Reducing Computational Waste in Configuration Management Through a Lazy Loading and Delta Driven Approach

Caio R. Oliveira [*], Arthur A. Melo [1], Carla Rocha [1] [†], Isaque Alves [2]

[1]Universidade de Brasília (UnB)

[2]Universidade de São Paulo (USP)

caiorocoli@gmail.com, arthuralves1538@gmail.com,

caguiar@unb.br, isaque.alves@ime.usp.br

Abstract. *Cloud-scale infrastructure has made Green Computing a major concern in software engineering. Continuous provisioning and management of parallel systems impose a direct, quantifiable environmental footprint. Contemporary agentless Configuration Management (CM) frameworks are designed around idempotency and bulk-fetching optimizations. However, they exhibit latent hardware-level inefficiencies due to the broad use of eager-loading initialization strategies. This approach requires loading of modules and dependencies at initialization, regardless of execution context. As a result, there are redundant I/O operations, unnecessary system calls, and increased CPU idle-wait states. Altogether, these factors contribute to unsustainable energy consumption in continuous delivery pipelines. To overcome this limitation, this study uses the Design Science Research (DSR) methodology to design and empirically evaluate Ch-aOS. Ch-aOS is a novel CM framework that implements a lazy-evaluation architecture via dynamic import dispatch and a delta-driven in-memory approach to idempotency. Modules are loaded strictly on demand at runtime, eliminating superfluous initialization overhead. This limits resource consumption to each execution context's operational requirements while maintaining batch collection mechanisms needed for idempotency. Experimental findings show that micro-level optimizations, such as reduced system calls and idle-wait CPU time, produce measurable macro-environmental benefits across large-scale deployments. This reduces hardware-level inefficiencies, specifically CPU idle-wait and unnecessary system calls, which are fundamental drivers of energy consumption at the node level.*

1. Introduction

Green Computing is the practice of designing, developing, and utilizing IT infrastructure to maximize energy efficiency and minimize negative environmental impacts [Angelaki et al. 2025]. While early efforts in this field predominantly focused on hardware optimizations, i.e., advanced cooling systems and low-power processors, recent literature

[*]UnB, LabLivre
[†]Brasil Participativo

emphasizes that software architecture directly dictates how hardware resources are utilized, making Green Software Engineering an essential subfield [Jin et al. 2025]. Within this domain, optimizing execution flows, reducing execution time, and minimizing unnecessary hardware activations are key strategies for achieving sustainability.

Configuration Management (CM) tools, such as Ansible [1] and Pyinfra [2], rely on idempotency and bulk-fetching to safely automate infrastructure. However, their execution architectures often introduce hardware-level inefficiencies. When a CPU waits for redundant disk or network responses, it enters an *idle-wait* state, elevating power consumption without productive work [Truong et al. 2017]. In continuous CI/CD pipelines, these micro-level inefficiencies scale into significant cumulative energy waste and increased carbon emissions. Consequently, a Green CM tool requires an architecture that minimizes the risk of both the control node and the target node from entering energy-draining states, as well as limiting CPU cycles, I/O operations, and network latency.

One cause of this computational waste may lie in the architectural design of current CM tools, specifically their use of an *eager-loading* approach. To generalize to a range of operating systems and contexts, traditional configuration managers load a comprehensive suite of modules and dependencies at initialization, irrespective of the actual execution context [Bâra et al. 2024]. For instance, when executing a playbook dedicated solely to package management, the tool imports modules for managing files, services, and users, even though these may be unnecessary for this playbook. While eager loading offers benefits such as simplified dependency management and fail-fast checks, it has Green Computing drawbacks, including redundant I/O operations during this process that increase unnecessary read syscalls and memory allocations, making its energy usage unsustainable for CI/CD pipelines.

To address these inefficiencies, we use the Design Science Research (DSR) methodology to design and evaluate Ch-aOS (Change an OS) [3]. While Ch-aOS preserves the batch collection system used for idempotency already existing in other tools, it also uses a *Lazy Evaluation* architecture [Guimarães and Pereira 2023], where, by implementing a dynamic import dispatch system, Ch-aOS ensures modules are loaded strictly on demand at runtime, reducing unnecessary initialization overhead and resource consumption, aligning with the operational requirements of each execution context. The optimizations proposed translate into environmental impacts, since in large cloud deployments and data centers, reducing idle-wait CPU time across nodes also decreases overall power draw and cooling demands, aligning automation tasks more closely with Green Computing requirements.

