIT compilation startup costs, but matches or outperforms other languages in workloads where LLVM generates vectorized native code for byte-buffer operations. The results also show that library choice and implementation details can substantially affect measured behavior, as seen in the JSON parser variants and the SHA-256/XOR vectorization behavior. Overall, this thesis shows that realistic energy benchmarking should compare representative implementations of workload classes rather than treating programming languages as universally more or less energy efficient. The results demonstrate that efficiency depends on the specific workload and implementation, meaning that no single language consistently performs best across all scenarios.

Keywords: Software benchmarking, programming language execution models, resource utilization, runtime environments, data processing pipelines, the relationship between execution time and energy consumption, and optimization techniques.

Contents

Abstract **2**

1 Introduction **5**
1.1 Research Question 5
1.2 Contributions 6
1.3 Paper overview 6

2 Background **8**
2.1 Software benchmarking 8
2.2 Programming language choice 8
2.3 Data-processing pipelines 8
2.4 Programming language execution models 9
2.5 Runtime environments 9
2.6 Static and dynamic typing 10
2.7 Implementation choices and library effects 10
2.8 Execution time, power, and energy 11
2.9 Bounds checking and vectorization 11

3 Related work **13**

4 Design and implementation **14**
4.1 Benchmark taxonomy principles 14
4.2 Workload categorization 14
4.3 Justification of workload categories 15
4.4 Benchmark implementation requirements 16
4.5 Workload overview 16
4.5.1 JSON parse-transform-serialize 17
4.5.2 SHA-256 / XOR processing 18
4.5.3 Sequential file copy 20
4.5.4 Composed data-processing pipeline 21
4.5.5 Streaming composed pipeline 23
4.6 Dataset generation 25
4.6.1 Yelp-derived JSON dataset 25
4.6.2 SHA256-XOR input generation 25
4.6.3 File copy input generation 26
4.6.4 Pipeline input generation 26
4.7 Scaling strategy 27
4.8 Implementation methodology and language-specific decisions 27
4.8.1 Library selection and JSON parser variants 28
4.8.2 SHA-256/XOR implementation behavior 29

5 Experiments **30**
5.1 Benchmark framework and execution workflow 30
5.2 Hardware configuration and software environment 31
5.3 Energy measurement method 33

5.4 Memory and resource monitoring 33
5.5 Main performance metrics 34

6 Results **35**
6.1 Energy results per workload 35
6.2 Runtime results per workload 36
6.3 Peak memory usage 37
6.4 Cross-language comparison 39
6.5 JSON parser variants 39
6.6 Isolated workloads and composed pipelines 40
6.7 Correlation Analysis 41

7 Discussion **42**
7.1 Interpretation of results 42
7.2 Threats to validity / comparability 45
7.3 Limitations and future work 46

8 Conclusion **48**

1 Introduction

The increasing reliance on software-driven systems contributes to increased electricity demand, placing additional pressure on already congested energy grids. In the Netherlands, for example, grid congestion has become a growing concern, leading to restrictions on new connections in several regions and highlighting the challenges of accommodating increasing electricity consumption [1]. As societies become increasingly dependent on digital services, improving the energy efficiency of computing systems has become an important sustainability challenge. Previous studies have estimated that Information and Communication Technology (ICT) could account for up to 14% of global greenhouse gas emissions by 2040 if current trends continue, highlighting the importance of improving the energy efficiency of software systems [2].

The energy consumption of computing systems is determined not only by the hardware, but also by the software that runs on them. Although hardware determines the physical energy cost of computation, software determines what computations are performed, how data are processed, and how efficiently hardware resources are used [3]. Consequently, software implementation choices can influence the energy required to complete a task. Since equivalent programs written in different programming languages can differ substantially in runtime, energy consumption, and memory usage, the choice of programming language may affect the resources required to complete the same task [4].

Current work on energy-efficient software and programming language comparison is largely based on benchmark-driven evaluation. In particular, studies such as Pereira et al. and Gordillo et al. provide reproducible methodologies for comparing programming languages in terms of runtime, energy consumption, and memory usage [5, 4, 6]. However, these approaches often rely on small synthetic benchmarks or isolated computational tasks, which may not fully capture how applications behave between CPU, memory, and I/O-bound workloads or how these workload characteristics interact during composed execution. Jansson and Johansson similarly motivate the need for application-level benchmarking by questioning whether synthetic benchmark scenarios reflect the performance of real-world applications [7]. As a result, there remains a need for reproducible evaluations that compare programming languages not only between isolated workload classes, but also in a controlled composed workload where CPU computation, memory access, and I/O operations occur together.

This thesis addresses this gap by implementing and evaluating a reproducible benchmark suite composed of CPU-, memory-, and I/O-bound workloads across multiple programming languages, including compiled, just-in-time (JIT) compiled, and interpreted languages. The proposed methodology evaluates these workloads both in isolation and as a composed end-to-end data-processing pipeline, where input data is read, processed in memory, transformed through CPU computation, and written back as output.

1.1 Research Question

To guide this work, the following primary research question is formulated.

What differences can be observed between programming languages in runtime, CPU package energy, DRAM energy, and peak memory usage across isolated and composed data-processing workloads?

In addition to this main research question, several sub-questions are addressed.

- **RQ1:** What criteria are required to design CPU-, memory-, and I/O-oriented workloads that are comparable and reproducible across programming languages?
- **RQ2:** What differences can be observed between individual workload classes and composed end-to-end data-processing pipelines in terms of runtime, energy consumption, and peak memory usage?
- **RQ3:** How do the size of the input and repeated executions affect the consistency of runtime, energy consumption, and peak memory measurements between workloads?

1.2 Contributions

The main contributions of this thesis are:

1. **Workload-based methodology** A method for comparing programming languages across isolated and composed CPU, memory, and I/O-oriented workloads, enabling evaluation of specific workloads between languages.
2. **Reproducible benchmark suite** This thesis implements a cross-language benchmark suite with shared inputs, equivalent workloads, and output validation, ensuring that measured differences reflect implementation behavior rather than differences in workload behavior.
3. **Energy-domain evaluation** This thesis evaluates runtime, CPU package energy, DRAM energy, and peak memory usage using Intel Running Average Power Limit (RAPL), providing insights into both computational and memory-related energy costs.
4. **Composition analysis** This thesis compares isolated workloads with end-to-end data-processing pipelines, showing whether isolated benchmark results remain representative when workloads are combined.
5. **Scaling and stability analysis** This thesis analyzes input-size scaling and repeated execution, supporting a reliable interpretation of measurement variability and fixed execution overheads.

1.3 Paper overview

This paper is structured as follows, **Chapter 2** provides the background information needed to understand the concept of benchmarking of software energy efficiency. **Chapter 3** discusses related work on programming language energy efficiency and benchmark design. **Chapter 4** Introduces the classification of workload types, in order to differentiate between CPU, Memory, and I/O-oriented workloads. Furthermore, it describes the selected workloads, including but not limited to their pseudo-code, their input generation, the scaling strategy used to evaluate behavior across different input sizes, and describes the implementation choices made. **Chapter 5** presents the experimental setup of the benchmark suite and explains how runtime, energy consumption, and memory usage

are measured. **Chapter 6** presents the benchmark results across languages, workloads, and input sizes, focusing on execution time, CPU package energy, DRAM energy, total energy-to-solution, and peak memory usage. **Chapter 7** discusses these results, answers research questions, and discusses threats to validity, limitations, and future work. Finally, **Chapter 8** concludes the thesis by summarizing the main findings.

2 Background

This chapter provides the necessary background for understanding the remainder of the paper. It introduces software benchmarking, programming language execution models, runtime environments, and the computational resources considered in this study. It also explains the relationship between execution time, power, and energy consumption and outlines relevant software optimization concepts that influence performance and energy efficiency.

2.1 Software benchmarking

Software benchmarking refers to the systematic measurement and comparison of software performance under controlled and repeatable conditions [8]. In the context of this research, benchmarking involves executing a set of predefined workloads across multiple programming languages to compare their execution behavior and energy efficiency. Furthermore, a workload is a piece of code that implements a specific data-processing task, such as parsing JSON data, hashing an input buffer, or copying a file. An orchestrator executes the workload by providing input data, invoking the workload function, recording the measured behavior, and optionally verifying the correctness of the output. The orchestrator and a workload implementation in a specific programming language together form a benchmark. For example, a JSON parsing workload implemented in Python constitutes one benchmark, while the same workload implemented in C constitutes another benchmark. The complete collection of benchmark implementations across all workloads and programming languages forms the benchmark suite used in this study.

2.2 Programming language choice

In this paper, four languages were selected to cover a representative range of runtime and memory management strategies. C was selected as a low-level performance baseline with manual memory management, as previous work has shown it to be among the most energy-efficient and fastest compiled languages [4]. Go was included as a garbage-collected alternative that remains competitive in performance, though Pereira et al. report it consuming roughly three times the energy of C in their benchmarks [4]. Python was selected for its widespread adoption in data processing pipelines [9]. Julia represents a JIT-compiled language designed for high-performance numerical computation, where compilation overhead may interact differently with varying input sizes.

2.3 Data-processing pipelines

Data-processing pipelines are commonly structured as multi-stage workflows in which data is collected from input sources, transformed, and written to a target system for storage or further analysis. A common model for this is the Extract-Transform-Load (ETL) paradigm, which is widely used in data warehouse and analytical systems [10]. Large-scale systems use similar pipeline structures in practice. For example, Facebook's data warehousing infrastructure combines log collection, storage, and analytics systems such as Scribe, Hadoop, and Hive to process large volumes of application data [11].

This thesis does not attempt to reproduce such a distributed production system. Instead, the ETL model is used as an example of a controlled benchmark pipeline. In this context, *realistic* means that the benchmark is based on stages from data-processing workflows, rather than on isolated arithmetic kernels alone. The pipeline therefore combines semi-structured data transformation, byte-level processing, and file output. This structure makes it possible to compare the behavior of isolated workloads with the behavior of a composed end-to-end data-processing pipeline.

2.4 Programming language execution models

Programming languages differ significantly in how they compile code, in other words, in how they translate human-written program code into machine-executable instructions. Compiled languages such as C and Go are translated into machine code prior to execution, typically producing optimized binary files that interact directly with the hardware. In contrast, interpreted or virtual-machine-based languages execute instructions through an interpreter or runtime environment during program executions [12]. These differences in execution models influence how programs interact with system resources and can lead to variations in runtime performance and energy consumption.

2.5 Runtime environments

Programming languages and runtime systems often apply various optimization techniques to improve performance and reduce resource usage. These techniques may occur during compilations or at runtime [13], and include strategies such as compiler optimizations, just-in-time (JIT) compilation, and garbage collection [14].

As mentioned above, compiled languages such as C use a compiler to translate code to machine instructions. During this translation, the compiler can apply optimizations, such as removing unnecessary operations, simplifying expressions, or reorganizing instructions to improve performance. Once the program has been compiled, the generated machine code is executed directly by the processor and is not normally changed during execution [15].

Go is also compiled ahead of time to native machine code, but differs from C because it includes a runtime system. This runtime provides services such as garbage collection, stack management, and scheduling of lightweight threads. Go uses automatic memory management through a tracing garbage collector: live objects are found by tracing references from roots such as stacks and global variables, and unreachable heap objects can then be reclaimed. Modern Go performs much of this garbage collection work concurrently with program execution, although garbage collection can still introduce runtime and memory-management overhead. These features can improve programmer productivity and safety, but may also affect execution time, memory usage, and energy consumption [16, 17].

Python, in the CPython implementation used in this thesis, follows a different model. Python source code is first translated to bytecode, which is then executed by the Python interpreter. CPython primarily manages the object lifetime using reference counting: when an object's reference count reaches zero, it can be deallocated. This is supplemented by a cyclic garbage collector for groups of objects that refer to each other and cannot be freed by counting references alone [18]. This memory-management model differs from Go's

tracing collector, but Python operations still typically involve more runtime interpretation overhead than equivalent operations in compiled languages. As a result, Python programs often depend heavily on the efficiency of the interpreter and on optimized libraries implemented in lower-level languages [19].

Julia uses just-in-time (JIT) compilation. Instead of compiling the entire program before execution, Julia compiles functions at runtime when they are called with specific argument types. This allows Julia to generate optimized native machine code using information that is only available during execution. Julia also includes automatic memory management through its runtime system. However, JIT compilation and runtime infrastructure mean that the first execution of a function may include compilation overhead, often referred to as a startup or warm-up cost [15].

Because C, Go, Python, and Julia rely on different optimization mechanisms, their execution behavior and energy efficiency may vary. These execution models are closely related to how languages represent and check variable types, which is discussed in the next section.

2.6 Static and dynamic typing

Programming languages also differ in how they handle variable types. In statically typed languages, such as C and Go, the types of variables and function arguments are known and checked at compile time. This allows the compiler to detect many type errors before execution and generate machine code based on explicit type information. The declared type of a variable normally remains fixed, although some languages provide abstractions such as interfaces or generic types that can refer to values with different concrete runtime types; these mechanisms are not central to the workloads used in this paper [20, 17].

In dynamically typed languages, such as Python, type information is generally checked at runtime. This provides flexibility, but it can also introduce overhead because the runtime system must inspect and manage object types while the program is executing. Julia is also dynamically typed at the language level, but its JIT compiler specializes in functions for the concrete argument types used at runtime. This allows Julia to combine a dynamic programming model with optimized machine-code generation for performance-critical code [19, 21]. These differences are relevant for this thesis because type handling can influence execution time, memory usage, and therefore energy consumption.

2.7 Implementation choices and library effects

Software performance is influenced not only by the programming language itself, but also by the way a program is implemented and by the libraries used to perform specific tasks. For example, JSON parsing, cryptographic hashing, and file I/O may be provided by standard libraries or external packages. Keiser and Lemire illustrate the differences in performance by comparing some JSON parsers for C, demonstrating this concept [22]. Highlighting that these implementations may differ in algorithmic design, memory allocation behavior, and the amount of work performed per job.

This is important for cross-language benchmarking because a comparison between languages is also affected by the system surrounding each language. A Python program that relies on an optimized native library may behave differently from a pure Python implementation, while a C implementation may expose more manual control over memory and

data layout. Similarly, Go and Julia provide runtime systems and standard libraries that influence allocation behavior, garbage collection, and I/O performance [17, 21].

