Code once, Run Green: Automated Green Code Translation in Serverless Computing

Preprint, compiled September 29, 2025

Sebastian Werner, Mathis Kähler, and Alireza Hakamian

University of Hamburg, Germany
{sebastian.werner, alireza.hakamian}@uni-hamburg.de
Technische Universität Berlin, Germany
mathis.kaehler@campus.tu-berlin.de

Abstract

The rapid digitization and the increasing use of emerging technologies such as AI models have significantly contributed to the emissions of computing infrastructure. Efforts to mitigate this impact typically focus on the infrastructure level such as powering data centers with renewable energy, or through the specific design of energy-efficient software. However, both strategies rely on stakeholder intervention, making their adoption in legacy and already-deployed systems unlikely. As a result, past architectural and implementation decisions continue to incur additional energy usage – a phenomenon we refer to as energy debt.

Hence, in this paper, we investigate the potential of serverless computing platforms to automatically reduce energy debt by leveraging the unique access to function source code. Specifically, we explore whether large language models (LLMs) can translate serverless functions into more energy-efficient programming languages while preserving functional correctness. To this end, we design and implement ReFaaS and integrate it into the Fission serverless framework. We evaluate multiple LLMs on their ability to perform such code translations and analyze their impact on energy consumption.

Our preliminary results indicate that translated functions can reduce invocation energy by up to 70%, achieving net energy savings after approximately 3,000 to 5,000 invocations, depending on the LLM used. Nonetheless, the approach faces several challenges: not all functions are suitable for translation, and for some, the amortization threshold is significantly higher or unreachable. Despite these limitations, we identify four key research challenges whose resolution could unlock long-term, automated mitigation of energy debt in serverless computing.

1 Introduction

Cloud computing promises cost-effective virtually infinite on-demand resources, fueling a decades-long migration to the cloud. The increasing demand for cloud resources has consequently led to a predictable rise in energy consumption and global carbon emissions associated with the IT sector [16, 35]. To combat this worrying trend, cloud providers and application developers are being urged to adopt more energy-efficient practices and implement strategies to minimize their carbon footprint, in line with the United Nations sustainable development goals [32] and to consider sustainability as a software quality [14]. Although most cloud providers have already taken measures to lower their carbon emissions [21], a significant amount of emissions is caused by long-standing design choices in already running applications.

It is already well known that the selection of programming language [22], the choice of architecture style [34], and the choice of cloud technologies [15] can have a significant impact on the energy profile of an application. While research and industry are developing new recommendations and new methods to reduce energy consumption for all these choices, most applications will not adopt them just for the sake of saving energy. Legacy applications, particularly those that are merely maintained and no longer actively developed, will remain long-lasting energy consumers operating on remote infrastructure. Hence, application owners have very little incentive to change their applications. We call the cost of these long-past taken energy-consuming design decisions energy debt, a form of technical debt [13].

While the cloud provides quick access to infinite cloud resources, it also makes it easy to forget about resources or lose track of them due to configuration drift or shadow IT [12]. Hence, we argue that cloud providers are in a unique position to help application owners reduce energy debt, especially for legacy applications. This can not only be beneficial to address the environmental sustainability responsibility of the cloud providers but also help them to deprecate old and inefficient platform technologies by moving applications away from them.

One unique cloud technology that can help in automatically reducing energy debt is serverless computing. Serverless computing is a cloud computing model that abstracts away the underlying infrastructure, allowing developers to focus on writing code without worrying about server management. This also enables cloud providers to propose and apply automatic improvements in the backend. For example, Amazon Web Services (AWS) Lambda already tries to automatically upgrade the runtime of the serverless functions to the latest version, which can bring performance and security improvements. However, we argue that serverless computing platforms can do even more, as they

typically have access to the source code of the application. This can allow the serverless provider to change the programming language, the architecture style of the application. Something that has already been explored to improve other serverless application qualities, for example, function fusion [27] to reduce cold starts.

In this paper, we propose a new approach to reduce energy debt in serverless applications by leveraging the unique capabilities of serverless computing and large language models. Asking the research question: **How can we enable cloud computing platforms to automatically improve the energy efficiency of running (legacy) applications through automated code translation?**

For this, we make the following contributions:

1. Evaluate the energy reduction potential in serverless applications by changing the programming language manually.
2. Implement ReFaaS an automatic serverless code transformation tool that can be integrated into existing platforms such as Fission.
3. Evaluate the energy reduction potential of ReFaaS using current large language models (LLMs).
4. We identify four key challenges that must be addressed to realize ReFaaS.

The remainder of this paper is structured as follows: First, in Section 2 we review related work in the area of energy efficiency, serverless computing, and automatic code transformation. Then, in Section 3 we describe the problem of energy debt and how it can be addressed, particularly in serverless computing, including a first evaluation of the energy reduction potential. Following that, we introduce ReFaaS in Section 4 and describe its integration into the existing serverless platforms Fission and evaluate ReFaaS using different configurations and LLMs in Section 5. Finally, we conclude the paper in Section 6 and discuss future work.

