Degree Project in Computer Science and Engineering

Second cycle, 30 credits

Dynamic macro to micro scale calculation of energy consumption in CI/CD pipelines

NIKOLAI LIMBRUNNER

Stockholm, Sweden, 2023

Dynamic macro to micro scale calculation of energy consumption in CI/CD pipelines

NIKOLAI LIMBRUNNER

Master’s Programme, Computer Science, 120 credits
Date: June 29, 2023

Supervisors: Christopher Kugler, Matias Martinez
Examiner: Martin Monperrus
School of Electrical Engineering and Computer Science
Host company: Siemens
Swedish title: Dynamisk beräkning av energiförbrukning i CI/CD-pipelines från makro- till mikroskala

Abstract

This thesis applies energy measurements to the domain of continuous integration (CI) and continuous delivery (CD) pipelines. The goal is to conduct transparent and fine-granular energy measurements of these pipelines, increasing awareness and allowing optimizations regarding their energy efficiency. CI and CD automate processes like compilation, running tests, and code analysis tools and can improve the software quality and developer experience and enable more frequent releases. Initially, the applicability of existing energy measurement approaches for these tasks is analyzed. Afterward, a generic framework consisting of a pipeline run analyzer, a resource consumption collector, and an energy calculator is proposed. A representative implementation for a state-of-the-art infrastructure is devised to demonstrate its functionality, enabling the collection, analysis, and interpretation of data from real-world examples. Finally, it is examined whether this data aligns with the theoretical considerations and can be used to optimize the pipelines. The overall goal is to contribute to the sustainability of DevOps processes and therefore counteract the disastrous consequences of unrestrained climate change.

Keywords

GreenIT, Sustainability, Energy measurement, Software Engineering, CI/CD pipelines, DevOps

Sammanfattning

Denna avhandling tillämpar energimätningar på området kontinuerlig integration (CI) och kontinuerlig leverans (CD). Målet är att genomföra transparenta och finkorniga energimätningar av dessa pipelines, vilket ökar medvetenheten och möjliggör optimeringar av deras energieffektivitet. CI och CD automatiserar processer som kompilering, testkörning och kodanalysverktyg och kan förbättra programvarukvaliteten och utvecklarens upplevelse samt möjliggöra tätare lanseringar. Inledningsvis analyseras tillämpligheten av befintliga metoder för energimätning för dessa uppgifter. Därefter föreslås ett generiskt ramverk som består av en analysator för pipelinekörning, en insamlare av resursförbrukning och en energikalkylator. För att demonstrera dess funktionalitet utarbetas en representativ implementering för en modern infrastruktur som möjliggör insamling, analys och tolkning av data från verkliga exempel. Slutligen undersöks om dessa uppgifter stämmer överens med de teoretiska övervägandena och kan användas för att optimera rörledningarna. Det övergripande målet är att bidra till hållbarheten i DevOps-processer och därmed motverka de katastrofala konsekvenserna av ohämmade klimatförändringar.

Nyckelord

GreenIT, hållbarhet, energimätning, programvaruteknik, CI/CD-pipelines, DevOps

Acknowledgments

I want to thank my supervisor Christopher Kugler, for having the idea for this thesis, supporting me with his extensive knowledge and experience throughout this project, and becoming a friend on the way. I sincerely thank Bayram Sezgin and Andreas Mirrring for their trust and support and all the other great colleagues from the DI IT ALM department. I would also like to express my deepest gratitude to Astrid Eder, who is responsible for this cooperation and has always stood by me as a friend.
I am grateful to Matias Sebastian Martinez for the insightful suggestions and numerous meetings crucial to this project's success. Lastly, I would like to thank Martin Monperrus for sparking my interest in DevOps through his lecture, displaying trust in me, and connecting me with such an exceptional supervisor.