For this reason, this thesis attempts to use idiomatic implementations of the same workload structure in each language rather than to attempt to produce manually optimized code for each language independently. Implementations are considered idiomatic when they follow each language's own conventions and make use of its standard libraries, even if this means the implementations differ in how they express the same underlying logic. This reflects how languages are commonly used in practice and reduces the risk that the comparison measures only the skill or optimization effort of the programmer. Cross-language benchmarking inevitably includes implementation choices made by the benchmark author. To reduce this human factor, each workload follows the same algorithmic structure across languages, uses equivalent input data and verification logic, and relies on standard or commonly used libraries rather than hand-optimized language-specific tricks.

In this paper, the term *human factor* refers to the influence of the decisions of the benchmark-author on the measured results. These decisions include algorithm design, library selection, coding style, use of language-specific features, and the amount of optimization effort applied to each implementation.

2.8 Execution time, power, and energy

Execution time and energy consumption are related but distinct performance metrics [5]. Execution time represents the duration required for a program or a task to complete, whereas energy consumption represents the total amount of energy used during execution. The relation between energy, power, and time can be expressed as: $E = P \times t$ [23]. where E denotes energy, P represents the average power consumption, and t is the execution time. This relationship illustrates that a reduction in execution time does not necessarily lead to a decrease in energy consumption, since power usage may vary during program execution. As will be shown in Chapter 6.7, the benchmarks in this thesis show a linear relationship between runtime and energy in all workloads and languages.

2.9 Bounds checking and vectorization

Bounds checking is a safety mechanism that verifies whether, for example, an array or slice access is within the valid range of that object. If a program attempts to access an element outside this range, the language can raise an error instead of allowing invalid memory access. This improves safety, but the additional checks may introduce overhead. Some compilers can remove bounds checks when they can prove that the accessed variables are always inside the object. This is known as bound-check elimination. Whether such checks are present depends on the programming language, the compiler, the data structure, and the specific loop structure [17].

C normally does not perform runtime bounds checks for array or pointer accesses. Go performs bounds checks for array and slice accesses, although the compiler can sometimes eliminate them. Python performs bounds checks for sequence accesses. Julia also performs bounds checks by default, but allows programmers to disable them in code regions using annotations such as `@inbounds` [20, 17, 19, 21].

Vectorization is an optimization technique in which multiple data elements are processed

by a single machine instruction. Modern processors provide vector instructions that operate on several values at once, instead of processing one value per instruction, which is what a scalar instruction does. For example, in a byte-processing loop, a scalar version may process one byte per loop iteration, while a vectorized version may process multiple bytes together using one vector instruction. A compiler may transform a scalar loop into a vectorized loop when it can prove that this transformation is safe and profitable. This is known as auto-vectorization. Vectorization can improve runtime performance and may also affect energy consumption, because the processor can complete the same amount of work using fewer loop iterations and instructions [24].

However, vectorization is not guaranteed. The language compiler must be able to reason that a code function is safe. As a result, two implementations in different languages of the same high-level algorithm may differ substantially depending on whether the bound checks remain in the loop and whether the loop is compiled into vector instructions [25].

3 Related work

Research on programming language energy efficiency has often used benchmark-driven evaluation. One of the most influential studies in this area is the work of Pereira et al. [5], which compares the energy consumption, execution time, and memory usage of 27 programming languages using a suite of benchmark programs. Their study shows that language implementations can differ substantially in energy consumption and that the fastest language is not always the most energy-efficient. This work provides an important foundation for cross-language energy comparison, but its evaluation is primarily based on isolated benchmark workloads, which may not fully capture interactions between different computational components present in data-processing applications.

More recent work has questioned the extent to which such benchmark results generalize. Martins et al. [26] revisit programming language energy-efficiency measurements and argue that the results can be strongly affected by implementation choices, runtime behavior, and experimental configuration. Their findings suggest that language comparisons should carefully control confounding factors and avoid drawing broad conclusions from a small set of benchmark implementations. Similarly, Gordillo et al. [6] study programming language rankings based on energy measurements and highlight the methodological challenges involved in producing reliable rankings. These studies show that energy-efficiency evaluation is sensitive not only to the language but also to the selected implementation, language runtime, and measurement procedure.

Energy measurement itself is also a relevant concern in software energy studies. Previous work has investigated different approaches for measuring or estimating software energy consumption, including hardware sensors, external power meters, and model-based techniques [3]. In this paper, energy is measured using Intel RAPL counters, which provide access to the CPU package and DRAM energy domains on the experimental platform. The use of such counters is common in software energy studies because they enable fine-grained energy measurements without requiring external measurement equipment.

A second line of related work concerns the limitations of microbenchmark-based evaluation. Jansson and Johansson [7] argue that benchmarks should reflect application-level execution patterns rather than rely only on isolated workloads. Their work is relevant because isolated workloads can be useful in understanding specific operations, but they may not capture the behavior that emerges when multiple operations are combined. Kharal and Brown [27] further demonstrate that microbenchmark design choices can strongly influence measured performance, in some cases changing results by orders of magnitude 10 to 100 or even reversing conclusions. These findings motivate evaluating both isolated workload classes and composed executions.

Workload characterization research also shows that programs can stress different parts of a computing system. The Roofline model distinguishes between compute-bound and memory-bound behavior by relating computational throughput to memory bandwidth [28]. The memory wall concept further illustrates how memory access costs can become a limiting factor in program performance [29]. These models motivate separating workloads according to dominant resource behavior, such as CPU-oriented, memory-oriented, and I/O-oriented execution.

4 Design and implementation

This chapter describes the design principles of the benchmark suite, the selected workload classes, their algorithms, and the input data generation methodology. The goal is not to create a complete taxonomy of software workloads, but rather to define a small, reproducible set of workload classes that exercise different system resources while remaining comparable across programming languages. The selection of workloads is guided by the ETL-inspired pipeline model introduced in Chapter 2.3. The remainder of this chapter first discusses the benchmark design principles and the resulting workload classes, followed by descriptions of the individual workloads, including pseudo-code and input dataset generation procedures.

4.1 Benchmark taxonomy principles

Herten et al. [30] survey high-performance computing (HPC) benchmarks and propose a taxonomy to characterize benchmarks along several dimensions, including the benchmark scale, compute-performance characteristics, memory-access characteristics, communication behavior, method type, programming language, and programming model. Their taxonomy illustrates that benchmark suites can be described not only by application domain, but also by the system resources and computational patterns they exercise. Inspired by this characterization approach, this thesis groups workloads by their dominant resource pressure: CPU-oriented, memory-oriented, I/O-oriented, and composed workloads.

1. CPU-bound workloads, dominated by arithmetic intensity and compiler efficiency.
2. Memory-bound workloads, dominated by allocation behavior, cache locality, and runtime memory management.
3. I/O-bound workloads, dominated by file system and buffering overhead.

4.2 Workload categorization

At this stage, we define abstract workload categories rather than concrete benchmark implementations. The specific workloads that initialize these categories are selected in the following section. Table 1 summarizes the dominant resource bottleneck associated with each workload category. Although many additional workload classes could be included in a broader benchmark suite, this thesis focuses on these four classes because they correspond to the implemented benchmarks and cover the main resource pressures studied in the evaluation: CPU package energy, DRAM energy, runtime, and peak memory usage.

Workload class	Concrete benchmark	Dominant resource pressure
Parsing and serialization	JSON parse-transform-serialize	CPU and memory
Cryptography and hashing	SHA-256 and XOR processing	CPU
File stream and I/O	Sequential file copy	I/O
Composed pipeline	JSON + cryptography + file output	Mixed

Table 1: Workload classes used in the benchmark suite

4.3 Justification of workload categories

1. **Parsing and serialization workloads** process structured text by reading JSON input, identifying JSON tokens, validating the document structure, decoding values, and often constructing language-level data structures for further processing. Previous work describes JSON ingestion as a potential performance bottleneck and notes that many parsers convert input text into in-memory representations of objects, arrays, strings, numbers, and primitive values [22]. JSON parser studies also evaluate both CPU and memory usage, reflecting that parsing involves more than simply reading bytes from an input file [31]. In this paper, the JSON workload performs a parse-transform-serialize cycle. It parses Yelp-derived JSON records, modifies fields, adds a derived field, and serializes the result. Therefore, this workload exercises both CPU work, through token scanning, validation, decoding, string transformation, and serialization, and memory pressure, through allocation and traversal of language-level data structures. For this reason, JSON parsing and serialization is categorized as a workload with both CPU and memory pressure.

2. **Cryptographic and hashing workloads**, such as SHA-256 hashing and XOR-based processing, repeatedly apply arithmetic and bitwise operations over input buffers. The SHA-256 specification defines its compression function using operations such as rotations, shifts, logical functions, and modular additions over fixed-size words [32]. This makes hashing a suitable CPU-oriented workload: the input is streamed through a compact computation kernel, and performance depends strongly on instruction throughput, compiler optimization, and hardware support for relevant operations. The work on SHA-512/256 also shows that the cost of SHA-family hashing depends on the architecture of the processor [33]. In this benchmark suite, hashing is therefore used to represent computationally heavy processing.

3. **File stream and I/O workloads**, such as sequential large-file copying, are dominated by moving bytes between storage, operating-system buffers, and user-space memory. Unlike hashing, where most work is performed by arithmetic and bitwise instructions, file copying depends heavily on file-system behavior, buffering, system-call overhead, page cache effects, and storage bandwidth. Previous work on disk and file-system performance shows that sequential I/O performance is strongly affected by storage-system behavior and benchmark configuration [34, 35]. This workload is therefore used to represent I/O-oriented behavior in the benchmark suite.

4. **The composed pipeline** combines the previous workload classes into a single end-to-end pipeline: JSON parsing and transformation, cryptographic processing, and file output. This category is included because isolated workload benchmarks may not necessarily capture the behavior that emerges when multiple operations are integrated into one execution flow. In the context of application performance testing, Grambow et al. distinguish between application benchmarks, which include related components, and isolated benchmarks, which repeatedly execute individual functions. They note that microbenchmarks may miss problems that appear only when functions or modules are integrated into the overall system [36]. This thesis tries to apply this idea to cross-language energy benchmarking by evaluating both isolated workloads and a composed data-processing pipeline. In the composed

pipeline, intermediate data sizes, allocation behavior, cache effects, garbage collection, and I/O buffering may interact. As a result, the dominant bottleneck may differ from the bottlenecks observed in isolated workloads. The pipeline benchmark therefore evaluates whether language behavior observed in isolated CPU-, memory-, and I/O-oriented workloads remains consistent when these stages are combined into one data-processing workflow.

4.4 Benchmark implementation requirements

After defining the workload categories in Section 4.2, concrete benchmark implementations must still satisfy a set of comparability and reproducibility requirements. These requirements ensure that the implemented workloads perform equivalent work across programming languages and can be evaluated consistently across input sizes.

1. The workloads should correspond to recurring stages in the data-processing pipelines, such as data parsing, byte-level processing, and file output.
2. Each workload should exercise one or more of the resource-pressure classes defined in Section 4.1.
3. The workload must be implementable in each selected language using the same high-level algorithmic steps.
4. The implementations should perform equivalent work and produce verifiable output across languages.
5. The workloads should support input scaling so that the runtime, energy, and memory behavior can be evaluated across different problem sizes.
6. The workload output should be deterministic for a fixed input.

Benchmark	Workload	Input	Dominant pressure
json_parse	Parsing and serialization	JSON records	CPU and memory
sha256	Hashing and byte processing	Binary buffer + keys	CPU
file_copy	Sequential file output	Binary file	I/O
pipeline_e2e	Composed pipeline	JSON + keys	Mixed
pipeline_e2e_stream	Streaming pipeline	JSON chunks + keys	Mixed

Table 2: Implemented benchmarks and their intended resource pressure

4.5 Workload overview

This section defines the concrete workloads used in the benchmark suite. The first three workloads isolate JSON processing, byte-buffer processing, and file-output behavior. The final two workloads combine these stages into non-streaming and streaming pipeline variants.

4.5.1 JSON parse-transform-serialize

Purpose The JSON workload evaluates structured-data parsing, record transformation, and serialization. The workload is designed to exercise both CPU and memory resources. CPU work is introduced through token scanning, decoding, field lookup, making strings uppercase, and serialization. Memory pressure is introduced through the construction, traversal, and serialization of data structures. The workload receives a JSON document that contains an array of objects under the key `elements`. Each implementation parses the document, iterates over all items, modifies selected fields, and serializes the modified document. The transformations chosen in Algorithm 1 represent common operations in data-processing workloads, including numeric updates, boolean changes, string manipulation, and derived-field computation, while remaining simple enough to ensure comparable implementations across programming languages.

Algorithm 1 WORKLOAD_JSON_PARSE

```
1:  sink
2:  function WORKLOAD_JSON_PARSE(input_json, output_json, output_len)
                          ▷ Parse input JSON text into language-level data structures
3:      root ← JSON_PARSE(input_json)
                                                          ▷ Retrieve array of items
4:      items ← root["items"]
                                                  ▷ Transform each object in the array
5:      for all item in items do
6:          item["id"] ← item["id"] + 1
7:          item["active"] ← ¬item["active"]
8:          upper ← TOUPPERCASECOPY(item["name"])
9:          item["name"] ← upper
10:         item["name_length"] ← UTF8BYTELENGTH(upper)
11:     end for
                                                          ▷ Serialize transformed JSON
12:     serialized ← JSON_SERIALIZE_UNFORMATTED(root)
                                  ▷ Release parsed JSON representation if required
13:     RELEASE_IF_REQUIRED(root)
                                                          ▷ Store output values
14:     output_json ← serialized
15:     output_len ← LENGTH(serialized)
                                                  ▷ Prevent compiler optimization
16:     sink ← sink ⊕ (output_len mod 256)
17:     sink ← sink ⊕ ByteAt(serialized, 0)
18:     return 1
19: end function
```

Algorithm 1 shows the JSON parse-transform-serialize workload. The algorithm first parses the input JSON document into language-level data structures and retrieves the `items` array. Then it applies the same deterministic transformation to every item: increment `id`, negating `active`, uppercase `name`, and adding `name_length`. Finally, the

modified document is serialized back to JSON and the selected output values are written to a sink variable to prevent the result from being optimized away. A sink in the context of this research refers to a variable which serves as a placeholder to XOR the output data, in order to prevent the compiler from optimizing any intermediate steps, as our sink would be dependent on the result of all steps combined.