2 RELATED WORK

Addressing the environmental and technical sustainability of software systems is a major challenge for the software engineering community [14]. This is especially relevant in the context of cloud computing, where the cost and impact of engineering for sustainability are spread across multiple stakeholders, including cloud providers, application developers, platform engineers, and end-users [34]. At the data center level, hyperscalers are already investing in renewable energy sources and custom hardware to improve energy efficiency [4, 16, 24]. Similarly, on the software engineering level, practitioners are investigating how various design patterns influence energy consumption, for example, within microservice architectural style [3, 34, 36]. More specifically, for software engineering, researchers are exploring the improvement of platform efficiencies [28], minimizing energy consumption and communication [9], enabling dynamic energy and emission controls [6, 10, 31] as well as evaluating the impact of auxiliary services [2, 8]. This also includes the application level with a focus on the energy efficiency of the code itself. Pereira et al., [22, 23] evaluate the energy usage of several programming languages and show a significant difference of almost two orders of magnitude. However, they observed that the relationship between energy consumption and factors such as time, memory, and resource varies across programming languages. Hence, it is evident that programming languages can have a significant influence, though the extent depends on the program type and programming context. For instance, scientific computing applications often involve heavy computation, making language choice affects power usage. Additionally, the context in which a program is developed – such as target hardware, runtime environment, or resource constraints – can amplify or reduce this influence. In this work, we are using this difference in programming language efficiency to enable data centers and platform operators to help application developers make more sustainable decisions.

Nevertheless, current cloud providers are still not equipping users with the necessary transparency and tools to reduce application-side energy consumption [17], which in aggregate might reduce the emissions of data centers more than simply using renewable energy or buying sufficient carbon offsets. In addition, even if developers are made aware of the energy consumption, they often do not act on it as this kind of re-engineering effort is often not rewarded [19]. Hence, with ReFaaS we aim to provide a tool that performs this task automatically, leaving the developer only with the task of reviewing and accepting the changes. Thus, reducing but not eliminating the development effort needed to pay off energy debt.

Automatically transforming code is a long-standing challenge in software engineering, with a variety of approaches proposed over the years. Most recently, the use of machine learning techniques, particularly large language models (LLMs), has gained traction in this area. Here, emerging models such as CodeT5 [33], translation models [29], and more generalized coding models such as Qwen-Coder [26], Gemma [30], or OpenAI's GPT-4 [20] have shown promising results in generating and transforming code. Thus, researchers have already started to explore the potential of LLMs in enhancing the performance of systems [5], as well as to improve the overall efficiency [18]. In this work, we build on these recent advances in LLMs and propose a novel approach to automatically transform code for energy efficiency in the specific context of serverless computing.

3 CLOUD ENERGY DEBT

The energy consumption of a software system is strongly influenced by the architecture, programming language, execution runtime, hardware, and workload of the application [14, 22, 34]. For example, systems that employ machine learning models or integrate with proof-of-work blockchains inherently consume more energy. However, while these choices are often made knowingly and are often a requirement of the context and functionality of the application, the choice of programming language is not necessarily a design decision that is made due to context or functionality, but due to skill and team preferences and for long-term maintainability. Consequently, design choices made years ago persist within applications, allowing energy inefficiencies to accumulate over time. This phe-

Table 1: Functions used for the model evaluation.

Function	Type	Description	Source
f_1	Basic	Basic "hello world" function.	B
f_2	Basic	Basic addition function.	A
f_3	Basic	Basic calculator function with error handling.	A
f_4	Basic	Basic percentage calculator.	A
f_5	Basic	Compound interest calculator.	A
f_6	Basic	Recursive Fibonacci function.	A
f_7	Data Structures	Google Chat webhook function.	B
f_8	Basic	Graph theory solver function.	C
f_9	API	Performs HTTP GET requests to external API and returns transformed output.	D
f_{10}	API	Fetches current weather data using OpenWeatherMap API.	D
f_{11}	Data Structures	Extract and validate user information with nested data structures.	B
f_{12}	Data Structures	Performs ELT on large JSON files.	A
f_{13}	Syntactic Sugar	Uses a decorator pattern for logging.	D
f_{14}	Syntactic Sugar	Implements a multi-stage data processing pipeline.	A

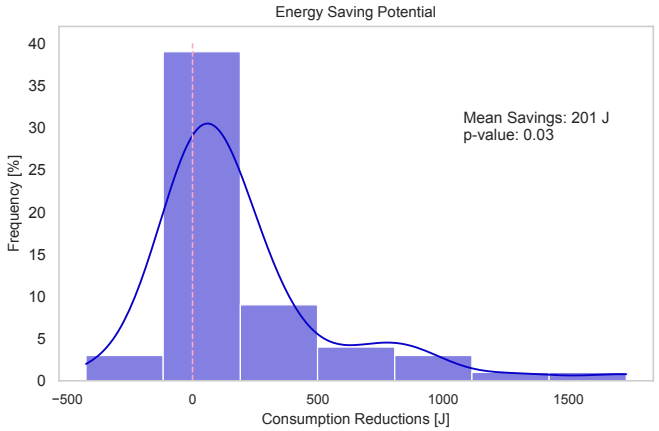

Figure 1: Energy savings per invocation, when manually translating the test functions (Table 1) from Python to Go. Negative values indicate that the Python Version was more energy efficient.

nomenon parallels the concept of technical debt [13], and we refer to it as energy debt.