Stockholm, June 2023
Nikolai Limbrunner

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

To prevent the disastrous consequences of unrestrained climate change, a transformation across all industries, including the Information and Communication Technology sector (ICT), is required. ICT's carbon footprint comprises embodied emissions from raw material extraction, manufacturing, and transport, operational emissions due to energy consumption and maintenance, and end-of-life emissions from disposal. Together these emissions are equivalent to those produced by the aviation industry [1].
This work focuses on operational emissions due to energy consumption. Data centers consume today between 1.5-2% of the world's energy consumption with a strong upward trend, despite the progress in efficiency for hardware and software [2]. The electricity depends on the required computing resources combined with the data center's power usage effectiveness (PUE). The associated carbon footprint depends on the composition of the electricity mix used.
A substantial part of the energy used in data centers can be attributed to continuous integration (CI) and continuous delivery (CD) pipelines. These play an essential role in modern software development. Automating the compilation, running tests, and code analysis tools improves the software quality and the developer experience and enables more frequent releases. At the same time, these processes are very resource-intensive and are sometimes executed several times a day, leading to enormous energy consumption.
Connecting the trends of sustainability and CI/CD, the need to balance the benefits of these pipelines with responsible energy consumption is the logical consequence. Intelligent decision-making requires increased transparency

regarding energy consumption and heightened awareness among users and stakeholders. Only in this way, the benefits of pipelines can be sustained while optimizing their carbon footprint.

1.2 Problem definition

CI/CD pipelines are triggered on scheduled intervals or by checking in changes into a code management platform. As these processes occur in the background, the user receives little to no feedback on the required computing power and the associated energy consumption. This research aims to quantify the energy consumption associated with these pipelines, thereby providing a basis for understanding and optimizing their environmental impact. More specifically, this study focuses on answering the following research questions:

- **RQ1:** *Which existing approaches can be used for transparent and granular energy measurements of CI/CD pipelines?*
  Current research explores various techniques for assessing energy consumption at different levels. This work aims to answer which approaches are applicable in the context of CI/CD pipelines. The measurements must be sufficiently detailed to determine the energy usage of individual stages, jobs, and their corresponding steps.

- **RQ2:** *How can energy measurements be efficiently integrated into existing infrastructure used to run CI/CD pipelines?*
  After determining appropriate methods, the study presents a measurement framework and examines how to incorporate them into the existing infrastructure. The pipeline's regular operation must always remain undisturbed, and the computational overhead should be minimized. Moreover, the objective is to monitor all pipelines continuously rather than relying on selective manual evaluations of single runs.

- **RQ3:** *Into which general categories can jobs and steps executed in CI/CD pipelines be divided, and how do these categories contribute to energy consumption?*
  The study analyses the energy consumption for exemplary pipeline runs. Given the diversity of jobs and steps executed in pipelines, we assign them to different categories and consequently analyze the energy consumption of these categories.

1.3 Approach

Researchers have explored diverse methodologies for energy measurement, which can be broadly categorized into physical (external) measurements, energy profiling and model-based measurements, and sensor-based measurements [3]. Many operating systems and software solutions provide access to this data via registers or Application Programming Interfaces (APIs).
The analysis of CI/CD pipelines can be conducted by examining logs created by default by the corresponding platforms. These logs are sufficiently detailed to allow the users to assess whether the system is functioning and to facilitate debugging and can therefore be leveraged for this use case. The recent trend of continuous monitoring allows prompt detection and response to issues, thereby minimizing downtime and enhancing overall system performance [4].
In this work, we combine these two domains to determine the energy consumption of CI/CD pipelines. The methodology starts with analyzing existing energy measurement approaches for CI/CD pipelines. Consequently, we introduce a generic framework called *Planetary* with shared responsibilities: pipeline run analysis, power or resource consumption monitoring, and merging and calculation of energy consumption values. The framework's components should be interchangeable to accommodate the heterogeneity of contemporary IT infrastructures.
A representative implementation for a state-of-the-art infrastructure is devised to demonstrate the functionality of *Planetary*, enabling the collection, analysis, and interpretation of data from real-world examples. Notable challenges encompass data collection from various levels, the diverse nature of IT infrastructures and their numerous components, and the complexities introduced by virtualization and containerization. Furthermore, the approach must introduce minimal overhead to maintain feasibility and enable large-scale deployment.