Input The input is a JSON document with the structure {`"items": [...]`}. The workload assumes that each item contains the fields required by the transformation. The procedure used to generate these input files is described in Section 4.6.1.

Output The output is a serialized JSON document containing the transformed `items` array. The output is kept in memory by the isolated JSON benchmark and is not written to disk. The output length is stored separately, and the selected output values are combined into a sink variable to ensure that the workload result is used.

Correctness condition A JSON workload execution is considered correct if the output can be parsed as valid JSON, contains an `items` array, and every item contains the required fields `id`, `active`, `name`, and `name_length` with the expected types. The verifier checks that each `name` field is uppercase and that `name_length` equals the byte length of the transformed `name`. The verifier therefore validates the structure of the transformed output and the derived field used by the workload, rather than comparing the full output against a stored reference file.

Scaling variable The workload is scaled by the input size S. Larger inputs contain more JSON records in the `items` array, while the transformation applied to each record remains unchanged.

4.5.2 SHA-256 / XOR processing

Purpose The SHA-256 workload evaluates CPU-oriented processing over a binary input buffer. The workload applies three deterministic transformations to the input data: XOR-based encryption, HMAC-SHA256 computation over the encrypted data, and XOR-based decryption. The XOR stages perform simple byte-wise operations over the full buffer, while the HMAC-SHA256 stage performs cryptographic hashing over the encrypted buffer.

This workload is intended to exercise processor execution rather than file-system I/O. It performs sequential reads and writes over memory buffers, but no external files are read or written during the measured workload. SHA-256 computation consists of repeated logical, rotation, shift, and modular-addition operations over fixed-size words, making it suitable for CPU-oriented workload [32]. Previous work on SHA hashing also shows that hashing throughput depends on processor architecture and implementation choices [33].

Algorithm 2 WORKLOAD_SHA256

```
1: sink
2: function WORKLOAD_SHA256(buffer, buffer_len, key_hmac, key_hmac_len, xor_key,
   xor_key_len, ciphertext, decrypted, output_hmac)
                                          ▷ Encrypt input buffer using repeated XOR
3:     for i ← 0 to buffer_len − 1 do
4:         ciphertext[i] ← buffer[i] ⊕ xor_key[i mod xor_key_len]
5:     end for
                                          ▷ Compute HMAC-SHA256 over encrypted data
6:     HMAC_SHA256(key_hmac, key_hmac_len, ciphertext, buffer_len, output_hmac)
                                          ▷ Decrypt ciphertext using the same XOR key
7:     for i ← 0 to buffer_len − 1 do
8:         decrypted[i] ← ciphertext[i] ⊕ xor_key[i mod xor_key_len]
9:     end for
                                          ▷ Prevent compiler or runtime from discarding the result
10:    sink ← sink ⊕ output_hmac[0]
11:    sink ← sink ⊕ decrypted[0]
12:    return 1
13: end function
```

Algorithm 2 shows the SHA-256 and XOR processing workload. The algorithm first encrypts the input buffer using a repeated XOR key. Then it computes an HMAC-SHA256 digest over the encrypted buffer. Finally, it decrypts the ciphertext by applying the same XOR operation again. Since XOR is its own inverse, applying the same XOR operation twice recovers the original input buffer. Selected output bytes are combined into a sink variable to prevent the compiler or runtime from removing the computation.

Input The input consists of a binary buffer of size S, a fixed HMAC key of the same size, and a fixed XOR key. The buffer contains deterministic pseudo-random bytes generated before benchmark execution. The same generated input files are used for all language implementations.

Output The workload produces three outputs: the encrypted buffer, the decrypted buffer, and the HMAC-SHA256 digest. In the isolated benchmark, these outputs are kept in memory and are not written to disk. Selected output bytes are used in a sink variable to ensure that the computation is observable.

Correctness condition Execution of a SHA256 workload is considered correct if the decrypted buffer is identical to the original input buffer prior to encryption and the HMAC output has a fixed length of 32 bytes. This ensures that the XOR-based encryption is reversible and that the HMAC computation is performed over the encrypted buffer.

Scaling variable The workload is scaled by the input buffer size S. Larger inputs increase the number of bytes processed by the XOR and HMAC-SHA256 stages, while the key sizes remain fixed.

4.5.3 Sequential file copy

Purpose The file workload evaluates sequential file output and synchronization behavior. The benchmark entrypoint first loads a binary input file into memory. The workload then writes this in-memory buffer to a destination file multiple times and periodically synchronizes the written data to storage. This design reduces computational work and emphasizes the cost of moving data through the operating system and the file system.

The workload is intended to exercise I/O behavior rather than arithmetic computation. Apart from loop control, byte counting, and sink updates, the workload performs no substantial data transformation. Its runtime and energy behavior are therefore expected to depend mainly on file-system behavior, buffering, system-call overhead, synchronization cost, and storage throughput. Previous work on disk and file-system performance shows that I/O results are strongly dependent on storage-system behavior and benchmark configuration [34, 35].

Algorithm 3 WORKLOAD_FILE_COPY

```
1:  sink
2:  function WORKLOAD_FILE_COPY(dst, buffer, buffer_len)
3:      bytes_since_sync ← 0
                                          ▷ Write the in-memory buffer multiple times
4:      for i ← 1 to WRITEAMPLIFICATION do
5:          WRITEALL(dst, buffer, buffer_len)
6:          bytes_since_sync ← bytes_since_sync + buffer_len
                                                  ▷ Flush written data periodically
7:          if bytes_since_sync ≥ SYNCINTERVAL then
8:              DATASYNC(dst)
9:              bytes_since_sync ← 0
10:         end if
11:     end for
                                                  ▷ Flush remaining written data
12:     DATASYNC(dst)
                          ▷ Prevent compiler or runtime from discarding auxiliary work
13:     sink ← sink ⊕ (buffer_len mod 256)
14:     sink ← sink ⊕ ByteAt(buffer, 0)
15:     return 1
16: end function
```

Algorithm 3 shows the sequential file-output workload. The algorithm receives a destination file descriptor and an in-memory buffer. It writes the complete buffer to the destination file multiple times, using a write amplification factor of 8, which is a constant and can be altered in the implementation. After every 64 MiB of written data, the workload calls a data synchronization operation, and it performs a final synchronization after all writes have been completed. Selected buffer values are combined into a sink for consistency with the other workloads. Although the benchmark is named `file_copy`, the workload function operates on a preloaded input buffer and measures sequential repeated output to a destination file.

Input The input is a binary file of size S. Before being invoked, the benchmark orchestrator reads this file into an in-memory buffer. This memory pointer is passed to the workload. The same generated input file is used for all language implementations.

Output The output is a file containing the input buffer repeated according to the amplification factor. In the implementation used in this thesis, the buffer is written eight times, so the output file size is $8S$. The output file is removed after benchmark execution by the orchestration script to avoid reusing stale output.

Correctness condition The execution of a file workload is considered correct if the output file contains exactly the input buffer repeated the expected number of times and contains no additional bytes. The verification step reads the output file and compares it byte-by-byte with the amplified input buffer.

Scaling variable The workload is scaled by the input buffer size S. Since the write amplification factor is fixed, the number of bytes written scales linearly with S.

4.5.4 Composed data-processing pipeline

Purpose The composed data-processing pipeline evaluates how isolated workload stages behave when combined into a single end-to-end execution flow. The pipeline combines the JSON-parse workload, the SHA-256/XOR processing workload, and the sequential file-output workload. This allows the evaluation to compare the behavior of isolated workloads with the execution of their composed task, where data sizes, allocation behavior, cache effects, garbage collection, and file-system output may interact.

The benchmark orchestrator reads the JSON input, the HMAC key, and the XOR key before invoking the workload function. The workload itself first transforms the complete JSON input, then cryptographically processes the transformed JSON buffer, and finally writes the processed buffer to an output file using the same amplified output behavior as the isolated file workload.

Algorithm 4 WORKLOAD_PIPELINE_E2E

```
1: sink
2: function WORKLOAD_PIPELINE_E2E(input_json, key_hmac, key_xor, ciphertext, decrypted, out_hmac, dst)
                                          ▷ Stage 1: parse, transform, and serialize JSON
3:     transformed_json ← ∅
4:     transformed_len ← 0
5:     WORKLOAD_JSON_PARSE(input_json, transformed_json, transformed_len)
                                ▷ Stage 2: encrypt, hash, and decrypt transformed JSON
6:     WORKLOAD_SHA256(transformed_json, transformed_len, key_hmac, key_xor, ciphertext, decrypted, out_hmac)
                                ▷ Stage 3: write decrypted output using amplified file output
7:     WORKLOAD_FILE_COPY(dst, decrypted, transformed_len)
                                ▷ Prevent compiler or runtime from discarding the result
8:     sink ← sink ⊕ (transformed_len mod 256)
9:     sink ← sink ⊕ ByteAt(decrypted, 0)
10:    sink ← sink ⊕ out_hmac[0]
                                          ▷ Release transformed JSON buffer if required
11:    DeleteIfRequired(transformed_json)
12:    return 1
13: end function
```

Algorithm 4 shows the non-streaming composed pipeline. The first stage invokes the JSON workload on the complete input document and produces a transformed serialized JSON buffer. The second stage invokes the SHA-256/XOR workload on this transformed buffer, producing a ciphertext buffer, a decrypted buffer, and an HMAC digest. The third stage invokes the file-output workload, which writes the decrypted buffer to the destination file using the fixed write amplification factor. Finally, selected output values are combined into a sink variable to prevent the compiler or runtime from discarding the computation.

Input The input consists of a JSON document of size S, an HMAC key, and an XOR key. The JSON document has the same structure as the isolated JSON workload, namely a root object containing an `items` array. The HMAC and XOR keys are generated deterministically as part of the SHA-256 dataset generation.

Output The output is a file containing the decrypted result of the cryptographic stage. Because the SHA-256/XOR stage decrypts the ciphertext back into the transformed JSON buffer, the file-output step writes the transformed JSON content. Similarly to the isolated file-output workload, the buffer is written using a fixed write amplification factor, so the output file contains the transformed JSON buffer repeated multiple times.

Correctness condition A composed pipeline execution is considered correct if the decrypted buffer produced by the SHA-256/XOR stage is equal to the transformed JSON buffer produced by the JSON stage and if the output file contains the decrypted buffer repeated according to the write amplification factor. The verification step recomputes the

JSON transformation, compares it with the decrypted buffer, and checks that the output file contains the expected amplified content.

Scaling variable The workload is scaled by the size of the JSON input S. Increasing S increases the number of records parsed and transformed, the number of bytes processed by the cryptographic stage, and the number of bytes written by the file-output stage.

4.5.5 Streaming composed pipeline

The streaming composed pipeline processes through the same logical states as the non-streaming pipeline, however, processes the JSON input in chunks. Before invoked, the orchestrator parses the complete input document and splits the `items` array into chunks of `chunk_items` items for a fixed size, except for the final chunk, which may contain fewer items. The workload then applies the JSON transformation, SHA-256/XOR processing, and file-output stage to each chunk independently.

This variant is included to evaluate how chunked execution changes the behavior of the composed pipeline. Compared with the non-streaming pipeline, the streaming variant avoids processing one complete transformed JSON document as a single buffer. Instead, each chunk produces a smaller transformed buffer that is cryptographically processed and written before the next chunk is handled. This can change the peak memory usage, cache behavior, garbage collection behavior, and the interaction with file-output buffering.

Algorithm 5 WORKLOAD_PIPELINE_E2E_STREAM

```
1: sink
2: function WORKLOAD_PIPELINE_E2E_STREAM(chunks, chunk_count, key_hmac, key_xor, ciphertext, decrypted, out_hmac, dst)
3:     bytes_since_sync ← 0
4:     for i ← 0 to chunk_count − 1 do
                                  ▷ Stage 1: parse, transform, and serialize one JSON chunk
5:         transformed_json ← ∅
6:         transformed_len ← 0
7:         WORKLOAD_JSON_PARSE(chunks[i].json, transformed_json, transformed_len)
                                  ▷ Stage 2: encrypt, hash, and decrypt transformed chunk
8:         WORKLOAD_SHA256(transformed_json, transformed_len, key_hmac, key_xor, ciphertext, decrypted, out_hmac)
                                  ▷ Stage 3: write decrypted chunk using amplified output
9:         for rep ← 1 to WRITEAMPLIFICATION do
10:            WRITEALL(dst, decrypted, transformed_len)
11:            bytes_since_sync ← bytes_since_sync + transformed_len
12:            if bytes_since_sync ≥ SYNCINTERVAL then
13:                DATASYNC(dst)
14:                bytes_since_sync ← 0
15:            end if
16:        end for
                                  ▷ Prevent compiler or runtime from discarding the result
17:        sink ← sink ⊕ (transformed_len mod 256)
18:        sink ← sink ⊕ ByteAt(decrypted, 0)
19:        sink ← sink ⊕ out_hmac[0]
                                  ▷ Release transformed JSON buffer if required
20:        DELETEIFREQUIRED(transformed_json)
21:    end for
                                  ▷ Flush remaining written data
22:    DATASYNC(dst)
23:    return 1
24: end function
```

Algorithm 5 shows the composed pipeline. For each JSON chunk, the workload applies the same three stages as the non-streaming pipeline: JSON parse-transform-serialize, SHA-256/XOR processing, and amplified file output. The output file is written incrementally as the chunks are processed. A data synchronization operation is performed periodically after a fixed amount of written data and once more after all chunks have been processed.

Input, output, correctness, and scaling The input, output, correctness condition, and primary scaling variable are the same as for the non-streaming-composed pipeline in Section 4.5.4. The main difference is that the non-streaming pipeline processes the complete transformed JSON document as a single intermediate buffer, whereas the streaming pipeline processes one transformed chunk at a time. The streaming variant therefore in-

troduces one additional parameter, `chunk_items`, which controls the number of records per chunk. This parameter is kept fixed to `1024` during the experiments.

4.6 Dataset generation

All input files for each workload are generated before running the benchmarks. These input files are generated using deterministic procedures and stored as static files. To ensure cross-language compatibility, identical workloads are used in each implementation. The benchmark suite uses two types of generated input data. The JSON workload and pipeline workloads use a Yelp-derived JSON document, while the SHA-256/XOR and file-output workloads process binary buffers. The generators use powers-of-two input sizes between the configured minimum and maximum size.