3.1 *Towards Automated Code Translation*

Unlike technical debt, where eventually the worsening of maintainability will drive the team to refactor the code, energy inefficiency rarely motivates such action–unless it also impacts cost or performance. Hence, we argue that we should address this energy debt through automation, e.g., automatic code refactoring to an energy-efficient language/runtime on the execution platform side. However, many platforms, especially in the cloud, do not have access to the executable code. Hence, code refactoring options are limited. Notably, the serverless paradigm is one of the few cloud models where the platform has access to the code and is fully in charge of the execution environment. Thus, we argue that serverless platforms have a unique opportunity to automatically repay the energy debt of legacy applications by refactoring the code to a more energy-efficient language/runtime. Hence, in this work we focus on how we can design serverless platforms such that they can improve the energy efficiency of (legacy) functions through automated code translation as a first step towards answering our research question.

3.2 *Serverless Energy Savings Potential*

However, before we can answer the research question, we need to evaluate how much energy is saved by changing the language/runtime of a function. For this, we selected Python as the source language, given its popularity in serverless function languages, and its relatively low energy efficiency [22]. Similarly, we selected Go as the target language for translation, as it is still a very common serverless language with a comparatively high energy efficiency. Based on Pereira et. al. [23], we can expect an up to 24x improvement. However, this included a broader range of workloads than typically found in serverless computing.

To get a more realistic picture of what we could expect, we collected a set of serverless functions (see Table 1) that are written in Python and manually translated them to Go. We collected these functions from several sources: (1) from simple introductory programming tasks, (2) online examples, e.g., from AWS or Google's documentations, (3) some we created based on CodeNet [25], and (4) some we created from API documentation.

We then run each function in a minimal container and invoked it **1,000** times using JSON-based events, measuring energy consumption in joules per invocation along with other performance indicators such as runtime and CPU usage(see FunctionMetrics Table 2). Each function is running on a single core, without any parallelism in a closed-loop setting. Hence, consuming the JSON test events as fast as possible (see an example event in Listing 3). We repeated these experiments five times and accumulated the results in Figure 1. Note, we also collected the results of each of these invocations to later use as indicators for correct translation of functions.

Based on these first experiments, we can see that the potential for savings exists. The savings are minimal for most of the tested functions, as the functions were rather basic. However, we already see larger savings for the more complex functions, such as the *data structures*. Moreover, even if the savings are small in absolute joules, they already represent a 70% average reduction in consumption.

However, we also see that some functions present with an increased consumption. Notably, these were the basic functions that did simple calculations. We assume that Python has a more efficient way to deserialize and serialize the JSON events, as this is the majority of the execution time used for these functions.

Thus, the savings depend on the function context, creating a varying translation budget for each function that we can spend to save energy after a reasonable number of invocations. This indicates that we need a way to predict if functions have a chance of reducing energy before attempting automatic translation, which is one of the key future challenges (**C1**) that we identified. Nevertheless, we see the potential, especially as more complex functions all showed strong saving potential and thus present a first system design in the following section that can perform automated code translation.

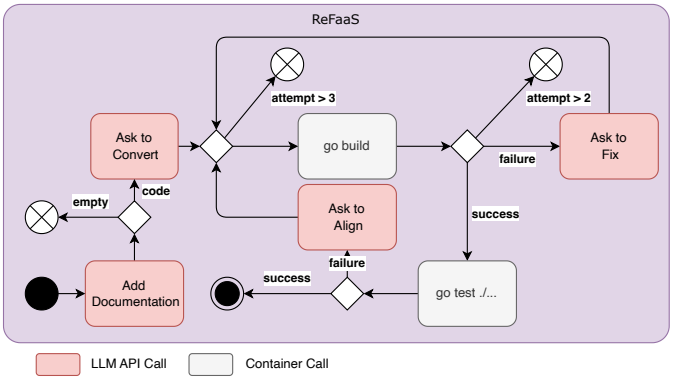

Figure 2: Translation Pipeline used in ReFaaS, highlighting the Chain of Thought approach with external validation (gray boxes) that is used to achieve high translation quality.

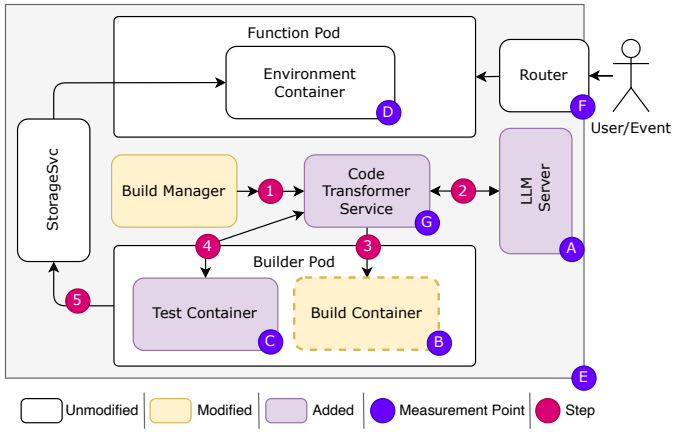

Figure 3: Integration of ReFaaS into Fission's architecture and process, including relevant Steps, changed components, and Measurement Points relevant for the evaluation in Section 5).