To guide this research, we formulated the following Research Questions: RQ1 How does the architectural design of CM tools (eager vs. lazy loading) impact hardware-level computational waste? and RQ2 To what extent can a delta-driven state resolution approach to CM reduce the energy footprint and execution time when compared to industry standards? To answer these questions, we provide (i) a specific approach to idempotency and architecture for CM, (ii) a tool utilizing said approach and architecture, and (iii) an empirical evaluation of said tool against industry standards.

[1]https://www.ansible.com
[2]https://pyinfra.com
[3]https://ch-aos-ch.github.io/Ch-aOS/

The remainder of this paper is organized as follows: Section 2 discusses related works in the field. Section 3 outlines the experimental setup. Section 4 demonstrates the artifact evaluation. Section 5 provides discussion and implications, Section 6 presents threats to validity, and the conclusion is in Section 7.

2. Background and Related Work

The intersection of software engineering and environmental responsibility has recently attracted significant attention under the *Green DevOps* theme, as studies advocate sustainable software engineering frameworks that balance deployment speed and environmental responsibility [Onoja et al. 2024, Ailane et al. 2025]. Additionally, [Kosbar and Hamdaqa 2025] identified "Sustainability Smells in IaC", demonstrating how inefficient code practices can directly impact cloud energy consumption. Although these works present methodologies and indicators for advancing Green DevOps, they do not propose entirely new tools.

Previous studies have empirically analyzed the performance and efficiency of CM tools. Studies such as [Syed and Anazagasty 2023] and [Benson et al. 2016] provide extensive surveys of the execution overhead in tools such as Ansible and Terraform. However, these investigations primarily focus on benchmarking tools under default configurations to characterize trade-offs rather than exploring or proposing alternative architectural paradigms. In performant CM, the software community has developed various tools and plugins to mitigate execution overhead, including Ansible's Mitogen plugin [4], which optimizes Ansible's performance by using a more efficient connection method. However, Mitogen is excluded from this comparison as it functions as an external optimization layer rather than a standalone architectural paradigm.

Other common tools in the industry, such as SaltStack and Chef, offer better performance than Ansible but rely on a client-server architecture, requiring pre-installed agents on target hosts. This requirement violates the *agentless* focus of this study, which is a requirement for modern ephemeral environments and a core design principle of Ch-aOS.

Pyinfra addresses several of Ansible's performance bottlenecks by translating declarative orchestration statements into pure shell commands for concurrent execution. While Pyinfra serves as a related baseline, this study evaluates how Ch-aOS's specific architectural choices further optimize this paradigm. It is important to note that Ch-aOS uses Pyinfra as an API for orchestration primitives; the Pyinfra analysis used a standard *deploy.py* script via Pyinfra's CLI.

There are industry-wide efforts to standardize on-demand execution, such as Python's PEP 810 [Bucher 2025], which proposes native explicit lazy imports. This macro trend toward deferred module loading strongly validates the relevance of Ch-aOS's architectural paradigm.

A significant advancement in this domain is presented by [Gurijala 2025], who demonstrates that declarative infrastructure can reduce waste in cloud operations by replacing iterative implementation stages with exact desired-state definitions. This declarative principle closely aligns with Ch-aOS's system management approach, which calcu-

[4]https://mitogen.networkgenomics.com/

lates the full delta needed to restore the system to the desired declared state. To the best of our knowledge, no prior work provides a Configuration Management architecture explicitly focused on reducing computational waste at the system-call level and empirically evaluates it against industry-standard tooling.

3. Study Design

This work adopts the Design Science Research (DSR) methodology [Hevner et al. 2004], a framework for solving practical problems by creating and evaluating IT artifacts. Following the DSR guidelines, we structure this study around two phases: (i) define the architectural requirements for a Green CM tool and implement said artifact, and (ii) evaluate the artifact's utility and efficiency empirically.

To evaluate the performance and environmental footprint of the proposed artifact against industry standards, we compare Ansible, Pyinfra, and Ch-aOS across a standardized package management workload. We structured nine distinct experimental scenarios, each one executed $N = 10$ times per tool using an automated script to mitigate manual interference. The Independent Variables are the CM architecture (Ch-aOS vs. Ansible vs. Pyinfra), the operational complexity (1-11 packages), and the environment in which we test the tools. The Dependent Variables measured are execution latency, total syscalls, and energy consumption.