4.6.1 Yelp-derived JSON dataset

The JSON input data are derived from the Yelp Open Dataset [37]. The JSON workload uses records derived from the Yelp Academic Dataset business file. The original dataset is distributed as newline-delimited JSON, where each line represents one business record. A business record contains the following fields; `business_id`, `name`, `city`, `state`, `stars`, `review_count`, `is_open`, `attributes`, `categories`, and `hours`. In this thesis, Yelp data are used as a reproducible source of semi-structured JSON records. Yelp-style JSON data have also been used in previous work on JSON analytics and parsing, for example, in Mison, which uses Yelp-derived examples when studying fast JSON parsing for data analytics [38], and Cui evaluates the Yelp dataset in the context of data analysis tasks. [39]

Each Yelp business record is mapped to a smaller benchmark item schema. This schema contains the fields required by the JSON workload, together with a small number of additional scalar and string fields. During dataset generation, each selected Yelp business record is mapped to the common schema expected by the benchmark workload.

Benchmark field	Source field / rule	Purpose
`id`	Generated item index	Identifier modified by the workload
`name`	`name`	String field transformed to uppercase
`active`	`is_open == 1`	Boolean field negated by the workload
`score`	`review_count`	Preserved numeric field
`category`	First entry in `categories`	Preserved string field

Table 3: Mapping from Yelp business records to benchmark JSON items

4.6.2 SHA256-XOR input generation

The SHA-256/XOR workload uses deterministic binary input data. For each size S, the generator creates a directory `datasets/sha256/`S containing three files: `buffer.bin`, `key_hmac.bin`, and `key_xor.bin`. The `buffer.bin` file contains the input bytes, while the two key files contain the fixed HMAC and XOR keys. Xorshift is used as a lightweight and deterministic pseudo-random number generator due to its simplicity and reproducibility in benchmarking contexts.

The bytes are generated using the deterministic xorshift procedure shown in Algorithm 6. A benchmark-specific seed is derived from the global seed and benchmark name, and the target size is added to this seed so that different input sizes produce different byte sequences. The HMAC key has a fixed length of 32 bytes, the XOR key has a fixed length of 16 bytes, and the input buffer scales with the target size S.

Algorithm 6 Generate deterministic xorshift bytes

function XorShiftBytes(length, seed)
    $x \leftarrow seed \,\&\, 0xffffffff$
    $out \leftarrow$ empty byte array of size $length$
    for $i \leftarrow 0$ to $length - 1$ step 4 **do**
        $x \leftarrow x \oplus ((x \ll 13) \,\&\, 0xffffffff)$
        $x \leftarrow x \oplus (x \gg 17)$
        $x \leftarrow x \oplus ((x \ll 5) \,\&\, 0xffffffff)$
        append up to four little-endian bytes of x to out
    end for
    return out
end function

4.6.3 File copy input generation

The file is generated using the xorshift procedure from Algorithm 6. For efficiency, the generator first creates a deterministic block of at most 4 MiB and repeats this block until the requested file size is reached, as shown in Algorithm 7. For inputs smaller than 4 MiB, the block size is reduced to the input size.

Algorithm 7 Generate file-copy input

function GenerateFileCopyInput(path, size, seed)
    $block\_size \leftarrow \min(4\,\text{MiB}, size)$
    $block \leftarrow$ XorShiftBytes($block\_size, seed$)
    while $size > 0$ **do**
        $n \leftarrow \min(block\_size, size)$
        write first n bytes of $block$ to $path$
        $size \leftarrow size - n$
    end while
end function

4.6.4 Pipeline input generation

The composed pipeline benchmarks do not generate independent input data. Instead, they reuse the generated datasets from the JSON and SHA-256/XOR workloads. For a pipeline input size S, the JSON input is read from the corresponding JSON dataset directory, while the HMAC and XOR keys are read from the corresponding SHA-256 / XOR dataset directory. The output file is written to the corresponding file-copy output directory.

4.7 Scaling strategy

All selected workloads are designed so that the amount of work scales linearly with the input size S, meaning that the expected algorithmic complexity of each workload is $O(S)$. For the JSON workload, increasing S increases the number of records in the `items` array. For the SHA-256/XOR workload, increasing S increases the number of bytes processed. For the file-output workload, increasing S increases the number of bytes written. The composed pipelines inherit this scaling behavior because their stages process the generated input sequentially. These complexities can be verified by inspecting table 9. Therefore, the expected algorithmic complexity of the workload functions is linear in the input size, while the measured runtime, energy, and memory behavior may still differ between languages due to runtime systems, libraries, allocation behavior, I/O effects, and hardware-level behavior.

Input sizes are generated in powers of 2 between the configured minimum and maximum size. For each size, all selected languages and implementation variants use the same generated input files. This ensures that differences in runtime, energy consumption, and memory usage are not caused by differences in input data.

Input scaling is included because very small inputs can be dominated by fixed costs such as process startup, runtime initialization, package loading, and JIT compilation. In such cases, the measured runtime and energy consumption may reflect the execution environment more than the workload itself. Larger inputs may increase the relative load of the workload and can expose effects such as cache behavior, DRAM traffic, allocation pressure, garbage collection, and file-system buffering. Previous workload-characterization studies also show that realistic benchmark suites can include workloads with substantial memory footprints and memory traffic [40].

4.8 Implementation methodology and language-specific decisions

As discussed in Section 2.7, cross-language benchmarking is affected by the *human factor*: the benchmark author must make choices about algorithms, libraries, coding style, and optimization effort. These choices can influence measured runtime, energy consumption, and memory usage.

To reduce the influence of the human factor, all implementations follow the same workload structure and input-output behavior. Each language implementation receives the same generated input data, applies the same high-level algorithmic steps, and produces verifiable output. The goal is not to produce the fastest possible implementation in every language, but to produce comparable implementations that follow the same benchmark design.

The implementations are written in a direct imperative style using explicit loops and named intermediate values where possible. Some languages provide compact constructs such as list comprehensions, maps, filters, iterator pipelines, or high-level convenience functions. Although these constructs may be idiomatic in some contexts, using different expression styles across languages can make it harder to determine whether observed differences are caused by the language runtime, the library implementation, or a different algorithmic structure. Therefore, the benchmark code favors a uniform control-flow structure across languages, especially in the workload functions.

4.8.1 Library selection and JSON parser variants

The benchmark suite uses standard libraries where possible and documented external libraries where the standard library does not provide the required functionality or where a library variant is part of the experiment. For file-output workloads, the implementations rely primarily on operating-system file operations. For cryptographic processing, library support is used for HMAC-SHA256, while the XOR step is implemented directly. For JSON parsing, multiple library variants are included because JSON performance can strongly depend on the parser implementation.

Consequently, the results should be interpreted as comparisons of the representative implementations used in this thesis, not as universal rankings of the programming languages. The goal is to evaluate how equivalent workload implementations behave across languages under a controlled and reproducible methodology, while acknowledging that different library choices or optimization strategies could lead to different absolute results.

For each language, at least two JSON libraries were used for the `json_parse` workload to show the difference in performance when using different libraries.

Language	Variant	Description
C	`cjson`	C JSON parser used as the default C implementation
C	`yyjson`	Alternative C JSON parser variant
Go	`encoding_json`	Go standard-library JSON implementation
Go	`goccy_go_json`	Alternative Go JSON implementation
Julia	`json`	`JSON.jl` implementation
Julia	`json3`	`JSON3.jl` implementation
Python	`json`	CPython standard-library JSON implementation
Python	`json_no_c_acc`	CPython standard-library JSON, C accelerators disabled
Python	`orjson`	Alternative Python JSON implementation

Table 4: JSON parser variants evaluated in the benchmark suite

The standard-library variants are used as baseline implementations, where available. Additional variants are included to show how much the result can change when a different parser is selected within the same language. The Python `json_no_c_acc` variant is included to separate the behavior of the CPython standard-library interface from the effect of its C acceleration paths. This variant modifies the internals of the CPython `json` module to disable the C scanner and encoder acceleration paths and rebuilds the default decoder and encoder. These modifications are based on CPython implementation details, rather than supported configuration parameters, because the standard-library `json` interface does not provide a public option for disabling these acceleration paths.

4.8.2 SHA-256/XOR implementation behavior

Although the SHA-256/XOR implementations follow the same high-level algorithm, low-level execution can still differ between languages. As discussed in Section 2.9, loops may be affected by bounds checks and vectorization. This is especially relevant for the XOR step, which is implemented directly in each language.

For Julia, bounds checks in the XOR loops were disabled using `@inbounds`. C does not perform runtime bounds checks for array and pointer accesses, while Go performs bounds checks unless the compiler can prove that they are unnecessary. For the Go implementation, compiler diagnostics showed that some bounds checks remain in the SHA-256/XOR workload. Python also performs index and bound checks for sequence accesses, but in this workload, the larger source of overhead is that the XOR loop is executed at interpreter level using standard `bytes` and `bytearray` indexing.

Languages such as C, Go, and Julia can vectorize loops, but this is not guaranteed. We checked this for the code compiled on DAS-5. For C, the loops were not vectorized. C can likely not vectorize the SHA256/XOR function due to the line: `key_xor[i % key_xor_len]`, where the compiler does not know the length of `key_xor_len`, thus cannot prove the loop is safe. Although the key length is fixed for this benchmark, it is represented as a variable in the function. Together with the use of dynamic pointers and lengths, this makes the loop harder for the compiler to reason about.

Unlike C and Go, Julia's XOR loop appears to have been vectorized. Inspection of the generated code showed vector instructions such as `vpxor`, `vpbroadcastb`, and `vmovdqu`, indicating that LLVM generated vectorized code for the loop. Some bounds-error paths may still be present in the compiled function, but these are not expected to correspond to per-access checks inside the loop annotated with `@inbounds`.

5 Experiments

5.1 Benchmark framework and execution workflow

The benchmark suite is implemented as a script-driven framework that separates workload logic, language-specific benchmark orchestrators, dataset generation, and measurement orchestration. This separation is important for reproducibility, as the same generated datasets are reused across languages, while the same orchestration script controls the order of execution, repetitions, runtime measurement, energy measurement, and memory monitoring. Each benchmark is launched by `run_benchmarks.py` as an isolated operating-system process, referred to throughout this thesis as a benchmark subprocess. Figure 1 summarizes the execution workflow used on DAS-5.

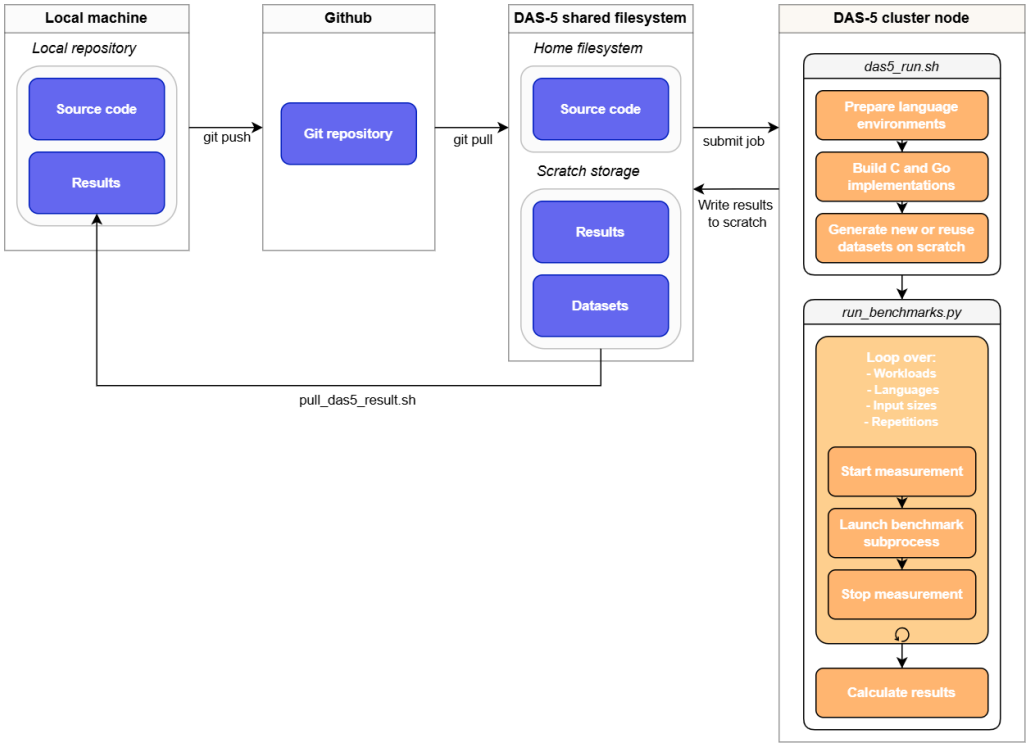


Figure 1: Benchmark execution workflow on DAS-5. The source repository [41] is maintained on the local machine, pushed to a cloud based Git repository, and pulled to the home filesystem of DAS-5 using GIT version control. On DAS-5 the `das5_run.sh` run script is queued to execute on a cluster node using `prun`. On the cluster node, `das5_run.sh` prepares the environments, builds compiled implementations, generates or reuses datasets on scratch storage, and starts `run_benchmarks.py`. The orchestration script iterates over benchmark configurations, launches benchmark subprocesses, measures runtime, RAPL energy, and peak RSS, and writes the results to scratch storage on DAS-5. The resulting files are manually copied back to the local machine using `pull_das5_result.sh`.

Table 5 lists the main components of the repository used by the benchmark suite.

Directory	Purpose
`benchmarks/`*benchmark*`/`*language*	Language-specific benchmark orchestrators
`benchmarks/workloads/`	Workload implementations per language
`scripts/generate_inputs.py`	Deterministic dataset generation
`scripts/run_benchmarks.py`	Benchmark orchestration and measurement
`scripts/analyze_results.py`	Normalized score computation
`scripts/das5_run.sh`	Run script specific to DAS-5
`scripts/pull_das5_result.sh`	Script to retrieve results stored from DAS-5
`datasets/`	Generated input datasets
`results/`	Raw and processed benchmark results

Table 5: Repository components used by the benchmark suite

5.2 Hardware configuration and software environment

The experiments were executed on DAS-5, the fifth Distributed ASCI Supercomputer. DAS-5 is used for experimental computer science research, including parallel and distributed systems research [42, 43]. The VU DAS-5 cluster consists primarily of dual-socket Intel Xeon E5-2630v3 compute nodes.