1.4 Contributions

The contribution of the work can be summarized as follows:

- The evaluation of the applicability of existing energy measurement approaches in the DevOps context

- The design of a generic framework to calculate the energy consumption of CI/CD pipelines

- An exemplary implementation of the framework and integration into an existing live monitoring system
- The evaluation of real-world data and derivation of generic improvement considerations

1.5 Practical context

The practical part, including the benchmarks, is carried out in the Digital Industries IT Application Lifecycle Management (ALM) solution department of Siemens AG in Germany. This department provides services to its customers in Siemens IT and Businesses, consisting of tooling which supports the whole software development lifecycle, including soft- and hardware support in on-premise, cloud, and hybrid environments. As such, the ALM department is an enabler for development teams at Siemens, which have yet to or just recently adopted the usage of CI/CD pipelines to automate as much of the software development lifecycle without having to set up the whole toolchain on their own. As part of Siemens' commitment to sustainability, this research ventures into relatively uncharted territory, representing one of the company's numerous efforts to develop greener IT processes by continuously seeking opportunities to enhance the environmental performance of its operations.

1.6 Outline

Chapter 2 provides the required background knowledge. Chapter 3 examines the current research and state-of-the-art in the domains of Green IT, energy measurement, and DevOps pipelines. Chapter 4 evaluates the applicability of existing energy measurement approaches within the context of DevOps. Chapter 5 introduces a generic framework for energy measurements of CI/CD pipelines, including an exemplary implementation. Chapter 6 analyses the gathered data. Chapter 7 discusses the results, suggests energy improvements for CI/CD pipelines, and critically reviews threats to validity. Chapter 8 concludes the work by summarizing the findings and offering recommendations for future research.

Chapter 2

Background

2.1 Green IT

Green IT, also known as Green Information Technology, refers to using technology to minimize its environmental impact. This concept emerged around 1990, as the concerns about environmental sustainability grew [5]. The key areas of Green IT include energy-efficient hardware, renewable energy, virtualization, cloud computing, recycling and waste reduction, and sustainable software development. Energy-efficient hardware consumes less energy and produces less heat. Using renewable energy sources such as solar, wind, and hydropower can reduce carbon emissions and dependence on fossil fuels. Virtualization and cloud computing minimize the required hardware by using shared resources. At the same time, sustainable software development involves using eco-friendly coding practices and software design principles that optimize resource consumption. Governments have established regulations and incentives to promote adopting renewable and energy-efficient technologies. Embracing sustainable practices can significantly enhance a brand's reputation and image. Moreover, by lowering energy consumption and extending product longevity, Green IT can simultaneously lead to cost savings [6]. While ICT systems are emitters, they can simultaneously be enablers for reducing the carbon footprint through smart systems and monitoring and optimizing processes.

2.2 Energy measurements

Measuring and improving the energy consumption of software, can lead to better battery life on mobile devices and lowered energy costs in data centers.

Additionally, it can be used for performance optimization. If a process consumes more energy than it should, it may indicate a performance issue that can be addressed to improve overall efficiency and effectiveness.
Measuring the energy consumption of software presents several challenges. It can be difficult to isolate the energy consumption of a specific software application from the overall energy consumption of the system it runs on, as the system also powers other hardware components and software processes. The energy consumption of software can be highly dependent on the specific workload and runtime conditions. This dynamic behavior can make obtaining consistent and representative energy consumption measurements difficult. Platform variability makes it difficult to generalize energy consumption measurements across different systems. Finally, measurement techniques may introduce overhead, potentially affecting the accuracy of the measurements or the software's performance during testing [7].
The methods for software energy measurements can be grouped into the following three categories:

- **Direct energy measurements** involve using specialized external hardware devices, often referred to as wattmeters, to directly measure the energy consumption of equipment. These devices can be connected to servers, switches, routers, or other components to obtain energy consumption data or power samples, depending on the device. They can provide precise, real-time energy consumption data. One popular device often used in literature is the Watt Up? Pro which has high reliability and accuracy [8]. On the downside, wattmeters have a limited scope, as they measure the energy consumption of the entire device or system, not just the software itself. This can make it challenging to decompose specific energy consumption of a single software application. Additional costs for buying devices and installing and calibrating a wattmeter can be time-consuming and may require technical expertise.

- **Built-in sensors** integrated into components such as CPUs, GPUs, and motherboards can provide data on parameters like temperature and voltage. This data can be accessed through hardware drivers or specific registers to gather information on power usage. Most servers are also equipped with dedicated sensors in the power supply. Using built-in sensors is cost-effective and allows continuous monitoring by providing real-time data. It also typically requires less setup and configuration than external measurement devices. These sensors provide reasonable estimations but often lack the accuracy of dedicated external measurement devices, potentially leading to imprecise energy consumption calculations [9] [10]. Additionally, not all hardware devices are equipped with built-in sensors, and accessing the sensor data might require platform-specific tools, presenting compatibility challenges.