We collected all metrics in a tightly controlled Arch Linux environment (16GB RAM, Intel i7-8665U). As a rolling-release distribution with an optimized package manager, it isolates the tools' architecture overhead. We utilized *time* for execution latency, *strace* for system call categorization, and *turbostat* to capture CPU package power draw (PkgWatt). Notably, PkgWatt captures the entire CPU power consumption, translating directly to real-world usage

For fairness and reproducibility, all tools were run with their out-of-the-box configurations, utilizing the latest stable versions with default privilege escalation settings to reflect real-world adoption scenarios. We executed the tests locally to isolate the tools' architectural overhead from external network latency, providing a uniform best-case scenario for each tool. To mitigate execution noise caused by OS-level background processes or caching anomalies, the results are based on the median and mean across iterations for each scenario.

An important methodological consideration is the usage of each tool. We implemented the experimental workload using an *Ansible playbook*, a *Pyinfra deploy.py*, and a *Ch-aOS role*. While Ansible and Pyinfra utilized their standard built-in package modules, Ch-aOS's role was explicitly designed to leverage its delta-driven declarative approach. As a result, the Ch-aOS role executes more complex internal logic to calculate the desired versus current states, even though the end user interacts with it through simple YAML files. This comparison remains methodologically sound, as each implementation faithfully reflects the officially recommended usage patterns of its respective framework, thereby ensuring that the evaluation captures the real-world operational characteristics of each tool, which reflects their underlying architecture, rather than artificially constrained or optimized configurations.

Finally, there is an intrinsic architectural difference regarding target-node execution. To converge the target's state, Ansible uses *Ansiballz* payloads, requiring the target

node to allocate resources to initialize a Python interpreter runtime, unpack the payload, and load dependencies for each execution. In contrast, Pyinfra (and, by proxy, Ch-aOS) translates orchestration logic into pure shell commands executed directly via SSH or a local shell. By bypassing the remote Python runtime overhead, Pyinfra and Ch-aOS possess a structural and tested advantage in target-node energy efficiency. Another clear difference between Ansible's and Pyinfra's architectures is their fact-gathering systems. At the same time, Pyinfra gathers context solely based on the operation called and/or the facts requested by the user, while Ansible's default settings force it to perform an automatic *fact gathering* based on eagerly loaded modules. Although manually disabling this feature would reduce Ansible's artificial overhead, it was maintained to reflect realistic out-of-the-box usage, highlighting how eager loading inherently penalizes baseline execution times. More importantly, even without fact gathering, Ansible's core *Ansiballz* deployment strategy inherently requires more CPU cycles and I/O operations per task than Ch-aOS's centralized delta resolution.

4. Results

We divide our results into two distinct parts, following the DSR methodology. First, the technological artifact conception, where we discuss our architecture and approach to idempotency, and secondly, the empirical comparison between our tool, representing our architecture, and industry-standard tools (Pyinfra and Ansible), representing the traditional agentless architecture.

4.1. The artifact: Ch-aOS

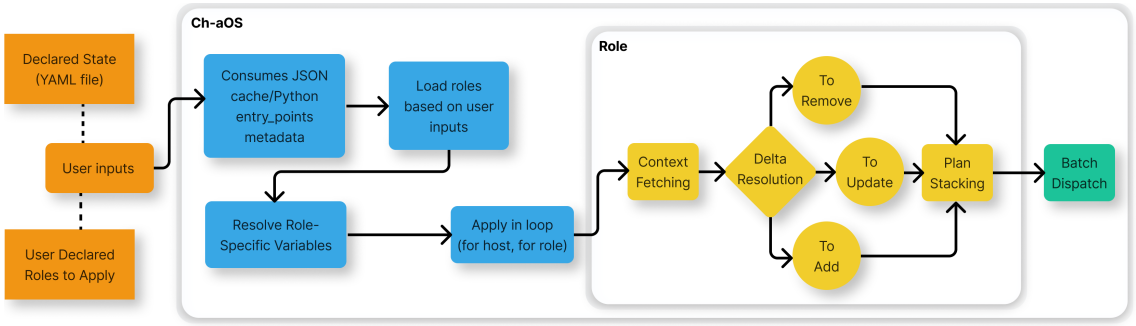

Figure 1. Ch-aOS's approach to idempotency and role flow.