4 REFAAS

In the following, we describe the design of ReFaaS ((Rewriting Functions as a Service)) and how we integrated it into the Fission[1] serverless platform, as an example.

4.1 *Design of ReFaaS*

ReFaaS is designed as a standalone service that expects a deployment package (e.g., a zip file) containing the function code and configurations to build. In addition, ReFaaS also requires a set of test files (see Listing 4,5), which are used to evaluate the function after it has been translated. Each test file contains an input event (as JSON) and an expected output response (as JSON). ReFaaS treats both the deployment package and the resulting artifact as black boxes; hence, the tests conducted as black box tests. Serverless computing comes with fixed invocation interfaces, which makes it rather simple to run these black-box tests against implementations of different languages.

ReFaaS enables the use of several different LLMs to translate functions. Moreover, we can also use different LLMs at each stage of the translation process. For this, ReFaaS provides an extensible pipeline interface that allows defining the LLM-Client, prompt, and verification and recovery steps. The pipeline is defined in a YAML file, which allows for defining the LLMs and their parameters. The verification steps are used as gateways to stop faulty or unnecessary translations. For example, if a function does not compile, we can stop trying to test it. Moreover, we allow for multiple recovery steps, for example, prompting the LLM to fix the code given the compilation error message and running the compiler again until we get to a buildable version. The main pipeline we used is presented in Figure 2.

For the pipeline, we follow a multi-shot, Chain-of-Thought approach. The pipeline consists of three main steps: (i) the code translation, (ii) the building of the new artifacts, and (iii) the testing. In each step, we also added additional steps to solve problems (recovery) to increase the likelihood of a successful translation. For example, in the code translation step, we added a step to ask the LLM to first document the original code excessively. In case the build step fails, we also have a step to ask the LLM to resolve the build errors. Lastly, in the testing step, we ask the LLM to ensure that the code is aligned with the original by providing the original source code and the current source code as input to the LLM. Each step, including the recovery steps, is always limited to a fixed number of attempts to ensure termination.

ReFaaS operates a simple job queue, so that multiple conversion requests can be processed in parallel. However, the limiting resource is the used LLM API and underlying hardware. We specifically designed ReFaaS to use locally deployed LLMs, as the source code would typically be confidential, although commercial cloud-based LLMs can also be used, in less critical cases. Moreover, ReFaaS also allows each of the steps to use a different LLM API, so that we can also combine more narrow LLMs, e.g., coding assistance LLMs, for some tasks.

Once a job is terminated, either successfully or unsuccessfully, a new deployment package is generated, either the original or the translated one. Hence, from the perspective of the serverless platform, it always gets a deployment package as a user would provide it. ReFaaS is available on GitHub at: `https://github.com/ISE-TU-Berlin/ReFaaS`.

4.2 *Fission Integration*

Figure 3 shows how we can integrate ReFaaS into the Fission deployment process. The Fission deployment process includes a build pod and an execution pod. The build pod is responsible for building the function so that it can be executed and scaled. The compiled and prepared build artifact is stored in a cluster-wide storage service. Hence, for us to integrate ReFaaS, we mainly need to modify the behavior of the build pod, and if a translation is successful, instruct the function pod to deploy a different environment container, i.e., a Go-based one instead of a Python environment. We augmented the build process by adding a fork into the build manager, which first calls ReFaaS to translate the function (1). Once ReFaaS receives the request, it starts the conversion pipeline (Figure 2), which requires one or multiple API calls to the LLM Service (2).

[1] `https://fission.io/docs/architecture/`

Once we receive the translated source code, we build it using the build pod (3). Once the build is successful, we create a new test container (4) and run the tests using the translated artifact. If these tests conclude successfully, we store the translated artifact in the cluster-wide storage (5) and update the function pod configuration to use the translated artifact. Otherwise, we build the original artifact and store it in the cluster-wide storage (5). Since ReFaaS works asynchronously, we can also first build the original artifact and deploy it while attempting to translate the function, thus minimizing potential cold-start delays. If the translation is successful, we can roll out the change.

4.3 Critical Assumptions

For ReFaaS to be effective, we make several assumptions. For one, we assume that the user provides a sufficient number of test events, e.g., archiving at least full test coverage in the original function. This gets increasingly difficult, and also less realistic for functions that have side effects, e.g., writing to a database or consuming an external API. We've experimented with the use of Local-Stack, a set of mock-up services for AWS, as one option to offer a predictable environment for ReFaaS to run dynamic black box tests. However, for now, we assume that the user is providing the test events and an appropriate test environment. Moreover, we assume that the function is translatable, i.e., that there are no critical dependencies in the original that don't exist in the target language, e.g., something like TensorFlow. We also assume the original function is correct and valid.

Moreover, ReFaaS currently performs semantics comparison of the resulting JSON responses (see Listing 4,5) to validate the test cases. Here, small differences in the content of error messages or changes due to external influence (e.g., time) can change the semantic similarity between expected and test output. Hence, the current prototype assumes that the tests produce deterministic output. This is also an issue, in case the functions use external APIs without a means to mock an environment. In general, the automatic translation validation is one of the future key challenges (**C2**) we identified for automatic code translation.

During the translation process, we measure the runtime and energy consumption of the translated function. Thus, we also try to detect if the translated function is consuming more resources than the original function for the same inputs, in which case we abort the translation process. However, this still can incur energy, time costs that do not contribute to the goal ReFaaS.