- **Software-based estimation** offers an alternative to approximate energy consumption in the absence of direct measurements or built-in sensors. These tools use energy models and algorithms to predict energy consumption based on hardware specifications together with workload characteristics or performance counters [11]. As no additional hardware is required, software-based measurement methods are more cost-efficient than hardware-based solutions. At the same time, they are platform-independent, making them more versatile in measuring energy consumption across various systems. Software-based tools are often more accessible, enabling developers to measure energy consumption without specialized knowledge or equipment. On the downside, these tools are generally less accurate than hardware-based measurement devices. They can add overhead regarding computational resources or power usage, significantly affecting the software's performance during testing [12].

Energy consumption can be estimated from power samples through two main methodologies: the interval-based and the weighted average methods, both providing the same total energy consumption. The interval-based method computes the energy consumption by calculating the average power for each time interval, multiplying it by the interval's duration, and then summing up these values. The weighted average method calculates the weighted average power over the total timespan, using this to determine total energy consumption.

2.3 DevOps and CI/CD pipelines

DevOps combines development (Dev) and operations (Ops) practices that focus on improving collaboration and communication between software developers and IT operations teams. This approach aims to streamline software development, deployment, and maintenance processes by relying on continuous improvement, automation, and shared responsibility. Today, Github, Gitlab, and Azure DevOps are the largest cloud-based platforms to store, share and collaborate on Git repositories. They enable development and operations teams to work more efficiently by providing a unified environment for managing the entire software lifecycle. They offer a good alternative to external dedicated Continuous Integration (CI) / Continuous Deployment (CD) servers, like Jenkins or CircleCI, since they can provide the whole functionality in one place, can react directly to changes in the repository, and are also easier to configure in most cases.

2.3.1 Pipelines

CI is a software development practice that involves merging working code into a shared repository multiple times daily. This approach minimizes integration issues and allows teams to detect and fix problems early in development. CI typically involves automated builds and tests, ensuring that the code base remains stable and releasable. CD is the practice of automatically deploying every code change to a production environment once it passes the required tests in the preceding CI stage. This approach reduces manual intervention and enables faster delivery of new features and bug fixes. CI and CD are closely intertwined and are often executed in a common pipeline. Most platforms use common concepts and the YAML file format to create pipelines. YAML is a human-friendly serialization language alternative to the widely used JSON format. The basic building blocks of a pipeline are jobs, which in turn consist of a series of steps. These steps execute commands or scripts to compile the code, run tests and publish/deploy it. A workflow combines the different parts and describes which actions are executed by which runners are triggered by which events. Scheduled events are triggered in a specific preset interval, manual events reacting to clicks or webhook events when creating pull requests or issues [13]. Other common concepts are artifacts to persist data across multiple jobs, caching to store calculated values and temporary variables, and secrets that can be configured and accessed by the pipeline [14].
An exemplary pipeline process could look as follows. Developers push their code changes to a shared version control repository. This event triggers the pipeline, which first compiles the source code and creates an executable or deployable artifact. After the build, automated tests, such as unit, integration, and performance tests, are run to ensure code quality and functionality. If the tests pass, the pipeline deploys the changes to the staging environment for further testing or directly to the production environment.

2.3.2 Runners

Runners are the machines on which the jobs are executed. When creating a pipeline, it can be specified on which operating system and version it should be executed. A runner is a lightweight, standalone application that listens for available jobs and runs them in a specified environment. Generally, we differentiate between hosted runners and self-hosted runners.

Hosted runners are pre-configured cloud instances provided by the DevOps platform, which can be used without additional configuration. These runners are provided and managed by the respective platform. They are available for different operating systems and come with pre-installed software and tools. Hosted runners are convenient, as no maintenance of the runner environment is required. However, the compute resources are shared among users, and there may be limitations on usage, such as the number of concurrent jobs or total minutes per month.