The run described in this thesis was executed on `node065`. The node runs Enterprise Linux 8 with the Linux kernel version `4.18.0-553.16.1.el8_10.x86_64`. The processor is an Intel Xeon E5-2630 v3 with a base frequency of 2.40 GHz and a reported maximum frequency of 3.20 GHz. The node contains two sockets, each with eight physical cores and two hardware threads per core, resulting in 32 logical CPUs. The system exposes two NUMA nodes and 62 GiB of main memory.

The available RAPL domains during the run were the CPU package and the DRAM domains for both CPU sockets. The benchmark metadata and the environment description were stored alongside the result files to support reproducibility.

Component	Configuration
Platform	DAS-5 compute node
Node used	`node065`
Operating system	Enterprise Linux 8
Kernel	`4.18.0-553.16.1.el8_10.x86_64`
CPU	Intel Xeon E5-2630 v3
Base frequency	2.40 GHz
Reported maximum frequency	3.20 GHz
Sockets	2
Physical cores	16 total, 8 per socket
Hardware threads	32 total, 2 per core
NUMA nodes	2
Main memory	62 GiB
L1 cache	32 KiB data, 32 KiB instruction
L2 cache	256 KiB
L3 cache	20 MiB

Table 6: Hardware and operating-system configuration of the DAS-5 node used for the experiments

The software environment is summarized in Table 7. The C and Go implementations were compiled prior to execution, while the Python and Julia implementations were executed using their respective runtime environments. The C workloads were compiled using the flag -03 for optimization. The benchmark suite also includes multiple JSON library variants for the JSON parsing workload.

Component	Version / configuration
GCC/G++	12.4.0
CMake	3.24.2
Python	3.11.9
Go	1.25.0
Julia	1.12.5
Git	2.43.5
OpenSSL	1.1.1k
Go JSON libraries	`encoding/json`, `goccy/go-json` v0.10.6
Julia JSON libraries	`JSON.jl` v1.5.0/v1.5.1, `JSON3.jl` v1.14.3
Julia SHA library	`SHA.jl` v0.7.0
Loaded compiler module	`gnu12/12.4.0`
Benchmark Git commit	`a19108899abd6cf1a07f3111e5a7097746682426`

Table 7: Software environment used for benchmark execution

The details of the software environment were written to a `native_environment.txt` file during the run, while benchmark metadata such as timestamp, hostname, RAPL availability, RAPL domains, language versions, and Git commit were stored in a separate metadata file.

5.3 Energy measurement method

Energy consumption is measured using Intel RAPL. It exposes energy counters through the Linux powercap interface and provides access to processor energy domains such as the CPU package and DRAM energy. Previous work has used RAPL extensively for software energy measurement, and validation studies show that RAPL provides a practical interface to measure processor and memory energy, while also noting that RAPL values are model-based estimates rather than direct wall-socket measurements [44, 45]. Therefore, the measurements in this thesis should be interpreted as CPU package and DRAM energy-to-solution, not as total machine energy consumption.

During benchmark execution, the orchestration script `run_benchmarks.py` first discovers all available RAPL zones under `/sys/class/powercap`. Each discovered zone contains an `energy_uj` file, which stores the cumulative energy counter in microjoules, and a `max_energy_range_uj` file, which is used to handle counter wraparound. For the DAS-5 node used in the experiments, the available RAPL domains were `package-0`, `package-1`, and one DRAM domain per socket. The metadata file stored with the results records these domains for each benchmark run.

For each benchmark run, the orchestration script reads all RAPL counters before launching the language-specific benchmark subprocess, and again after the subprocess terminates. The difference between the start and end counter values gives the energy consumed during execution. If a counter wraps around, that is, it reaches its maximum counter range and continues counting from zero, the delta is corrected using the maximum counter range. Energy values are converted from microjoules to joules before being written to the result file.

The package energy reported in the results is the sum of all RAPL domains whose names start with `package`, while the DRAM energy is the sum of all RAPL domains named `dram`. This means that the reported package energy includes both CPU sockets and the reported DRAM energy includes the DRAM domains exposed for both sockets. The measurement is performed around the complete benchmark subprocess, including language runtime startup, library loading, workload execution, and output cleanup performed inside the subprocess.

5.4 Memory and resource monitoring

Memory usage is monitored by the benchmark orchestration script `run_benchmarks.py` using the Python `psutil` library. For each benchmark run, the script launches the language-specific benchmark implementation as a subprocess and records the resident set size (RSS) of that process and its child processes. RSS represents the amount of physical memory currently resident for the process. The maximum sampled RSS value during the benchmark run is stored as the maximum memory usage.

The memory sampler runs in a separate thread while the benchmark subprocess is active. At a fixed sampling interval, by default set to 10 ms which can be modified by adding the flag `--memory-interval-ms`, the sampler reads the RSS of the benchmark process and recursively includes child processes. This is relevant for language runtimes or tools that may spawn helper processes. At the end of the run, the highest observed RSS value is written to the results file as `rss_peak_bytes`, together with the start and end RSS values and the number of samples collected.

5.5 Main performance metrics

Table 8 summarizes the main metrics collected for each benchmark run and used during the evaluation.

Metric	Unit	Meaning
Wall-clock time	seconds	Duration of the complete benchmark subprocess
CPU package energy	joules	RAPL package energy across both CPU sockets
DRAM energy	joules	RAPL DRAM energy across both sockets
Peak RSS	bytes	Maximum sampled resident memory of the subprocess
Energy-to-solution	joules	Energy required to complete one benchmark run

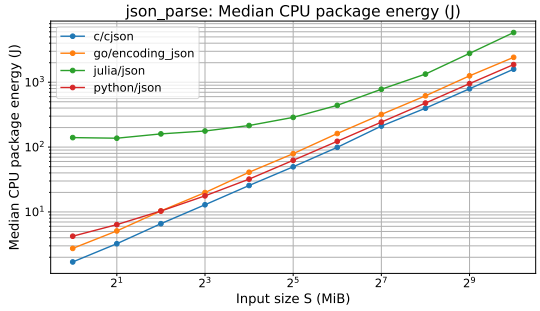
Table 8: Main metrics collected during benchmark execution

6 Results

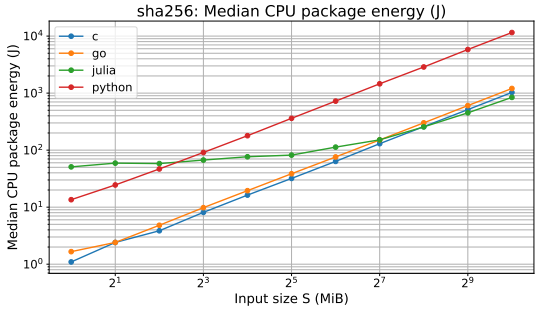
In this chapter, the results gathered by running the experiments on DAS-5 will be shown. We will compare the **energy results**, **runtime**, **peak memory**, **cross language**, and different **json parse variants** for each workload.

6.1 Energy results per workload

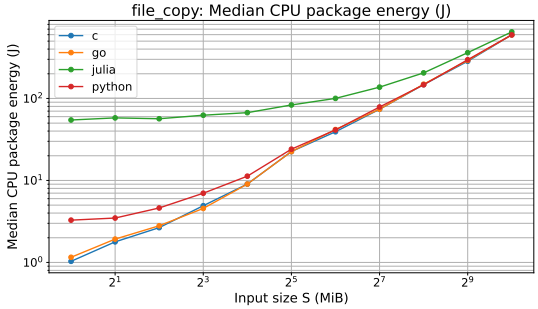
Figure 2 shows the median CPU package energy for the isolated workloads. The results are shown as a function of the input size S, using one baseline implementation per language. For the JSON workload, the baseline variants are `cjson` for C, `encoding_json` for Go, `JSON.jl` for Julia, and CPython's standard-library `json` for Python. Additional JSON parser variants are analyzed separately in Section 6.5.



(a) JSON parse



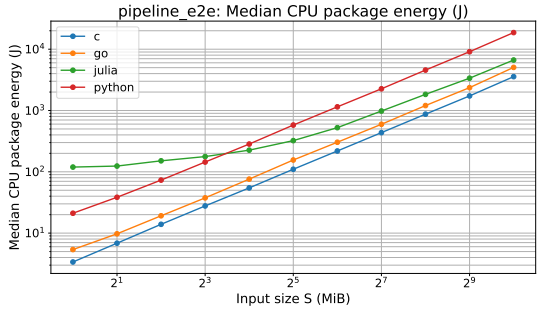
(b) SHA-256/XOR



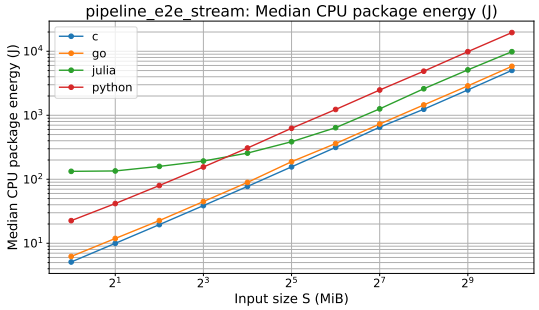
(c) File copy

Figure 2: CPU package energy for isolated workloads.

Across the isolated workloads, the package energy generally increases with input size, as expected from the linear scaling of the workload definitions. The magnitude of the differences between languages depends on the workload. For SHA256/XOR and JSON parse workloads, language runtime and library behavior have a larger effect, while for the file-copy workload, the differences most likely become smaller at larger input sizes because execution mainly consists of operating-system file operations.



(a) Non-streaming pipeline

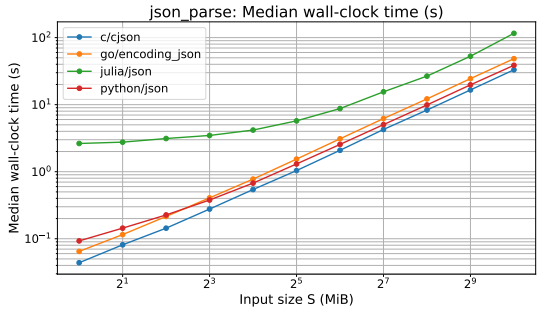


(b) Streaming pipeline

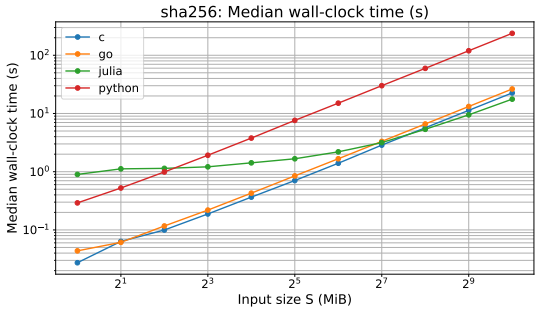
Figure 3: CPU package energy for composed workloads.

Figure 3 shows the package energy for the composed pipeline workloads. The composed workloads combine JSON transformation, SHA-256/XOR processing, and file output. As a result, their energy behavior reflects the combined cost of multiple stages rather than the behavior of a single isolated workload. The streaming pipeline is shown separately because it changes the way the JSON input is processed and written, which may affect both runtime and memory behavior.

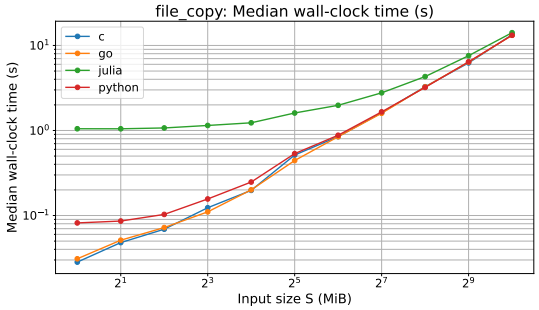
6.2 Runtime results per workload



(a) JSON parse



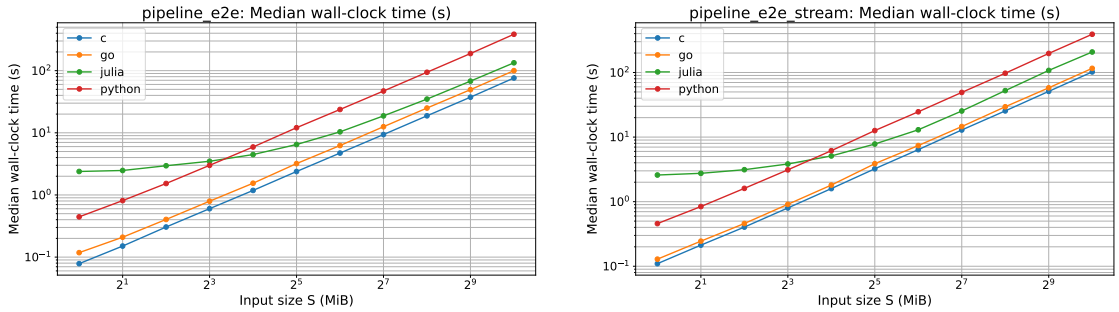
(b) SHA-256/XOR



(c) File copy

Figure 4: Runtime for isolated workloads.

Figure 4 shows the median wall-clock runtime for the isolated workloads. Runtime follows the same general scaling direction as the package energy, but the relationship is not identical because energy also depends on the power draw during execution.



(a) Non-streaming pipeline

(b) Streaming pipeline

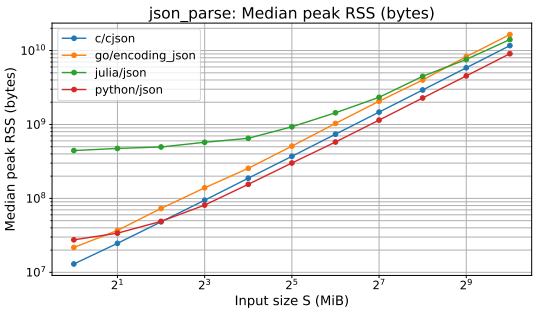
Figure 5: Runtime for composed workloads.

Figure 5 shows the runtime of the composed pipeline workloads. These results are important because the composed workloads evaluate whether the behavior observed for isolated stages remains visible when the stages are executed together.