Ch-aOS is a free and open-source agentless configuration management suite. While it leverages Pyinfra's thoroughly tested SSH command dispatching under the hood, it implements a custom architectural overlay focused on pure declarativity and a minimal I/O footprint. The suite introduces a built-in observability tool, the *Logbook*, designed to provide granular, per-host metrics on memory usage and CPU load, writing telemetry directly to a local database and optionally to a JSON file. The project's source code and data extracted can be found in section 8.

The main execution entry points in Ch-aOS are *Roles*, pre-built, isolated logic modules distributed as Wheel files (*.whl*) that receive the desired end-state as input via pure YAML data (see Figure 1), analogous to Ansible's *modules*. To ensure consistency and avoid dependency conflicts, Ch-aOS is packaged and distributed as a self-contained

PYZ archive. This packaging strategy allows users to dynamically extend the plugin-based architecture without polluting the host's global environment.

Lazy Import Mechanism and Plugin Discovery

Ch-aOS's architecture is that of a *microkernel*, which dictates that all non-essential functionality must be modular and pluggable into the main monolith only when required. To mitigate an initialization tax observed in traditional tools, Ch-aOS leverages the standard Python *entry_points* via the *importlib.metadata* built-in API.

Instead of executing Python code to discover plugins, Ch-aOS reads the static metadata files generated during the installation of its *.whl* packages. The actual code compilation into memory (via the *.load()* method) is strictly deferred until the exact moment the user invokes a specific role. Heavy internal dependencies (such as the *Rich* library) are also strictly lazy-loaded. Additionally, discovery mapping is cached locally in a single *.json* file, reducing file system traversal across N plugins to a single read.

Batch Idempotency and Centralized Delta Resolution

The most significant architectural change in Ch-aOS is in its approach to idempotency and state resolution. Traditional tools often re-check the target's current state multiple times when the same abstraction is called repeatedly (e.g., during package installation or uninstallation), resulting in significant syscall overhead. To maintain declarativity and maintain speed, Ch-aOS isolates this workload into the control node's memory using three different phases:

1. **Context Fetching:** Instead of iterative queries during execution, a batched sequence of commands (e.g., fetching package lists or reading *shadow* hashes) is dispatched via the Pyinfra API to collect the current state of the target system in a single initialization phase.
2. **Delta Resolution:** The execution then calculates the state drift entirely in memory using set theory (see Figure 1). While simple presence checks (like package installation) are resolved using relative set complements ($I = D \setminus C$), complex configurations (e.g., user management and permissions) require structural delta resolution. Let D be the desired state mapping and C be the current state mapping for an entity e. The Enforcement set E is defined as:

   $$E = \{e \in D \mid e \notin C \vee C(e) \neq D(e)\}$$

   Where the equivalence $C(e) = D(e)$ is the structural attributes, such as groups and declarative file contents. By delegating the structural evaluation to the CPU's cache, Ch-aOS can reduce syscalls by a significant amount.
3. **Plan Stacking and Batch Dispatch:** The resulting enforcement sets are stacked into unified execution plans and applied in bulk, preventing iterative network degradation.

Security, Isolation and Plugins

Ch-aOS operates entirely in user space, deferring privilege escalation to the final Pyinfra SSH execution phase. The local control node is insulated from arbitrary root-level execution. The distribution script installs Ch-aOS in a strict root-level directory within */opt*. Furthermore, Ch-aOS's registry enforces ecosystem trust. To mitigate supply chain attacks, plugins accepted into the registry must be compliant with the following constraints: (i) submissions must be entirely open-source with standard licensing; (ii) registry additions require verified cryptographic signatures; (iii) plugins are distributed exclusively as pre-built *.whl* files to avoid malicious code injection; (iv) SHA256 checksums are mandatory; and (v) dependencies must pass an automated *pip-audit* vulnerability scan prior to installation.

4.2. Artifact Evaluation: Comparative Analysis

The aggregate metrics collected across all scenarios show a clear advantage for Ch-aOS. On average, Ch-aOS's execution time was 3.34s, representing a 46.56% decrease compared to Pyinfra (6.25s) and a 63.54% decrease compared to Ansible (9.16s). These reductions are correlated with significant reductions in system calls, with Ch-aOS improving by 51.10% over Pyinfra and 58.61% over Ansible on average. Ultimately, this resulted in a drastic reduction in hardware-level power consumption.