Alternatively, self-hosted runners can be arbitrary machines with the respective runner software installed and registered as runners on the DevOps platform. They support customizable hardware, operating system, software, and security requirements and allow running jobs in private networks. They offer more flexibility but also require the initial setup, including installing the runner software and adding it to the runner pool of the DevOps platform. Self-hosted runners can be physical, virtual, in a container, on-premises, or cloud instances [15]. All three DevOps platforms (Github, Gitlab, and Azure DevOps) considered in this study offer hosted runners and the option to add self-hosted runners. To use a self-hosted runner, the basic steps are similar for the different platforms

1. Set up the environment: Prepare a machine (virtual or physical) with the required operating system and software.

2. Install the runner application: Download and configure the runner application.

3. Register the runner: Use a registration token to authenticate the runner with the repository.

4. Start the runner: Run the application to listen for jobs and execute them when available.

5. Configure the workflow: Update the pipeline workflow files to specify the use of the self-hosted runner by setting the runs-on attribute to the runner's label.

The runner applications' primary responsibility is to execute assigned jobs. Therefore it has to communicate with the platform to receive job assignments and report job status, logs, and artifacts back to the platform. It has to provide an environment specified by the pipeline definition. It then uses an executor to run the assigned job in the given environment. A commonly used executor technology to run jobs in isolated environments is Docker. This simplifies the use of different images. Alternatively, jobs can be run in virtual machines or the host directly.

Running CI/CD pipelines locally can be helpful for testing, debugging, and development purposes and can be leveraged for energy measurements. Github Actions can be run locally using an open-source project called *act*[*]. Similarly, Gitlab Workflows can be run locally using the Gitlab Runner [†]. For Azure DevOps pipelines, there is currently no option to run them locally. However, we can register our local machine as a runner and trigger the run via the platform.

2.3.3 DevOps monitoring

Monitoring is an essential aspect of the DevOps approach, as it enables the performance and reliability of systems. Application performance monitoring to track the performance of application code, database queries, and API calls to identify and resolve performance bottlenecks. Infrastructure monitoring is used to monitor the health and performance of servers, containers, networks, and other infrastructure components to detect and fix issues before they cause downtime or poor responsiveness. Log and event monitoring collects and analyzes log data and system events to identify patterns and debug problems. One popular monitoring setup is the combination of Prometheus [‡] and Grafana [§]. Prometheus is a monitoring and alerting platform that uses a pull-based model to collect and store metrics from various sources, such as applications and infrastructure components. Components expose metrics, and Prometheus scrapes them regularly and stores them in its own time-series database. It also provides its query language, PromQL, which allows aggregating, filtering, and performing calculations on the collected metrics. Grafana is a powerful data visualization platform that can connect to various data sources, including Prometheus. Grafana enables users to create customizable and interactive

[*] `https://github.com/nektos/act`

[†] `https://gitlab.com/gitlab-org/gitlab-runner`

[‡] `https://prometheus.io/`

[§] `https://grafana.com/`

dashboards for visualizing metrics, logs, and traces. These dashboards help identify trends, patterns, and anomalies in the monitored data, making it easier for teams to diagnose and resolve issues.

2.4 IT infrastructure

IT infrastructure and technologies have undergone significant transformations in recent years by moving from traditional on-premise data centers to cloud computing and containerization. On-premise infrastructure refers to the physical hardware and software resources that an organization owns, manages, and maintains within its premises. On-premise solutions offer complete control over the infrastructure, data, and security. However, they require investments, ongoing maintenance, and a dedicated IT staff to manage and support the environment. Cloud computing delivers computing resources, such as servers, storage, and applications, over the Internet on a pay-as-you-go basis. It enables scaling IT infrastructure on-demand, reduces initial investments, and offloads maintenance and management burden to the cloud provider. The are three primary cloud service models. Infrastructure as a Service (IaaS) provides virtualized computing resources, such as virtual machines, storage, and networking Platform as a Service (PaaS) offers a platform for developers to build, test, and deploy applications without worrying about the underlying infrastructure. Software as a Service (SaaS) is the highest level of abstraction and delivers ready-to-use software applications. Large cloud providers with the infrastructure and resources to scale their services rapidly and globally are called hyperscalers. Today's top three hyperscalers are Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP) [16]. They offer a wide range of services, including compute, storage, databases, and machine learning. Through redundant and robust infrastructure, they can maintain high levels of availability and reliability, ensuring that their services remain accessible even in the event of hardware failures or extremely high access rates.