5 EVALUATION

In this section, we evaluate the performance of ReFaaS in terms of translation success and the savings it can provide. For this, we selected four state-of-the-art LLMs that can be run locally (for accurate energy measurements) and three pipeline configurations.

5.1 Experiment Design

Towards that end, we selected four models: `qwq` the reasoning model and `Qwen2.5-Coder` from [26], `Deepseek-R1` [7] a recent well performing reasoning model and `Gemma3` [30]. For each model we tested three different pipeline configurations:

Table 2: Relevant metrics we collected in ReFaaS evaluation.

Metric	Description	Measurement Point
Conversion Metrics		
Runtime [s]	Time it took to complete the pipeline for the function.	G
Token [#]	Token it took the LLM to prompt and generate the response.	A
Energy [Wh]	The total energy consumption of ReFaaS and the LLM-Service for producing the code.	A, E
Temperature [°h]	The total accumulated temperature per hour that the server produced in additional heat during the code transformation.	A, E
Function Metrics		
Validation [binary]	Indicates if all black-box tests produced valid results.	C
Test [#]	Number of tests that produced valid results.	C
Buildable [binary]	Indicates if the function was built without errors.	B
Δ Energy [Wh]	Change in energy consumption between the original and transformed function.	C, D
Δ CPU [s]	Change in CPU consumption between the original and transformed function.	C, D
Δ Memory [MB]	Change in Memory consumption between the original and transformed function.	C, D
Δ ColdStart [s]	Change in cold start duration between the original and transformed function.	F

(i) **Chain of Thought** (CoT), as described in Figure 2 and Section 4, (ii) **Creative CoT**, the same as CoT but with a high temperature of 0.8 instead of 0.1, and (iii) **Single-Shot**, where we followed the same pipeline as in CoT but without any repetitions or recovery steps. We run each experiment three times and always utilize all test functions (see Table 1) in random order. In all cases, we performed translations from Python to Go functions. We mainly selected Go over other, more energy-efficient languages such as C++, Rust, or C, due to the authors' familiarity with Go, which enabled manual evaluation of translation accuracy. During the experiments, we measured the time and energy consumption of the translation process, as well as the time and energy consumption of the resulting functions. A detailed description of the metrics can be found in Table 2, for example, to measure the conversion Energy [Wh], we deployed Scaphandre[2] to probe the energy consumption of each container in the Fission namespace (E), and the dcgm-exporter[3] to collect the consumption of the GPU (A) in Figure 3. We also measured the accumulated temperature of the GPU and CPU using dcgm-exporter and Prometheus node-exporter, as cooling significantly contributes to the energy consumption of the compute infrastructure [1]. We performed all experiments on a Precision 3680, using an Nvidia 4500 Ada GPU, an Intel i7 14700K, and 64 GB of memory.

For the stability of ReFaaS and the selected pipeline, we observed (i) if a function was buildable after finishing the pipeline (Buildable), (ii) how many of the tests passed after a translation (Test) and (iii) if all tests of the transformed function passed (Validation). If a function was valid, we ran the same micro-benchmark as described in Section 3.

5.2 Experiment Results

Table 3 shows the overall results of the evaluation with a buildable ratio of at least 25 percent, showing that the better-performing pipelines produce up to 79 percent of buildable

[2] `https://github.com/hubblo-org/scaphandre`
[3] `https://github.com/NVIDIA/dcgm-exporter`

Table 3: Aggregated experiment results that resulted in at least 25% of buildable functions.

Model	Configuration	Validated [%]↑	Tests [%]↑	Buildable [%]↑	$\overline{\text{Runtime}}$ [s]↓*	$\overline{\text{Tokens}}$ [#]↓*	$\overline{\text{Energy}}$ [Wh]↓*	$\overline{\text{Temp.}}$ [°h]↓*
qwq	Chain of Thought	36	35	50	177	4685	14	7
	Creative-CoT	21	24	50	45	4187	7	3
	Single-Shot	14	20	64	22	2448	5	2
qwen2.5-coder 32b	Creative-CoT	43	51	79	41	2319	7	3
	Chain of Thought	36	39	71	127	2534	11	6
	Single-Shot	29	39	71	19	1462	5	2
gemma3 27b	Single-Shot	43	41	43	28	2075	5	3
	Creative-CoT	36	39	50	51	3366	7	3
deepseek-r1 32b	Chain of Thought	14	12	29	37	4386	6	3

* Only successful translations are considered.

functions. Additionally, the Creative-CoT `Qwen2.5-Coder` and Single-Shot `Gemma3` yield the best results with a 43 percent translation success rate. Notice that `Deepseek-R1` did not perform well, which is mainly due to extraction problems, where the model would always return more text than the requested code, causing issues in parsing and using the output. For validation of the functions, we use the same JSON-based test files in both Python and Go and then compare the output semantically. For each function, we used between 4 test inputs. We selected the test inputs to achieve a high degree of code coverage and test important edge cases. Here, we often saw issues, especially for error-handling cases, see Figure 4. In the original functions (Listing 1), some errors were just JSON-wrapped Python errors, which, even if translated faithfully (as in Listing 2), would not result in the same type of output (Listing 4,5). This highlights one of the main challenges in translating functions, which is a way to fairly and accurately validate the generated code (**C2**).