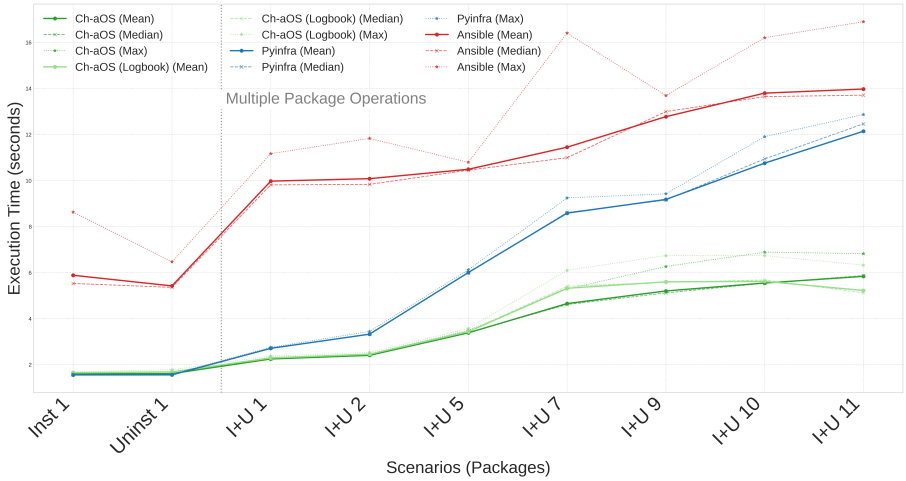

Figure 2. Baseline execution time across scenarios.

Regarding pure time consumption (see Figure 2), Ch-aOS consistently maintained lower execution times across multi-package runs compared to both competitors, leaning towards CPU-bound processing rather than I/O-bound waiting. This advantage becomes more pronounced as the package size grows, where Ch-aOS's centralized delta resolution outpaces the iterative dispatching of traditional tools. Additionally, alongside the baseline executions, supplementary runs were conducted utilizing Ch-aOS's built-in observability tool, the Logbook (see Figure 3). The data indicates that, even when accounting for the Logbook's expected I/O overhead, Ch-aOS's memory and CPU footprint are generally stable and low, reflecting expected usage.

To define exactly how Ch-aOS achieves this optimization, the *strace* analysis was

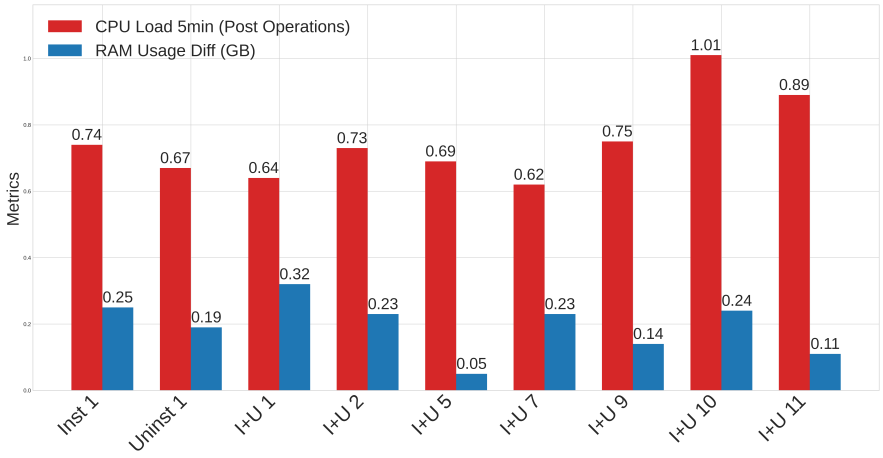

Figure 3. Logbook observability overhead metrics.

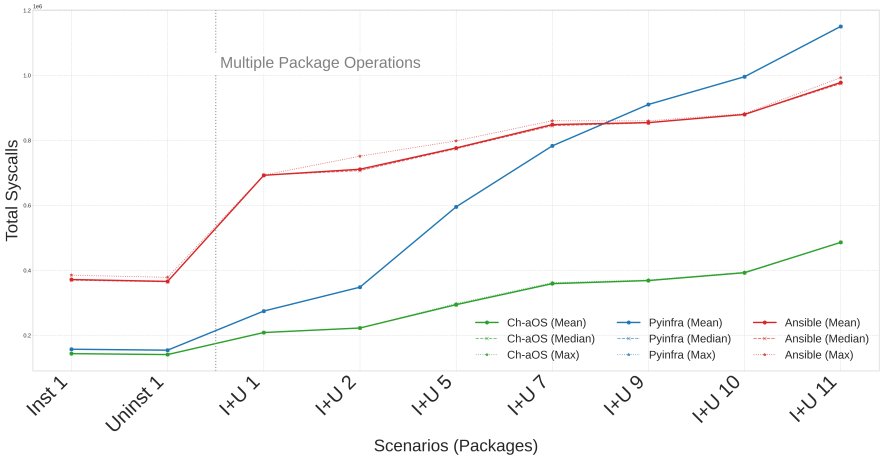

Figure 4. Total syscalls per scenario demonstrating the cost of iterative loading.