2.4.1 Virtualization

Virtualization is a technology that enables the creation of multiple virtual instances of computing resources, such as servers, storage, and networks, on a single physical host. By abstracting the underlying hardware, virtualization allows for better resource utilization, easier management, and increased flexibility. The most common type is server virtualization, which allows to create multiple virtual machines (VMs) on a single physical server, each with its operating system and resources. This is possible through a hypervisor, which is software providing a layer of abstraction between the hardware and the virtualized environments. Typical implementations for server virtualization (type 1 hypervisors) include VMware ESXi, Microsoft Hyper-V, and KVM. VMware vSphere, a centralized management platform, allows managing multiple ESXi hosts and their VMs through a single interface. Storage can be virtualized by aggregating the capacity of multiple physical devices into a single logical pool and network by decoupling services from the underlying physical hardware, enabling the creation of virtual networks, switches, routers, and firewalls. Virtualization has significantly impacted DevOps and CI/CD pipelines by enabling resource optimization, environment consistency, and faster provisioning through virtual machines.

2.4.2 Containerization

Containerization is an alternative approach to virtualization that focuses on encapsulating applications and their dependencies within self-contained, portable units called containers. In contrast to traditional VMs, which include a full operating system and allocate resources individually, containers share the host operating system's kernel and run as isolated processes. This is made possible through a Linux kernel feature called control groups (cgroups). This feature allows limiting, accounting for, and isolating the resource usage (CPU, memory, disk I/O, network, etc.) of a collection of processes [17].
Docker containers are instantiated from images, which are immutable binary snapshots consisting of layered file systems and encapsulating all necessary components, such as run time, system tools, libraries, and the target application code. To create a container, a structured text document with a sequence of commands called Dockerfile defines the base image, dependencies, configurations, and other parameters to construct an image. Container images are lightweight and portable blueprints from which a runnable container can be created. They can be quickly created, replicated, or destroyed, facilitating the horizontal scaling of applications. Therefore containerization is well-suited for implementing a microservices architecture, where applications are broken down into smaller, independent services that can be developed, deployed, and scaled independently.

2.4.3 Container orchestration

Container orchestration is the automated management, coordination, and deployment of containerized applications and services to maintain the desired state of a system while optimizing resource usage and handling failures. This is particularly important in microservices architectures, where multiple containers may need to work together to deliver the desired functionality and where the system needs to be resilient and easily scalable. Kubernetes is the leading container orchestration platform, initially developed by Google. It provides a declarative approach to container management, where the system's desired state is defined using a set of human-readable configuration files called manifests. These manifests describe various resources such as Pods (the smallest deployable units in Kubernetes, containing one or more containers), Services (abstractions that expose applications running on Pods to the network), and Deployments (higher-level constructs that manage the deployment, scaling, and updating of Pods). Kubernetes provides several advanced features for automatic scaling of Pods based on the observed load, rolling updates without downtime, self-healing mechanisms that can restart failed containers or reschedule them to other nodes, and load balancing of network traffic to distribute it evenly among the available Pods [18] [19].

Chapter 3

Related work

3.1 Software focused energy measurements

Measuring the energy of software is complex because many factors influence the results. Luís Cruz gives with his blog post *Green Software Engineering Done Right* [20] a practical guide on how to set up energy efficiency measurements derived from this scientific work in this field. Meaningful measurements must be automated, have a frozen setting, and be repeated often. He also stresses the correlation between the temperature of the CPU and the energy consumption, which fixed breaks between the measurements can address. The gathered data has to be validated first and cleaned from unexpected errors leading to early cancellation and can afterward be used to perform hypothesis testing.

Ardioto et al. outline major issues and guidelines for measuring the energy consumption of software applications in [3]. Physical measurements have the challenge of isolation of the energy consumption of a program when others are running concurrently on the same device, while model-based measurements using resource usage indicators collected at runtime might be only valid for a specific device because the underlying hardware might have a different energy consumption. The authors provide a repeatable process for hardware and software instrumentation, correct synchronization, and sampling frequency.