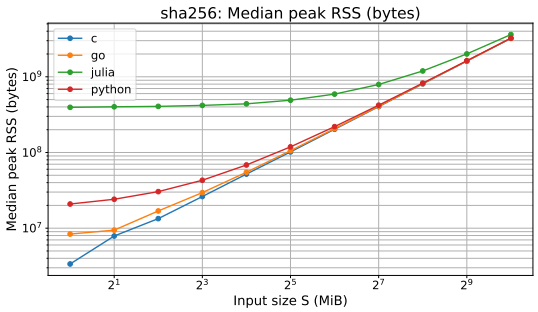
6.3 Peak memory usage

Figure 6 shows the median peak resident set size for the isolated workloads. Peak RSS includes the benchmark subprocess and, therefore, contains both workload memory and fixed runtime overhead. This is especially visible in simple workloads, such as file copy, where the actual workload consists mainly of loading an input buffer and writing it to an output file. In such cases, fixed runtime costs can be a large part of the measured memory footprint.

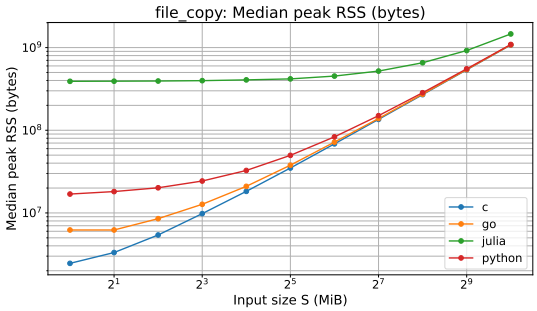
The JSON workload shows memory behavior associated with the parsing of structured input, the construction of language-level data structures, the transformation of fields, and the serialization of the result. The SHA-256/XOR workload mainly processes byte buffers and therefore has a simpler memory profile. The file-copy workload is included because it shows how runtime overhead can dominate memory measurements when the workload itself performs little allocation beyond the input buffer.



(a) JSON parse



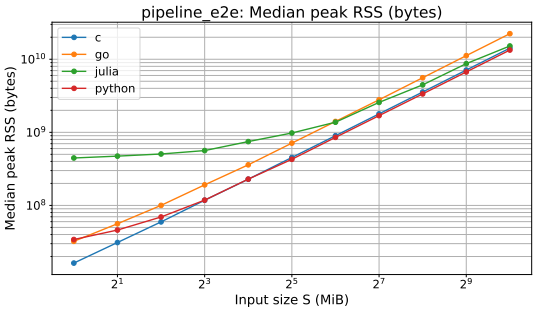
(b) SHA-256/XOR



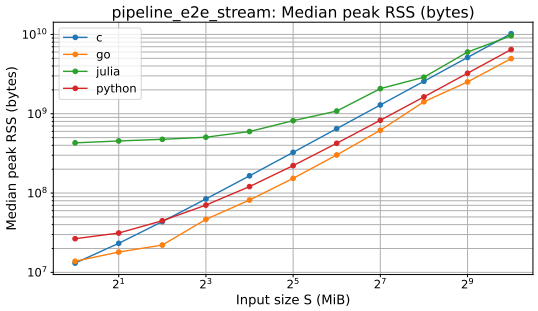
(c) File copy

Figure 6: Peak RSS for isolated workloads.

Figure 7 shows the peak RSS for the composed pipeline workloads. The non-streaming pipeline materializes the transformed JSON before passing it to the cryptographic and file-output stages. The streaming pipeline processes smaller chunks, which is expected to reduce peak memory usage when intermediate data dominate the memory footprint. The measured results therefore indicate whether streaming changes the memory behavior compared with processing the complete input as one pipeline unit.



(a) Non-streaming pipeline



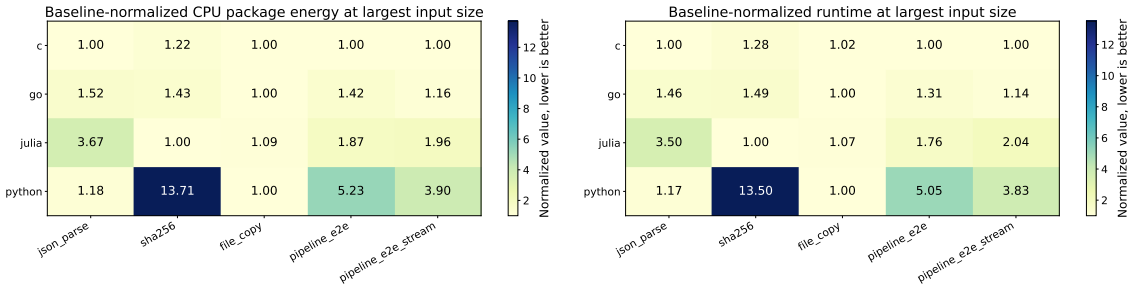
(b) Streaming pipeline

Figure 7: Peak RSS for composed workloads.

6.4 Cross-language comparison

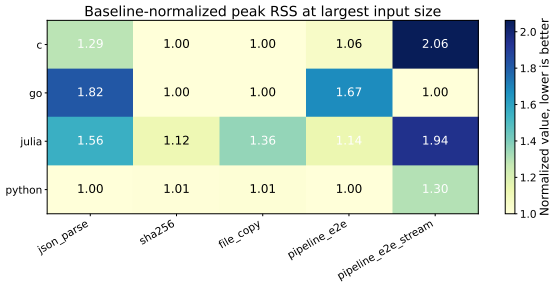
To compare workloads with different absolute scales, the results are normalized relative to the best baseline implementation for the same workload, input size, and metric. A normalized value of 1.0 therefore indicates the best observed baseline result for that configuration, while larger values indicate proportional overhead. For example, a value of 2.0 means that the implementation required twice as much energy, runtime, or memory as the best baseline implementation for the same workload and input size.

Figure 8a shows the baseline-normalized CPU package energy at the largest input size, in this case 1GiB. This figure summarizes the relative energy behavior across workloads and languages. The heatmap should be interpreted as a workload-specific comparison: values are normalized within each workload and should not be compared as absolute joule values across workloads.



(a) CPU package energy

(b) Runtime



(c) Peak RSS

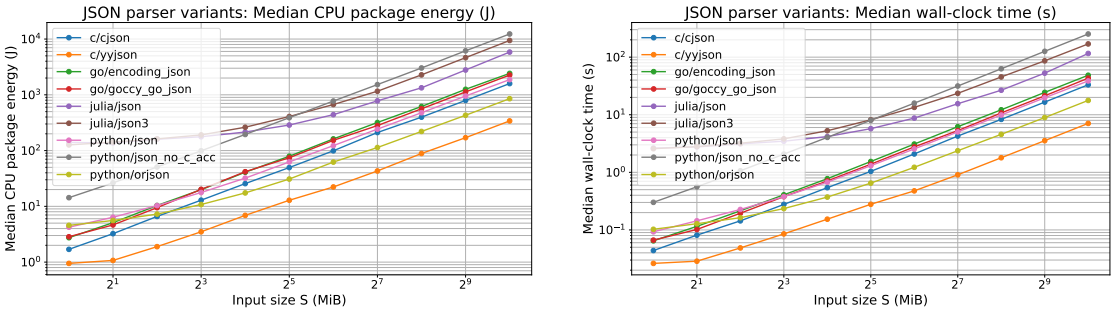
Figure 8: Baseline-normalized comparison at the largest input size. Lower values are better.

6.5 JSON parser variants

The cross-language comparison in the previous section uses one baseline JSON implementation per language. However, JSON parsing performance can depend strongly on the selected parser library. Therefore, the `json_parse` workload was also executed with additional parser variants.

Figure 9a shows the CPU package energy for all variants of the JSON parser. Figure 9b shows the corresponding runtime results. These plots illustrate how changing the parser implementation within the same language can change the measured result, even though

the surrounding workload structure and input data remain the same.

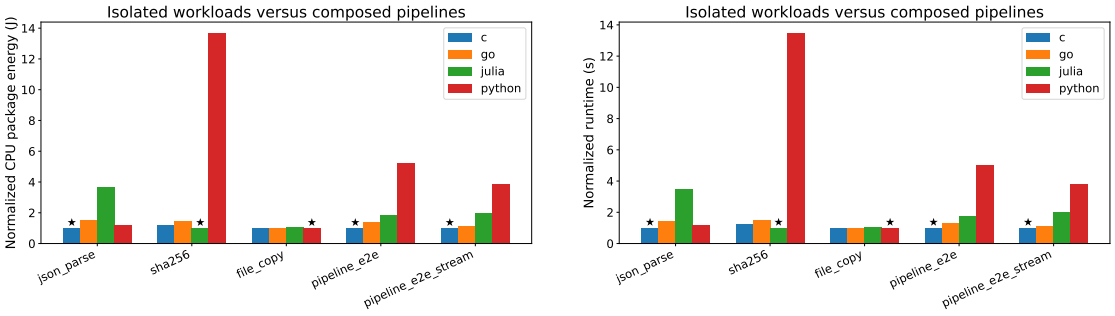


(a) CPU package energy (b) Runtime

Figure 9: Energy and runtime results for JSON parser variants.

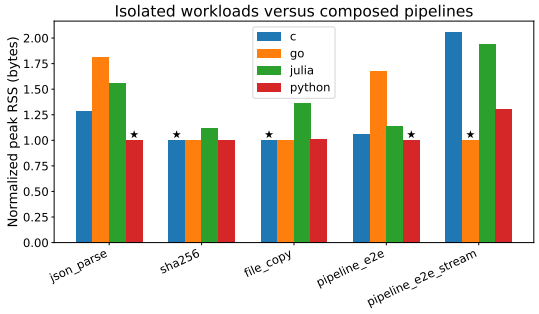
6.6 Isolated workloads and composed pipelines

Figure 10 compares isolated workloads with composed pipeline workloads at the largest input size, in this case 1GiB. The goal is not to compare the isolated workloads against the pipeline workloads directly, but rather to assess whether the relative ordering of languages observed in isolation is preserved when these workloads are composed into an end-to-end workload. The starred bar indicates the language with the lowest normalized score per workload, where lower is better.



(a) CPU package energy (b) Runtime

Figure 10: Baseline-normalized runtime, CPU package energy, and peak RSS for isolated and composed workloads at the largest input size. The starred bar indicates the language with the lowest normalized score per workload, where lower is better. (Continued on next page.)



(c) Peak RSS

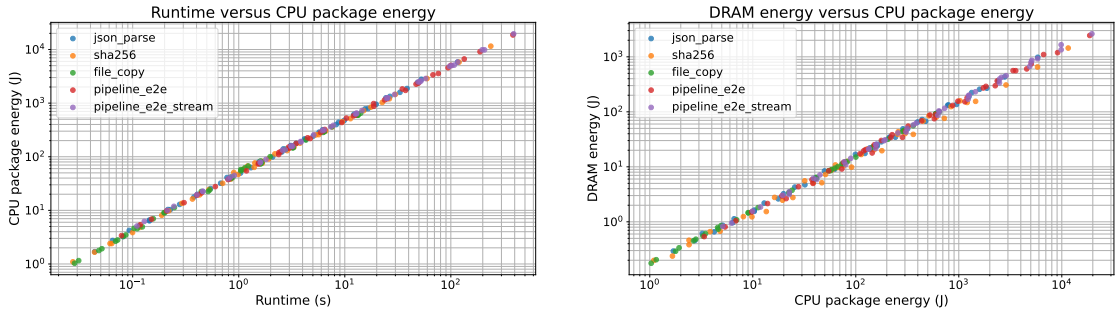
Figure 10: (Continued) Baseline-normalized runtime, CPU package energy, and peak RSS for isolated and composed workloads at the largest input size.

6.7 Correlation Analysis

Figure 11 presents the correlation between energy and runtime, as well as between the CPU package energy and the DRAM energy, aggregated over all workloads. The relationship between power and time in figure 11a, as discussed in Section 2.8, appears approximately linear, with a scaling coefficient close to 1. This indicates that energy increases proportionally with runtime across the evaluated workloads.

A similar trend is observed between the CPU package and DRAM energy in figure 11b. The data points align largely along a single linear trend, with a small number of outliers forming a secondary but still linear pattern. This suggests a strong proportional relationship between the two energy components, with deviations that remain structured rather than random.

These observations can also be seen when zooming in on an isolated workload such as `json_parse` in the appendix, figure 13.



(a) Energy & Runtime

(b) PKG & Dram

Figure 11: Energy & Runtime correlation results on the left and PKG & Dram correlation on the right, plotted for all workloads.

7 Discussion

This chapter interprets the results of Chapter 6 in relation to the research questions. The goal of this thesis was not just to rank programming languages in isolation, but to evaluate how realistic in the sense defined in Section 2.3 implementations behave across controlled data-processing workloads.

7.1 Interpretation of results

RQ1: Workload design criteria. The first research question asked what criteria are required to design CPU-, memory-, and I/O-oriented workloads that are comparable and reproducible across programming languages. The results support five criteria: deterministic input generation, algorithmic equivalence, verifiable output correctness, scalable input sizes, and documented implementation choices. The necessity of controlling implementation choices is demonstrated by the JSON parser variant results (Figure 9). Within Python alone, replacing the standard json module with a pure-Python variant that disables the C accelerator increases runtime by 3.2× and energy consumption by 3.4× at 1 MiB, growing to 6.5× and 6.6× respectively at 1 GiB, despite identical input, output, and high-level algorithm. This shows that a shared workload description does not guarantee a comparable measurement: the specific library and its internal implementation must be documented and controlled as an explicit design criterion. The necessity of scalable input sizes is demonstrated by Julia's JIT compilation behavior. For the sha256 workload at 1 MiB, Julia's runtime is 32.9× that of C, and its peak memory usage is 117× higher (376 MB versus 3 MB), reflecting fixed JIT compilation overhead. At 1 GiB, Julia is faster than C by 22%. A benchmark restricted to small input sizes would produce a systematically misleading characterization of Julia's efficiency. The 11 $\log^2$-scaled sizes from 1 MiB to 1 GiB make this scaling behavior visible and allow the comparison to operate across regimes where JIT costs are both dominant and amortized. Together, these criteria enabled the cross-language comparisons reported in RQ2 and supported the consistency analysis in RQ3.