particularly useful to categorize and count syscalls (see Figures 4 and 5). In the highest complexity scenario (*I+U 11*), the architectural divide becomes more evident. When evaluating typical execution cost, calculated as the sum of median syscalls across all categories, Ansible generated 6,363,398 system calls, and Pyinfra reached up to 5,255,750. In contrast, Ch-aOS managed the same workload with only 2,569,927 syscalls. This represents a reduction of approximately 51% compared to Pyinfra and 59% compared to Ansible. These metrics indicate that centralized, in-memory delta is more resource-efficient than traditional eager or iterative I/O patterns.

This efficiency starts at the very beginning of the tools' initialization. We conducted profiling using Python's *-X importtime* flag (see Figure 6), which revealed that Ch-aOS's lazy imports did indeed reduce the tool's overhead by a significant amount.

Ultimately, these temporal and system-level optimizations determine each architecture's energy footprint. We calculated the total energy consumption (see Figure 7) using the formula $Energy = Power \times Time$, integrating the average CPU package power draw (PkgWatt) over the execution window. Ansible's eager-loading approach es-

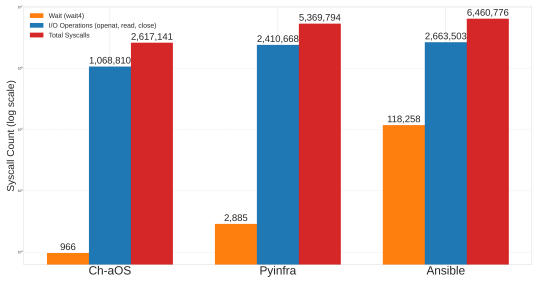

Figure 5. Syscall category breakdown (Wait vs. I/O).

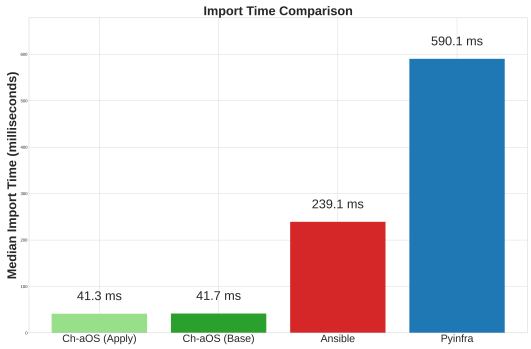

Figure 6. Import time comparison in milliseconds.

tablishes a baseline cost of approximately 80 Joules, even for single-package operations, and scales to over 175 Joules in high-complexity scenarios (*I+U 11*). Pyinfra shows a noticeable degradation as operations scale. Its architecture forces the CPU into continuous active-wait cycles, causing energy consumption to rise sharply to nearly 165 Joules. Ch-aOS, however, maintains a stable, easy-to-scale curve, with its maximum Joule consumption just over 100 Joules. This indicates that the highest workload performed by Ch-aOS yields energy measurements comparable to those of the lowest workload performed by Ansible.

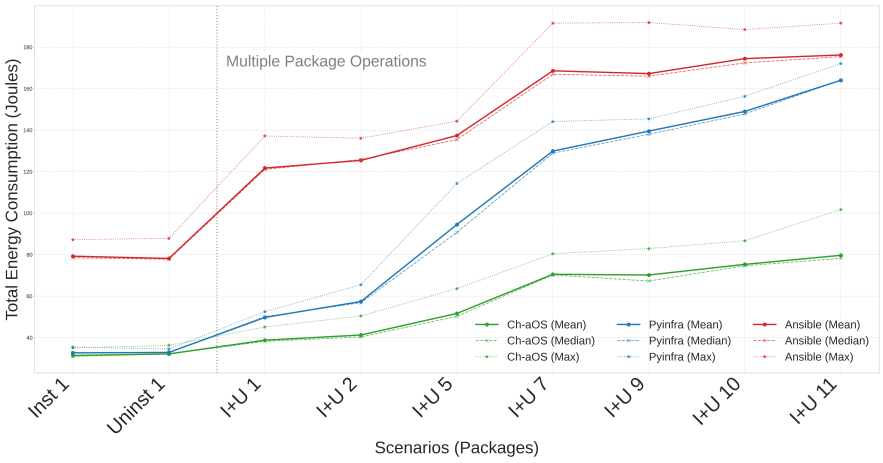

Figure 7. Energy consumption comparison between tools.