Still, we see that the models can translate a function into a valid form, as we have seen in the validation results. However, only 43 percent is not satisfying, as this implies a large number of functions are not yet translatable into a valid version. However, if we look across all runs, we can see that most functions were translated in a valid form at least once, see Figure 5.

We assume that the size of the used models, or the number of recovery attempts, was not sufficient to reach a valid translation. Similarly, it could be that more test cases would have yielded better results by providing more errors for the LLM to align the solution more with expected outputs. However, here we have to also balance overfitting to test cases. Thus, more thorough prompt engineering and use of larger models could improve the results. However, the potential of the approach is evident from the validation percentage.

In terms of the cost associated with the valid translation of a function, we observe that these models range from 5 to 14 Wh, with an increased temperature load of 2 to 7 degrees per hour. With a mean runtime of 61 seconds and 3050 tokens, which would amount to around 2 USD cents per function using the Google Gemini API. Figure 6 shows the savings potential for the three best-performing models and configurations in terms of invocations until the translation is amortized, i.e., the point at which the accumulated energy reduction of invoking the Go function is greater than the total energy consumption of us-

Listing (1) Original Python Function for f_5

```python
#...
try:
    result = num1 / num2
    return success(result)
except Exception as e:
    return error(400, str(e))
#...
```

Listing (2) Generated Go function for f_5

```go
//...
if num2 == 0 {
        return error(400, "division by zero")
} else {
        return success(num1 / num2)
}
//...
```

Listing (3) Test2 input for f_5

```
{
    "num1": 1, "num2": 0, "operation": "/"
}
```

Listing (4) Expected Output of Test 2 in f_5

```
{
    "statusCode": 400,
    "body": "{\"error\": \"ZeroDivisionError:
        division by zero\"}"
}
```

Listing (5) Output for Test 2 using Listing 2

```
{
    "statusCode": 400,
    "body": "{\"error\": \"division by zero\"}"
}
```

Figure 4: Example of a translation error.

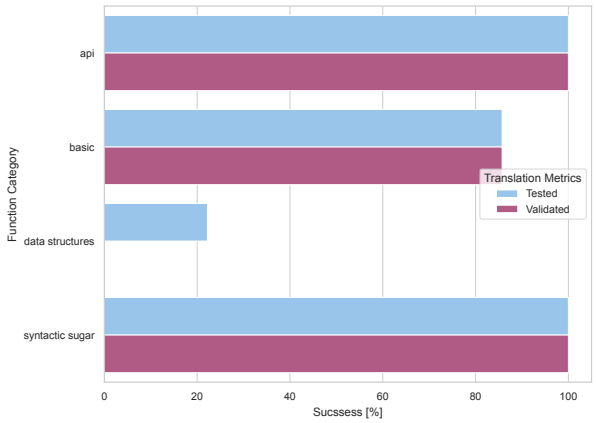

Figure 5: Aggregated code translation success across all models and configurations.

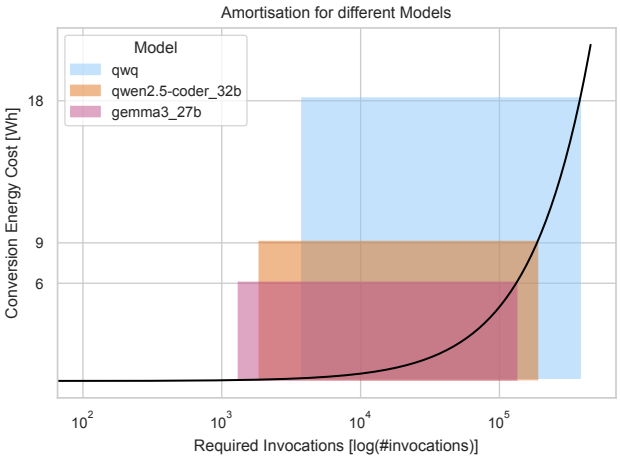

Figure 6: Number of invocations before the conversion is amortized. The area indicates the range between the best and worst case we observed within the test set table 1.

ing ReFaaS. The shaded area indicates the best and worst case for the given model, depending on how much energy reduction the function can achieve (see Figure 1). Here, we can see that some functions, with very little savings potential, require nearly 10^6 invocations, which might be more than we can expect during the lifetime of some functions. On the other hand, we can see that some functions would already be amortized after around 3000 to 5000 invocations. Figure 6 does not include failed attempts or functions that consume more energy than before the translation, as these costs can never be amortized. Instead, these costs would act as additional conversion costs, increasing the time until functions amortize. Hence, to make the ReFaaS-approach viable, we need to reduce these two drivers, the energy consumption of failed attempts and the energy consumption of running ReFaaS. The first driver could be managed by addressing challenge (**C1**) – predict if functions have a chance of reducing energy before attempting automatic translation. The second driver could be managed by finding more energy-efficient code translation models, another key future challenge (**C3**) we identified.

Overall, we can see, however, that we can design serverless platforms in such a way that they can automatically improve the energy efficiency of functions with the help of current large language models. A more thorough investigation, using also different programming languages and a more robust set of test functions, could further refine our results (**C4**); however, currently no representative sample of real-world serverless functions exists to see how well ReFaaS would perform in practice.