RQ2: Cross-language differences across isolated and composed workloads. The second research question asked what differences can be observed between programming languages in runtime, CPU package energy, DRAM energy, and peak memory usage across isolated workload classes and composed data-processing pipelines. Across all workloads, a correlation was observed between runtime and CPU package energy as mentioned in Chapter 6.7, indicating that languages with longer execution times consumed more energy. Similarly, the DRAM energy scaled with the energy of the CPU package, suggesting that memory energy mostly follows the overall activity of the system. Nevertheless, the ranking of languages varied between workload classes. The main observation is that there is no single language that performs best for all workloads and metrics. The relative behavior strongly depends on the workload class, the selected libraries, and the runtime characteristics of each language.

For JSON parsing, the results show that parser implementation has a large effect. The baseline JSON comparison in Figures 2 and 4 shows the behavior of one selected parser implementation per language, while Figure 9 shows that changing the JSON parser within

the same language can also change runtime and CPU package energy. This means that the JSON results should be interpreted as comparisons of selected parser implementations, not as pure language-level results.

The SHA256/XOR workload shows that equivalent pseudocode can still lead to different low-level executions. In Figures 4 and 2, Julia performs strongly for SHA256/XOR compared to its behavior in simpler workloads such as file copy. This can be related to the implementation behavior described in Section 4.8.2. The C implementation was compiled with optimization enabled, but the GCC diagnostics reported that the XOR encryption and decryption loops were not vectorized. The Go compiler diagnostics showed that some bound checks remained in the SHA256/XOR workload. In contrast, Julia's generated native code contained vector instructions such as `vpxor`, `vpbroadcastb`, and `vmovdqu`, indicating that LLVM generated vectorized code for the XOR loops. Python performs poorly in this workload because the XOR encryption and decryption stages are executed as Python-level byte loops, even though the HMAC-SHA256 operation itself uses library code.

For file copy, the differences between C, Go and Python become smaller with larger input sizes, as shown in Figures 2 and 4. This workload mainly performs operating-system file operations and explicit synchronization, so language-level computation becomes less dominant as the input size grows. Julia remains more expensive in peak memory, as shown in Figure 6, which is consistent with the larger fixed runtime and JIT footprint being visible in a simple I/O-oriented workload.

Memory results also show that peak RSS does not always follow runtime or energy behavior. Figures 6 and 7 show that an implementation can have a low maximum memory usage, while still requiring more runtime or CPU package energy. This is visible for Python in several workloads. As discussed in Section 2.5, CPython uses reference counting together with a cyclic garbage collector, but the low peak RSS observed here might not be a result of only garbage collector efficiency. A possible explanation is that peak RSS measures how much memory the process keeps in RAM, while runtime and energy are also affected by the amount of work performed by the interpreter. This is especially relevant for byte-wise loops, where Python may use relatively little memory but still spend substantial time executing many small operations.

The **composed pipeline** workloads show that the isolated workload behavior does not always transfer directly to end-to-end execution. Figures 3, 5, and 7 show the behavior of the non-streaming and streaming pipelines. In isolated workloads, Julia performs best for SHA256/XOR and Python is competitive for file copy at larger input sizes. However, in both composed pipeline variants, C is the strongest baseline implementation for runtime and CPU package energy. The memory behavior also changes between pipeline variants: Python has the lowest peak RSS in the non-streaming pipeline, while Go has the lowest peak RSS in the streaming pipeline for all inputs, whereas Go had the highest peak in the non-streaming variant. This shows that composed workloads introduce interactions between parsing, byte-buffer processing, file output, allocation behavior, and runtime systems that are not captured by isolated workloads alone. Figure 10 further summarizes this comparison in the largest input size.

RQ3: Input size and repeated executions. The third research question asked what effects input size and repeated execution have on the stability and variability of runtime, energy, and memory measurements. The input size plots in Figures 2, 3, 4, and 5 show that the runtime and CPU package energy increase consistently with input size. The log-log scaling plots make this relationship explicit: the runtime scaling exponents in Table 9 are close to 1.0 across all benchmarks and languages at larger input sizes, confirming linear $O(S)$ scaling.

However, the increase is not perfectly proportional for all workloads and languages for each input size. The measured values include process startup, runtime initialization, JIT compilation, where applicable, library behavior, allocation behavior, garbage collection, file-system behavior, and hardware-level effects. No warmup run was performed prior to measurement, meaning that Julia's JIT compilation cost is included in all reported values. This reflects a cold-start execution model, which is representative of data-processing pipelines that are invoked once per input rather than running as persistent services. These effects are especially visible for small input sizes, where fixed costs can form a larger part of the total measurement.

Repeated executions were handled by running multiple repetitions and reporting the median of the runs. This reduces the influence of occasional outliers and gives a stable summary of the main result figures. However, this thesis does not perform a detailed statistical analysis of the variance from run-to-run. This is particularly relevant for I/O-oriented workloads, where file-system state and synchronization behavior can vary between repetitions. Therefore, the results support the main scaling trends, while a more detailed variance analysis is left for future work.

Main RQ The results show that substantial differences exist between programming languages in runtime, CPU package energy, DRAM energy, and peak memory usage, but that these differences depend on workload characteristics and workload composition. No single language achieved the best results for all workloads and metrics. For the SHA256/XOR workload, Julia achieved the lowest runtime and CPU package energy consumption at larger input sizes, while C used less memory. For JSON parsing, C achieved the lowest runtime and energy consumption, whereas Julia showed higher execution times. In contrast, the file-copy workload exhibited only minor differences in runtime and energy consumption between C, Go, and Python at larger input sizes, indicating that language effects become less important in I/O-bound workloads.

Across all benchmarks, CPU package energy closely followed runtime, with longer execution times resulting in higher energy consumption. DRAM energy showed a similar trend and scaled linear with CPU package energy rather than forming an independent characteristic between languages. Peak memory usage did not necessarily correlate with runtime or energy consumption, demonstrating that efficient execution time does not automatically imply efficient memory usage. This was particularly visible in the composed pipeline workloads, where C achieved the lowest runtime and CPU package energy consumption, but required more memory in the streaming pipeline than Go.

The composed workloads further showed that observations from isolated workloads do not always transfer directly to end-to-end data-processing pipelines. Although languages like Julia performed strongly for specific isolated workloads, combining parsing, processing, and file-output stages changed the relative rankings between languages. Overall, the

results indicate that language choice can significantly influence runtime, energy consumption, and memory usage, but the magnitude and direction of these differences depend on the type of workload, input size, implementation choices, and whether workloads are evaluated in isolation or as part of a complete processing pipeline.

7.2 Threats to validity / comparability

The experiments were executed through the DAS-5 scheduler on a reserved compute node. Processes were not explicitly pinned to a specific CPU core, and CPU frequency scaling or Turbo Boost settings were not manually modified by the benchmark framework. The recorded environment reports a CPU frequency range between 1.20 and 3.20 GHz. Therefore, the remaining variation due to operating-system scheduling and dynamic frequency behavior is treated as a threat to comparability and is mitigated by repeated measurements.

RAPL measurements are system-level energy-domain counters rather than per-process counters. Since the benchmark job is executed on a reserved DAS-5 compute node and only one benchmark subprocess is measured at a time, background activity is reduced, but not eliminated. Consequently, the measured energy includes the benchmark process and any system activity that occurs during the same interval. To reduce the influence of this noise, each benchmark configuration is executed multiple times and analyzed using aggregated statistics. Consequently, the subprocess running the benchmark is not tied to a specific CPU core. Therefore, energy is currently measured at the whole processor-domain level rather than only for the core that executes the benchmark. On a multi-socket node, such as DAS-5 dual socket nodes, this can introduce additional noise from processor domains and cores that are not directly executing the benchmark workload. This limits the precision of the measured CPU package energy, although repeated measurements reduce the influence of occasional variation.

Because memory is sampled periodically, very short-lived allocation peaks may be missed. However, the metric should be sufficient to compare the approximate peak memory footprint of the benchmark implementations, as it will capture sustained memory growth.

A further threat to comparability is that equivalent high-level workloads do not necessarily result in equivalent low-level execution across languages. Differences in runtime systems, standard libraries, JSON parsers, compiler optimizations, bounds checking, garbage collection, and memory allocation behavior can all influence runtime, energy consumption, and memory usage. Therefore, the results should be interpreted as comparisons of the selected implementations and runtime environments, rather than as absolute properties of the programming languages alone.

Additionally, all measurements use a cold-start execution model, which means that no warmup run was performed before the measured invocation. For C and Go, compilation is performed before measurement and is excluded entirely. For Julia, JIT compilation occurs within the measurement subprocess and is therefore included in the results. This creates an asymmetry that is mostly visible at small input sizes, where JIT overhead dominates, and diminishes at large input sizes as the compilation cost is amortized. The cold-start model was chosen to reflect realistic single-invocation pipeline execution, which could run once per invocation. A warmup-based measurement would better isolate steady-state

throughput but would represent a different deployment scenario.

The human factor is also relevant when comparing programming languages. The language alone does not produce a benchmark implementation: the programmer chooses data structures, libraries, coding style, and the amount of optimization effort. Ray et al. show that language effects can be difficult to isolate from process factors such as project size, team size, and commit size [46]. Kharal and Brown similarly show that the details of the implementation of the benchmark can strongly affect the performance reported in concurrent microbenchmarks of data-structure [27]. In this thesis, this effect is visible in the JSON parser variants and in the SHA256/XOR workload, where similar high-level algorithms led to different low-level behavior due to bounds checks and vectorization. Therefore, the results should be interpreted as comparisons of representative implementations, not as absolute properties of the languages alone.

7.3 Limitations and future work

Future work could improve measurement isolation by pinned the benchmark subprocess to a specific CPU core and running measurement tasks, such as RSS sampling, on a separate core. In addition, energy measurements can be performed on the specific core on which the benchmark is executed. This would reduce background noise from unrelated cores and improve the comparability of CPU package energy measurements.

Another extension would be to record RSS samples, CPU package energy counters, and DRAM energy counters to log files during execution. Then these time-series measurements could be visualized per language and workload. This would provide a more detailed view of memory growth, energy peaks, and the timing of runtime or garbage-collection effects, rather than only reporting aggregate values such as total energy and peak RSS.

The benchmark suite could also be extended with additional languages that use different runtime and memory-management strategies. For example, adding a garbage-collected language such as Java would make it possible to study whether memory-management activity leads to visible DRAM energy peaks and whether such peaks occur independently of CPU package energy. As discussed in Section 2.5, Python combines reference counting with a cyclic garbage collector. In contrast, Java objects reside in a heap managed by the JVM garbage collector, which identifies unreachable objects and reclaims their memory when garbage collection is performed. This can result in memory patterns where memory usage increases during allocation and decreases after collection [47], a pattern that is also visible in familiar JVM-based applications such as Minecraft. These differences may lead to distinct runtime, memory, and energy-consumption results, particularly in workloads with high memory traffic. Comparing Java with Python and with languages that use manual or more predictable memory-management strategies could provide further insight into how memory-management mechanisms influence both performance and energy behavior.

Future work could also include additional implementation variants within the same programming language. In particular, a NumPy-based Python variant would provide a useful comparison with the current Python implementations. The Python implementations in this thesis use explicit byte-loops. This represents one way of implementing Python code that might not be well optimized. A NumPy variant would make it possible to compare pure-Python execution with a more optimized Python implementation style.

Finally, the current benchmark suite could be extended with additional workload types

and larger application-level scenarios. The present work focuses on controlled CPU, memory and I/O-oriented workloads and composed data-processing pipelines. More complex applications could provide further insight into how language/runtime behavior changes when workloads include, for example, networking, concurrency, or database interaction.

8 Conclusion

This thesis investigated how C, Go, Julia, and Python differ in runtime, energy consumption, and peak memory usage across controlled data-processing workloads. The benchmark suite was designed around recurring stages in ETL-inspired pipelines: JSON parsing, byte-buffer processing, file output, and composed pipeline execution. The goal was not to rank programming languages in isolation, but to evaluate representative implementations under comparable and reproducible conditions.

The results show that no single language is best for all workloads and metrics. C achieves the lowest runtime and CPU package energy in most workload configurations, while Go remains within a small margin of C across workloads. Python frequently achieves low peak memory usage, but incurs high runtime and energy costs in workloads where large byte-wise loops execute at interpreter level rather than through native library code. Julia carries a larger fixed runtime and memory overhead in simple workloads due to JIT compilation startup costs, but matches or outperforms other languages in workloads where LLVM generates vectorized native code for byte-buffer operations, as observed for SHA256/XOR.

The results of the composed pipeline show why isolated workloads are not sufficient on their own. Julia achieves the lowest runtime and CPU package energy in the isolated SHA-256/XOR workload, and Python's runtime and energy approach those of C and Go for isolated file copy at large input sizes. However, in the composed pipeline workloads, C is the strongest baseline implementation for runtime and CPU package energy. The memory results also change between pipeline variants: Python has the lowest peak RSS in the non-streaming pipeline, while Go has the lowest peak RSS in the streaming pipeline. This shows that combining workload stages introduces behavior that is not visible from isolated benchmarks alone.

The results also show that runtime, energy, and memory capture different aspects of execution. Runtime and CPU package energy are often related, but peak memory usage does not always follow the same ranking. An implementation can use relatively little memory while still requiring more time and energy to complete.

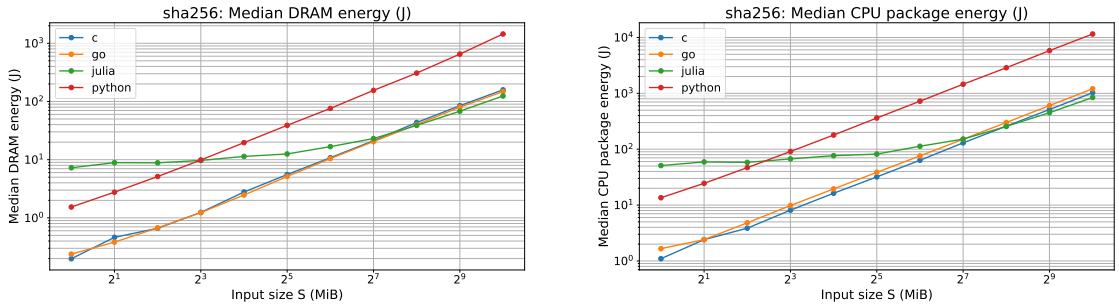
A central finding is that implementation choices strongly influence the results. The JSON parser variants changed performance within the same language, and the SHA256/XOR workload showed different low-level behaviors across languages due to bound checks and vectorization. This confirms that cross-language benchmarking compares representative implementations, including their libraries, runtimes, compiler behavior, and coding choices, rather than programming languages as abstract concepts.