5. Discussion

It is a significant finding that Ch-aOS outperforms Pyinfra in execution time, system calls, and energy efficiency, despite using Pyinfra's own SSH transport and command dispatch primitives. This performance disparity suggests that the primary bottleneck in current CM tools may not lie in the programming language, network protocol, or the raw execution engine, but rather in the tool's architecture and design.

The performance gains achieved by wrapping an existing framework with a lazy-evaluating, delta-driven microkernel indicate that resolving architectural inefficiencies

may yield scalable performance improvements. This approach allows developers to retain the ecosystem and faster development cycles of higher-level languages without rewriting execution primitives in lower-level languages.

An important methodological consideration is the complexity of the Ch-aOS *Role* utilized for the benchmarks. While its centralized state-resolution logic makes the implementation more verbose than Ansible's YAML or Pyinfra's scripts, this complexity is entirely abstracted from the end-user via simple YAML configuration dictionaries. Furthermore, to maintain a rigorous comparison, we explicitly avoided utilizing Pyinfra's built-in package modules inside the Ch-aOS Role. Since Pyinfra's modules already include internal idempotency checks, wrapping them would have created a "double idempotency check" scenario—wasting CPU cycles verifying the state in the Ch-aOS control plane only to redundantly verify it again on the execution plane. By preventing this overlap, we avoided an artificially unfair scenario and ensured that the benchmark measurements accurately reflected each tool's true architectural overhead.

An important observation from the baseline execution metrics (see Figure 2) is that in the single-package scenarios (*Inst 1* and *Uninst 1*), Pyinfra outperformed Ch-aOS specifically in raw execution time. This performance difference highlights a clear trade-off of a strictly declarative architecture. Pyinfra's imperative, command-translating design allows it to dispatch a task with near-zero local-state resolution immediately. Ch-aOS, however, must pay a fixed initial computational tax: calculating the mathematical delta for both additions *and* subtractions prior to any execution. For a singular package, this overhead exceeds Pyinfra's immediate dispatch time.

Conversely, this overhead is what enables Ch-aOS to scale efficiently. As operational complexity increases, Pyinfra becomes slower, resulting in longer run times and higher energy consumption. Ch-aOS absorbs this complexity in its delta calculation step, resulting in a flatter scaling curve. This scaling behavior reinforces that Ch-aOS was optimized specifically for more complex tasks, rather than simpler ones.

6. Threats to Validity

As with any empirical evaluation, this study has validity threats worth noting.

Internal Validity: Internal threats concern factors that might influence variables without our knowledge. To mitigate OS noise, we ran tests $N = 10$ times, using median and mean aggregations to reduce caching anomalies and random job spikes. Additionally, isolating execution to a local environment ensured that network latency did not inflate execution times.

Construct Validity: This relates to whether our metrics accurately reflect the concept of "computational waste." We utilized *strace* and *turbostat* as proxies for software inefficiency. While *turbostat* provides highly accurate PkgWatt readings, it captures the entire CPU package power, including background OS processes, though this is uniform across all tools.

External Validity: External threats concern the generalization of the results. We established the baseline using Arch Linux's Pacman, an optimized C-based package manager. The absolute latency numbers and specific syscall counts may differ when using different package managers or when execution is subject to network constraints imposed

by globally distributed servers. However, we anticipate that the proportional advantage of our approach will remain consistent.

7. Conclusion

In this work, we investigated the computational waste and architectural built-in inefficiencies in traditional agentless CM tools. By introducing Ch-aOS, we demonstrated that replacing conventional, eager execution methods with a Lazy-Loading Microkernel and in-memory delta resolution successfully reduces I/O operations and execution overhead. Compared to the industry standard, Ansible, Ch-aOS reduced overall execution time by 63.54% and mitigated syscalls by 59.61%. In high-complexity multi-package operations, Ch-aOS mitigated idempotency's energy consumption. While traditional tools escalated to approximately 165–175 Joules, Ch-aOS maintained a stable footprint of under 90 Joules, cutting the energy expenditure by more than half. Ch-aOS shows that architectural decisions at the foundation of DevOps tools can help enable a greener digital transformation under climate constraints.