The authors of [21] combine the areas of Automated Program Repair and Green Software. They emphasize that software developers usually lack information and knowledge about software energy consumption, even though it should be one of the main factors taken into account when designing and developing automated program repairs. After defining energy-related metrics applicable to repair programs, measurements are taken using a physical

Wattmeter. The resulting energy profiles are saved together with timestamps to determine the energy consumption between given time frames. The experiments are conducted on a high-performance computing cluster and follow the methodology and best practices presented in [20].
The different possibilities to improve energy efficiency in the software development life cycle are presented in [22]. These range from typical design patterns, parallel computing, and data structures to programming languages. Energy values are mainly characterized by computations (CPU usage), data management (I/O operations), and data communication (network-bounded operations). These can be indirectly monitored through estimation models, performance counters, or by relying on hardware energy analyzers.

3.2 Low level energy measurements

Intel's Running Average Power Limit (RAPL) interface allows energy measurements by leveraging an on-chip energy model and enabling power management and optimization of CPU cores, integrated graphics, and the memory controller.
The authors of [9] conducted experiments using multiple server platforms and workloads and compared the results from the RAPL interface with those from a high-precision power meter to assess accuracy. They conclude that RAPL can provide accurate measurements for CPU and DRAM power consumption with a high sampling rate of approximately 1000hz and a correlation score of 0.99 with the power meter. Desrochers et al. [10] note that values generated by RAPL are not well documented and that results need to undergo more validation. They monitor the overall system-wide power consumption as well as the CPU power by intercepting the ATX connector of the motherboard and the DRAM by using a DIMM extender. The accuracy of the reported values depends on the processor family but generally lies within 20% physical measurements with a constant offset.
In the work of Gerhorst et al. [23], the authors introduce a method for incorporating external energy measurement devices into existing performance profiling subsystems. They focus specifically on the Linux kernel tool *perf*, which can be utilized to access RAPL counters. The proposed framework integrates external measurement values using a microcontroller into the Linux *perf* subsystem, allowing easy access to perform energy measurements in cases where the RAPL values based on internal sensors are unavailable, are not accurate enough, or do not cover all desired components. Intel's biggest competitor is AMD. Their newer processor generations have a comparable

feature called Application Power Management (APM), while their RAPL implementation should be considered inaccurate [24].

3.3 VM energy measurements

Power consumption of VMs can not be measured directly using wattmeters or sensors on virtualized platforms which are typically used in data centers.
Kansal et al. [25] present a solution called Joulemeter, to enable monitoring and managing the power consumption of VMs. This can reduce energy costs and improve the overall efficiency of the system. Joulemeter tracks the major system resources, namely CPU, memory, and disk, used by a VM running on a host and then translates these into energy estimations based on power models for the individual hardware. The model additionally takes the *IDLE* energy and *shared* system energy into account and achieves good accuracy, with errors around 2% .
BITWATTS is a middleware proposed by [26], which integrates into a generic PowerAPI framework for fine-grained and real-time estimates of the power consumption of software processes. The host operating system (OS) exposes power probes to the guest OS, which uses these values to estimate the power consumption of processes within the VM.
The authors of [27] found a significant difference in the power signature between a VM process and a process running on a physical server. This is because physical machines have idle power, which should be set to zero for a VM, and due to the efficiency of the virtualization layer. Based on these findings, they propose a model to estimate VM power by monitoring the virtualized CPU utilization. A linear regression model is applied when only a single VM is running on a server. With multiple co-located VMs on a server, an iterative algorithm optimizes the model parameters, considering the workloads of all VMs for the power consumption assessment.
Yu et al. [28] group the models for power estimation into three categories: statistical, linear, and nonlinear. They start by determining power signature, which focuses on the impact of CPU, memory, disk, and network resources on power consumption while ignoring hardware-related metrics, like temperature and CPU cache, since they are difficult to measure across heterogeneous platforms. They introduce a power model that employs a Long Short Term Memory using an attention mechanism to capture the interdependent relationships among input metrics. Experimental results demonstrate that their approach achieves a relative error of 2%-5% and provides good generality.