In general, this thesis shows that realistic energy benchmarking requires careful workload design, deliberate implementation choices, and cautious interpretation of the results. Language choice matters, but its effect depends on workload type, library selection, runtime behavior, compiler optimizations, input size, and the human choices made during implementation.

Future work could improve measurement isolation by pinning the benchmark to a specific CPU core. The addition of languages that use different runtime strategies or add additional implementation variants could provide more insight. Further analysis of the implementation choices and compiler optimizations performed by the current code may provide a more detailed understanding of the factors that influence energy efficiency.

Appendix

The source code for the benchmark suite and figures can be found on Github `https://github.com/J-Davidson01/realistic-energy`.



(a) DRAM energy

(b) Runtime

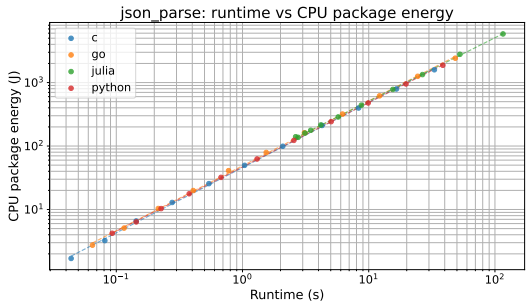
Figure 12: DRAM energy and runtime results for SHA256, where the x- and y-axis are both log scaled

Table 9 presents the scaling exponents derived from logarithmic plots, where both the x-axis and the y-axis are logarithmically scaled. The input data for this table represent the average values of the scatter plot data points that lie above the mean.

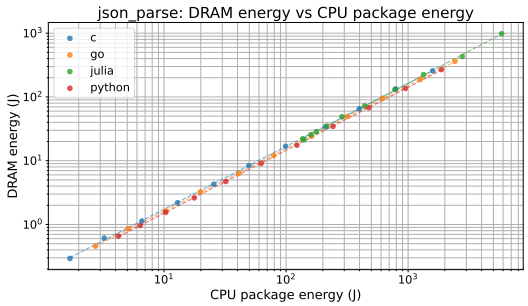
Julia begins with a higher runtime value due to its inherent runtime overhead, as illustrated in Figure 12. Consequently, Julia's scaling exponent increases more slowly at the outset compared to other languages but converges to similar growth rates after an initial period. Because of this slower initial growth, Julia's average scaling exponent sits below 1. Nonetheless, the exponents for all languages stabilize around 1, indicating a linear complexity of O(n).

Benchmark	C	Go	Julia	Python
file_copy	0.9473	0.9786	0.6324	0.9338
json_parse	0.9974	0.9943	0.8642	0.9792
pipeline_e2e	0.9987	0.9940	0.8819	0.9984
pipeline_e2e_stream	0.9967	0.9843	0.9680	0.9931
sha256	1.0010	0.9922	0.6888	0.9942

Table 9: Runtime scaling exponents obtained from the log–log runtime plots. An exponent of 1 indicates linear runtime complexity.



(a) Energy & Runtime



(b) PKG & Dram

Figure 13: Energy & Runtime correlation results on the left and PKG & Dram correlation on the right for the json_parse workload.

Figure 14 is included in addition to Section 6.5, sharing similar results for the languages as were visible for the CPU package and runtime.

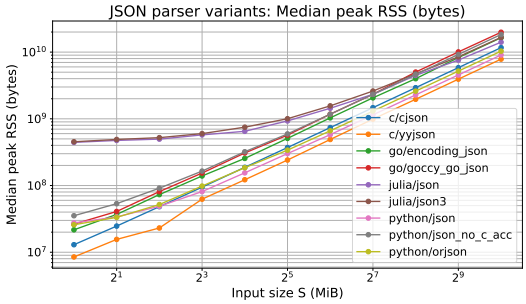


Figure 14: Peak RSS for JSON parser variants, lower is better.

More figures can be generated by plotting results obtained by running the experiment, using the script `plot_results.py` included in the repository.

References

[1] Apr. 2026. URL: https://www.dutchnews.nl/2026/04/no-new-electricity-connections-in-utrecht-due-to-grid-capacity/ (visited on 06/25/2026).

[2] Lotfi Belkhir and Ahmed Elmeligi. "Assessing ICT global emissions footprint: Trends to 2040 recommendations". In: *Journal of Cleaner Production* 177 (2018), pp. 448–463. ISSN: 0959-6526. DOI: 10.1016/j.jclepro.2017.12.239.

[3] Adel Noureddine, Romain Rouvoy, and Lionel Seinturier. "A review of energy measurement approaches". In: *ACM SIGOPS Operating Systems Review* 47.3 (2013), pp. 42–49. DOI: 10.1145/2553070.2553077.

[4] Rui Pereira et al. "Ranking programming languages by energy efficiency". en. In: *Sci. Comput. Program.* 205.102609 (May 2021), p. 102609. DOI: 10.1016/j.scico.2021.102609.

[5] Rui Pereira et al. "Energy efficiency across programming languages: how do energy, time, and memory relate?" In: *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2017. Vancouver, BC, Canada: Association for Computing Machinery, 2017, pp. 256–267. DOI: 10.1145/3136014.3136031.

[6] Antonio Gordillo et al. "Programming Languages Ranking Based on Energy Measurements". In: *Software Quality Journal* 32 (2024), pp. 1539–1580. DOI: 10.1007/s11219-024-09690-4.

[7] Simon Jansson and Johan Johansson. *Benchmarking for Realistic Application Workloads: Experiences Using the Java Virtual Machine*. Tech. rep. TRITA-ICT-EX-2012:144. KTH Royal Institute of Technology, 2012. URL: https://chschulte.github.io/teaching/theses/TRITA-ICT-EX-2012:144.pdf (visited on 06/28/2026).

[8] Stefan Bouckaert et al. "Benchmarking computers and computer networks". In: *EU FIRE White Paper* (2010). URL: http://www.crew-project.eu/sites/default/files/Benchmarking%20computers%20and%20computer%20networks.pdf (visited on 06/28/2026).

[9] Ankit Srivastava. "Use of Python in data science, data integration and data engineer". In: *Int. J. Sci. Res. Eng. Manag* 8.7 (2024). URL: https://www.researchgate.net/profile/Ankit-Srivastava-62/publication/389582098_Use_of_Python_in_Data_Science_Data_Integration_and_Data_Engineer/links/67c8706032265243f58294d5/Use-of-Python-in-Data-Science-Data-Integration-and-Data-Engineer.pdf (visited on 06/28/2026).

[10] Surajit Chaudhuri and Umeshwar Dayal. "An overview of data warehousing and OLAP technology". en. In: *SIGMOD Rec.* 26.1 (Mar. 1997), pp. 65–74. DOI: 10.1145/248603.248616.

[11] Ashish Thusoo et al. "Data warehousing and analytics infrastructure at facebook". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 2010, pp. 1013–1020. DOI: 10.1145/1807167.1807278.

[12] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools*. 2nd ed. Boston, MA: Addison-Wesley, 2007. ISBN: 9780321486813.

[13] Thomas Kistler. "Dynamic runtime optimization". In: *Joint Modular Languages Conference*. Springer. 1997, pp. 53–66. DOI: 10.1007/3-540-62599-2_30.

[14] John Aycock. "A brief history of just-in-time". In: *ACM computing surveys (CSUR)* 35.2 (2003), pp. 97–113. DOI: 10.1145/857076.857077.

[15] Nnaemeka Kingsley Ugwumba and Peter Sunday Jaja. "Ahead-of-time vs. just-in-time compilation trade-offs: Empirical performance studies". In: (2025). DOI: 10.21203/rs.3.rs-7915532/v1.

[16] Timo Keskiniemi. "Measuring Performance in Go". In: (2022). URL: https://www.theseus.fi/bitstream/handle/10024/703715/keskiniemi_timo.pdf?sequence=2 (visited on 06/21/2026).

[17] Alan AA Donovan and Brian W Kernighan. *The Go programming language*. Addison-Wesley Professional, 2015. ISBN: 978-0134190440.

[18] Python software Foundation. *GC - garbage collector interface*. May 2026. URL: https://docs.python.org/3/library/gc.html (visited on 06/21/2026).

[19] Gergö Barany. "Python Interpreter Performance Deconstructed". In: *Proceedings of the Workshop on Dynamic Languages and Applications*. Dyla'14. Edinburgh, United Kingdom: Association for Computing Machinery, 2014, pp. 1–9. ISBN: 9781450329163. DOI: 10.1145/2617548.2617552.

[20] Brian W Kernighan and Dennis M Ritchie. *The C programming language*. Pearson Educación, 1988. ISBN: 978-9332549449.

[21] Jeff Bezanson et al. "Julia: A Fast Dynamic Language for Technical Computing". In: *CoRR* abs/1209.5145 (2012). arXiv: 1209.5145.

[22] John Keiser and Daniel Lemire. "On-demand JSON: A better way to parse documents?" In: *Software: Practice and Experience* 54.6 (2024), pp. 1074–1086. DOI: 10.1002/spe.3313.

[23] David Halliday, Robert Resnick, and Jearl Walker. *Fundamentals of physics*. John Wiley & Sons, 2013. ISBN: 978-0471320005.

[24] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011. ISBN: 978-0128119051.

[25] GNU Project. *Auto-vectorization in GCC*. 2011. URL: https://gcc.gnu.org/projects/tree-ssa/vectorization.html (visited on 05/26/2026).

[26] Pedro Martins et al. *It's Not Easy Being Green: On the Energy Efficiency of Programming Languages*. 2024. arXiv: 2410.05460.

[27] Rosina F Kharal and Trevor Brown. "Performance Anomalies in Concurrent Data Structure Microbenchmarks". In: (2022). arXiv: 2208.08469 [cs.DC].

[28] Samuel Williams, Andrew Waterman, and David Patterson. "Roofline: An Insightful Visual Performance Model for Multicore Architectures". In: *Proceedings of the 2009 ACM/IEEE Conference on Supercomputing (SC)*. ACM, 2009. DOI: 10.1145/1654059.165407.

[29] William A. Wulf and Sally A. McKee. "Hitting the Memory Wall: Implications of the Obvious". In: *ACM SIGARCH Computer Architecture News*. Vol. 23. 1. 1995, pp. 20–24. DOI: 10.1145/216585.216588.

[30] Andreas Herten, Olga Pearce, and Filipe S. M. Guimarães. *An HPC Benchmark Survey and Taxonomy for Characterization*. 2025. arXiv: 2509.08347 [cs.PF].

[31] Hardeep Kaur Dhalla. "A performance analysis of native json parsers in java, python, ms. net core, javascript, and php". In: *2020 16th International Conference on Network and Service Management (CNSM)*. IEEE. 2020, pp. 1–5. DOI: 10.23919/CNSM50824.2020.9269101.

[32] Fips Pub. "Secure hash standard (shs)". In: *Fips pub* 180.4 (2012), p. 2012. URL: https://luca-giuzzi.unibs.it/corsi/Support/papers-cryptography/fips-180-4.pdf (visited on 06/27/2026).

[33] Shay Gueron, Simon Johnson, and Jesse Walker. "SHA-512/256". English. In: *Proceedings - 2011 8th International Conference on Information Technology*. Proceedings - 2011 8th International Conference on Information Technology: New Generations, ITNG 2011. United States: IEEE Computer Society, 2011, pp. 354–358. DOI: 10.1109/ITNG.2011.69.

[34] C Ruemmler and J Wilkes. "An introduction to disk drive modeling". In: *Computer (Long Beach Calif.)* 27.3 (Mar. 1994), pp. 17–28. DOI: 10.1109/2.268881.

[35] Avishay Traeger et al. "A nine year study of file system and storage benchmarking". In: *ACM Transactions on Storage (TOS)* 4.2 (2008), pp. 1–56. DOI: 10.1145/1367829.1367831.

[36] Martin Grambow et al. "Using microbenchmark suites to detect application performance changes". In: *IEEE Transactions on Cloud Computing* 11.3 (2022), pp. 2575–2590. DOI: 10.1109/TCC.2022.3217947.

[37] Yelp Inc. *Yelp Open Dataset*. 2026. URL: https://business.yelp.com/data/resources/open-dataset/ (visited on 06/27/2026).

[38] Yinan Li et al. "Mison: a fast JSON parser for data analytics". In: *Proceedings of the VLDB Endowment* 10.10 (2017), pp. 1118–1129. DOI: 10.3929/ethz-b-000234616.

[39] Yan Cui. "An Evaluation of Yelp Dataset". In: *CoRR* abs/1512.06915 (2015). arXiv: 1512.06915.

[40] Sarabjeet Singh and Manu Awasthi. "Memory centric characterization and analysis of SPEC CPU2017 suite". In: (2019). DOI: 10.1145/3297663.3310311.

[41] Jonathan Davidson. *Realistic Energy*. Version 1.0. May 2026. URL: https://github.com/J-Davidson01/realistic-energy (visited on 06/27/2026).

[42] Henri Bal et al. "A medium-scale distributed system for computer science research: Infrastructure for the long term". In: *Computer* 49.5 (2016), pp. 54–63. DOI: 10.1109/MC.2016.127.

[43] DAS-5. *DAS-5: Distributed ASCI Supercomputer 5*. 2026. URL: https://www.cs.vu.nl/das5/ (visited on 05/19/2026).

[44] Spencer Desrochers, Chad Paradis, and Vincent M Weaver. "A validation of DRAM RAPL power measurements". In: *Proceedings of the Second International Symposium on Memory Systems*. 2016, pp. 455–470. DOI: 10.1145/2989081.2989088.

[45] Kashif Nizam Khan et al. “Rapl in action: Experiences in using rapl for power measurements”. In: *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)* 3.2 (2018), pp. 1–26. DOI: 10.1145/3177754.

[46] Baishakhi Ray et al. “A large scale study of programming languages and code quality in github”. In: *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*. 2014, pp. 155–165. DOI: 10.1145/2635868.

[47] Stephen M Blackburn, Perry Cheng, and Kathryn S McKinley. “Myths and realities: The performance impact of garbage collection”. In: *ACM SIGMETRICS Performance Evaluation Review* 32.1 (2004), pp. 25–36. DOI: 10.1145/1012888.1005693.