At the start of this work, we presented two specific RQs that asked how this architecture could be applied to Green Computing issues in industry. These are their respective answers: (i) Lazy Loading can be correlated with a lower syscall amount. (ii) A lower syscall amount, coupled with a delta-driven idempotency approach, can be correlated with a lower energy consumption.

While the reductions are significant, the practical impact of Ch-aOS in highly distributed networks remains uncertain. We gathered our data locally, utilizing a specific package manager. Furthermore, while the current architecture performs well in "fire-and-forget" CLI environments, the possible memory leaks in long-running enterprise server deployments remain untested. There is also an architectural bottleneck regarding the sequential application of roles across multiple hosts, which may limit horizontal scalability.

To address the sequential bottleneck, future research aims to scale the execution model for concurrent environments. We are working on a refactor to parallelize the context-gathering step across hosts simultaneously, calculate the delta centrally, and stack the execution plans using Pyinfra's topological sorter. We also aim to expand Ch-aOS's ecosystem to integrate with more complex user interfaces and test how it scales across multiple scenarios. Although the current research is scoped to package management, we intend to conduct broader research across different workloads, including file templating, permission management, and service orchestration.

8. Data Availability

The scripts used in this project are available in our metrics repository `https://github.com/Dexmachi/csbc_data`, and the project's source code is available in the project's repository `https://github.com/Ch-aOS-Ch/Ch-aOS`.

9. Acknowledgements

This study was financed in part by the General Secretariat of the Presidency of the Republic within the scope of the Brasil Participativo Project and*Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil* (CAPES) - Finance Code 001.

References

Ailane, M. T., Rubner, C., and Rausch, A. (2025). Green devops: A strategic framework for sustainable software development. In *Computer Sciences & Mathematics Forum*, volume 10, page 5. MDPI.

Angelaki, E., Garefalakis, A., Kourgiantakis, M., Sitzimis, I., and Passas, I. (2025). Esg integration and green computing: A 20-year bibliometric analysis. *Sustainability*, 17(7):3266.

Bâra, R.-M., Boiangiu, C. A., and Tudose, C. (2024). Analysing the performance impacts of lazy loading in web applications. *Journal of Information Systems & Operations Management*, 18(1):1–15.

Benson, J. O., Prevost, J. J., and Rad, P. (2016). Survey of automated software deployment for computational and engineering research. In *2016 IEEE SysCon*, pages 1–6.

Bucher, B. (2025). PEP 810 – Explicit lazy imports.

Guimarães, B. C. F. and Pereira, F. M. Q. (2023). Lazy evaluation for the lazy: Automatically transforming call-by-value into call-by-need. *Proc. 32nd ACM SIGPLAN CC*.

Gurijala, S. (2025). Sustainable infrastructure: How declarative and immutable systems reduce waste in cloud operations. *Intl. Journal of Computational and Experimental Science and Engineering*, 11(3):6332–6338.

Hevner, A. R., March, S. T., Park, J., and Ram, S. (2004). Design science in information systems research. *MIS quarterly*, pages 75–105.

Jin, J. N., Ji, E. K., and Zhang, Q. (2025). Green software engineering: A study on energy-efficient design and deployment in cloud infrastructure. *Journal of Data Analysis and Information Processing*, 13:241–254.

Kosbar, S. and Hamdaqa, M. (2025). Smells-sus: Sustainability smells in iac. In *2025 IEEE/ACM 22nd MSR*, pages 801–812.

Onoja, M. O., Onyenze, C. C., and Akintoye, A. A. (2024). Devops and sustainable software engineering: Bridging speed, reliability, and environmental responsibility. *Intl. Journal of Technology, Management and Humanities*, 10(04):60–83.

Syed, A. A. M. and Anazagasty, E. (2023). Ansible vs. terraform: A comparative study on iac efficiency in enterprise it. *Intl. Journal of Emerging Trends in CS and IT*, 4(2):37–48.

Truong, N. K., Pape, C., Rieger, S., and Reißmann, S. (2017). Energy-efficient live migration of i/o-intensive virtual network services across distributed cloud infrastructures. *ICSNC 2017*, page 13.