6 CONCLUSION

In this paper, we identified the problem of energy debt, a special form of technical debt, that accumulates due to energy inefficient design choices. We argued that developers are not incentivized or aware to pay off this debt, even less so than regular technical debt. Hence, we propose ReFaaS an automatic code translation service that can be used to translate serverless functions from one language to another, which is one strategy to pay off energy debt.

6.1 Discussion

With ReFaaS we showcase the potential of automatic code translation to reduce the energy consumption of cloud services. However, for this, the platform providers need access to the source code and configurations of the applications. While this is not always the case, we argue that serverless platforms, such as Fission, are uniquely able to perform this and other automatic improvements.

We showed that ReFaaS can translate serverless functions with a success rate of up to 43 percent, which is already a promising result, and that in the best case, such translation can save up to 70% of invocation energy costs. However, we also see several limitations with the current approach. Firstly, the selected functions were still basic, and we still need to compare ReFaaS with more complex functions. However, early coding benchmarks [11] already show promising trends even for complex functions, e.g., with dependencies. Secondly, we used representative test inputs to validate the functions and also performed manual reviews of the generated code. However, with increased function complexity, this becomes increasingly difficult, requiring more sophisticated validation methods. Here, we could see a human-in-the-loop design, where instead of automatically applying the refactoring changes, developers can review the changes together with potential energy and cost savings. While this is not a trivial task for developers it could reduce the barrier to getting started. Moreover, a switch from Python to Go can also reduce execution time and memory use, which translates to direct cost savings in the serverless model, hence, further incentivizing developers to take a look. Thirdly, LLMs are prone to hallucinations, which also applies to generated code, creating an issue in providing stable translation time and code quality. Here, we argue that the pipeline approach with multiple validation checkpoints (see Figure 2) can offer a strategy to improve translation stability.

6.2 Challenges and Future Work

Throughout the paper, we identified current limitations on using the ReFaaS approach. From them, we thus identified four main challenges that need to be addressed in future work:

C1 Translation Prediction – that can predict the likelihood that a function can be translated and will offer a reasonable energy reduction.

C2 Automatic Translation Validation – programmatic means to fairly compare the behavior of two function implementations beyond back-box testing.

C3 Energy-Efficient Translation Models – that can deliver good code-to-code transformations with less energy consumption.

C4 Robust FunctionSet – that can offer a robust and broader set of functions to test the translation process, not only for the aim of energy reduction, but also for serverless evaluation in general.

Nevertheless, we believe that these challenges can and will be addressed in the future, as already promising work is being done in these areas. Moreover, while ReFaaS only performed code translation, other automatic refactoring could follow a similar approach, e.g., changing configurations or replacing known energy-consuming libraries. Thus, offering further options to code once and run green.

REFERENCES

[1] Ahmed A. Alkrush, Mohamed S. Salem, O. Abdelrehim, and A.A. Hegazi. Data centers cooling: A critical review of techniques, challenges, and energy saving solutions. *International Journal of Refrigeration*, 160:246–262, 2024.

[2] Maria C. Borges, Joshua Bauer, Sebastian Werner, Michael Gebauer, and Stefan Tai. Informed and assessable observability design decisions in cloud-native microservice applications. In *2024 IEEE 21st International Conference on Software Architecture (ICSA)*, pages 69–78, 2024.

[3] Carlo Centofanti et al. Impact of power consumption in containerized clouds: A comprehensive analysis of open-source power measurement tools. *Computer Networks*, 245:110371, 2024.

[4] Pritam Jaywant Chaudhari, Satoshi Kaneko, and Taku Okamura. Estimating power consumption of collocated workloads in a real-world data center. *2023 International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, pages 1–7, 2023.

[5] Bowen Cui, Tejas Ramesh, Oscar Hernandez, and Keren Zhou. Do Large Language Models Understand Performance Optimization?, March 2025. arXiv:2503.13772 [cs].

[6] Wellington Silva de Souza, Arman Iranfar, Anderson Bráulio, Marina Zapater, Samuel Xavier de Souza, Katzalin Olcoz, and David Atienza Alonso. Containergy—a container-based energy and performance profiling tool for next generation workloads. *Energies*, 13:2162, 2020.

[7] DeepSeek-AI, Daya Guo, et al. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning, January 2025.

[8] Madalina Dinga, Ivano Malavolta, Luca Giamattei, Antonio Guerriero, and Roberto Pietrantuono. An empirical evaluation of the energy and performance overhead of monitoring tools on docker-based systems. In *International Conference on Service-Oriented Computing*, pages 181–196. Springer, 2023.

[9] Zhengxin Fang, Hui Ma, Gang Chen, and Sven Hartmann. Energy-efficient and communication-aware resource allocation in container-based cloud with group genetic algorithm. In *International Conference on Service-Oriented Computing*, pages 212–226. Springer, 2023.

[10] Guillaume Fieni, Romain Rouvoy, and Lionel Seinturier. SmartWatts: Self-Calibrating Software-Defined Power Meter for Containers. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 479–488, Melbourne, Australia, May 2020. IEEE.

[11] Mingsheng Jiao, Tingrui Yu, Xuan Li, Guanjie Qiu, Xiaodong Gu, and Beijun Shen. On the Evaluation of Neural Code Translation: Taxonomy and Benchmark. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1529–1541, Luxembourg, Luxembourg, September 2023. IEEE.

[12] Stefan Klotz, Andreas Kopper, Markus Westner, and Susanne Strahringer. Causing factors, outcomes, and governance of shadow it and business-managed it: a systematic literature review. *International Journal of Information Systems and Project Management*, 7(1):15–43, 2019.

[13] Philippe Kruchten, Robert L. Nord, and Ipek Ozkaya. Technical debt: From metaphor to theory and practice. *IEEE Software*, 29(6):18–21, November 2012.

[14] Patricia Lago, Sedef Akinli Koçak, Ivica Crnkovic, and Birgit Penzenstadler. Framing sustainability as a property of software quality. *Communications of the ACM*, 58(10):70–78, September 2015.

[15] Jialun Lyu, Jaylen Wang, Kali Frost, Chaojie Zhang, Celine Irvene, Esha Choukse, Rodrigo Fonseca, Ricardo Bianchini, Fiodar Kazhamiaka, and Daniel S. Berger. Myths and Misconceptions Around Reducing Carbon Embedded in Cloud Platforms. In *Proceedings of the 2nd Workshop on Sustainable Computer Systems*, pages 1–7, Boston MA USA, July 2023. ACM.

[16] T. Mastelic, A. Oleksiak, H. Claussen, I. Brandic, J.-M. Pierson, and A. Vasilakos. Cloud computing: Survey on energy efficiency. *ACM Computing Surveys*, 47:36, Jan. 2015.

[17] D. Mytton. Assessing the suitability of the greenhouse gas protocol for calculation of emissions from public cloud computing workloads. *Journal Cloud Computing*, 9(1):45, 2020.

[18] Changan Niu, Ting Zhang, Chuanyi Li, Bin Luo, and Vincent Ng. On Evaluating the Efficiency of Source Code Generated by LLMs. In *Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering*, FORGE '24, pages 103–107, New York, NY, USA, June 2024. Association for Computing Machinery.

[19] Adel Noureddine, Martín Diéguez Lodeiro, Noëlle Bru, and Richard Chbeir. The Impact of Green Feedback on Users' Software Usage. *IEEE Transactions on Sustainable Computing*, 8(2):280–292, April 2023.

[20] OpenAI, Josh Achiam, et al. Gpt-4 technical report, 2024.

[21] Massoud Pedram. Energy-efficient datacenters. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(10):1465–1484, 2012.

[22] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 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*, pages 256–267, Vancouver BC Canada, October 2017. ACM.

[23] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Ranking programming languages by energy efficiency. *Science of Computer Programming*, 205:102609, May 2021.

[24] Jean-Marc Pierson, Gwilherm Baudic, Stephane Caux, Berk Celik, Georges Da Costa, Leo Grange, Marwa Haddad, Jerome Lecuivre, Jean-Marc Nicod, Laurent Philippe, Veronika Sonigo, Robin Roche, Gustavo Rostirolla, Amal Sayah, Patricia Stolf, Minh-Thuyen Thi, and Christophe Varnier. Datazero: Datacenter with zero emission and robust management using renewable energy. *IEEE Access*, PP:1–1, 07 2019.

[25] Ruchir Puri et al. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks, 2021.

[26] Qwen, An Yang, et al. Qwen2.5 Technical Report, January 2025.

[27] Trever Schirmer, Joel Scheuner, Tobias Pfandzelter, and David Bermbach. Fusionize: Improving serverless application performance through feedback-driven function fusion. In *2022 IEEE International Conference on Cloud Engineering (IC2E)*, pages 85–95, 2022.

[28] P. Sharma. Challenges and opportunities in sustainable serverless computing. *ACM SIGEnergy Energy Informatics Review*, 3(3):53–58, 2023.

[29] Marc Szafraniec, Baptiste Rozière, Hugh Leather François Charton, Patrick Labatut, and Gabriel Synnaeve. Code Translation with Compiler Representations. 2022.

[30] Gemma Team, Aishwarya Kamath, et al. Gemma 3 Technical Report, March 2025.

[31] John Thiede, Noman Bashir, David Irwin, and Prashant Shenoy. Carbon containers: A system-level facility for managing application-level carbon emissions. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, SoCC '23, pages 17–31, New York, NY, USA, 2023. Association for Computing Machinery.

[32] United Nations. Transforming our world: the 2030 agenda for sustainable development, 2015.

[33] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, Online and Punta Cana, Dominican Republic, 2021. Association for Computational Linguistics.

[34] Sebastian Werner, Maria C. Borges, Karl Wolf, and Stefan Tai. A comprehensive experimentation framework for energy-efficient design of cloud-native applications. In *2025 IEEE 22st International Conference on Software Architecture (ICSA)*, 2025.

[35] World Bank and ITU. Measuring the Emissions and Energy Footprint of the ICT Sector: Implications for Climate Action. Publication, World Bank, 2024.

[36] Xingwen Xiao, Chushu Gao, and Justus Bogner. On the effectiveness of microservices tactics and patterns to reduce energy consumption: An experimental study on trade-offs. In *2025 IEEE 22st International Conference on Software Architecture (ICSA)*, 2025.