3.4 Container energy measurements

Santos et al. [29] compare the energy consumption of workloads in a container vs. it on the host itself. They could show a slight but measurable overhead for the same workloads running inside a container, which can be attributed to the docker daemon and the run time environment. Despite the measurable difference, they state that the overhead introduced by containers is low and significantly lower compared to VMs.
The paper [30] presents a power monitoring system for containers. They combine RAPL and perf events, similar to the work of Gerhorst et al., to address the trade-off between hardware meter accuracy and software meter granularity. By employing online calibration techniques, the system calculates fine-grained power consumption values and scales them according to coarse-grained measurements. By this approach, the authors reconcile discrepancies between hardware and software meters.
Silva et al. [31] introduce *Containergy*, a profiling tool for containerized applications, utilizing software containers, Dynamic Voltage and Frequency Scaling (DVFS), control groups, and hardware performance counters. Containergy offers low overhead and accurate profiling. It requires a container engine and kernel support and is limited to applications packaged within a container.
cAdvisor[*] is an open-source container monitoring tool that provides real-time resource usage and performance metrics for containerized applications in Docker and Kubernetes environments. It automatically identifies containers, collects container-specific statistics, event notifications, and historical data. The information can be visualized via a web interface or exported to monitoring backends such as Prometheus, InfluxDB, or Elasticsearch. cAdvisor aids in optimizing resource allocation, identifying bottlenecks, and ensuring efficient operation of containerized applications.

3.5 Data center and cloud energy measurements

Zhang et al. [32] propose a decision tree-based regression using operating system metrics as input features to build a power estimation model for VMs. The model can be applied to containers, regarded as a group of isolated processes running inside the VM by substituting the same input features. One

[*] `https://github.com/google/cadvisor`

of the main challenges is power-saving technologies implemented by modern CPU architectures leading to nonlinear power consumption and dynamic scheduling of VMs on the physical server. The model performs well on various benchmarks but fails to reveal the dependencies between counters in operating system metrics and different types of workloads.
The limitations of linear models to capture the dependencies between multiple parameters are recognized by [33]. The authors propose a Support Vector Regression model running on top of the Xen virtualized environment without guest OS modifications (black box approach). It performs better than a linear reference model with consistently lower error rates for multiple tested workloads.
Lin et al. [34] propose an energy consumption measurement system for heterogeneous cloud environments. The authors use hardware-based, virtualization technology-based, energy model-based and simulation-based approaches. The study focuses on the energy contributions of three main hardware components, namely CPU, memory, and disk, and creates separate energy models for them. The system has a Master-Slave architecture, with the Master responsible for aggregating statistics and displaying cluster-wide energy consumption while the Slave monitors CPU, memory, and disk usage. The mean relative estimation error remains minimal at 2.39% when subjected to mixed workloads in heterogeneous cloud computing environments.
The blog post *Estimating AWS EC2 Instances Power Consumption* [35] introduces a method for converting EC2 usage metrics into kilowatt-hours (kWh). Elastic Compute Cloud (EC2) are virtual servers provided by Amazon Web Services (AWS) available in different configurations. Initially, the scaling and underlying hardware of the different instances are analyzed. Subsequently, results from the standardized specPower benchmark, indicating power consumption at varying load levels, are used for servers with comparable hardware configuration [36]. The usage metrics are based on RAPL values, which have been successfully accessed in EC2 instances by [9]. On newer EC2 instances employing KVM virtualization, RAPL might not be applicable since underlying registers are not available, which limits measurements to bare metal instances. Based on the gathered data, the authors present a table displaying the power consumption of typical EC2 instance sizes in relation to vCPU usage. Key insights include the influence of external factors, such as varying CPU generations and thermal conditions, on power consumption. Moreover, power consumption is not solely dependent on CPU utilization and memory usage but also on the nature of the processed workload.
Ismail et al. [37] present a taxonomy of software-based power models,

distinguishing between linear and nonlinear models. Each category can be further divided into mathematical models with fixed slope and intercept and machine learning models with variable slope and intercept. The authors conduct a performance analysis, comparing the software-based power models by evaluating their standard estimation error. They observe a nonlinear relationship between CPU utilization and power, which linear models inadequately represent. Similarly, linear models struggle to capture the correlation between features. The authors emphasize the importance of differentiating between workload types. The study incorporating additional features, such as memory, disk, and network usage, can enhance the accuracy of the power models, especially on less CPU-dominant workloads.

3.6 CI/CD pipeline energy measurements

Currently, there is still a lack of research combining energy measurements with the emerging field of DevOps and the associated CI/CD pipelines *Incorporating energy efficiency measurements into CI/CD pipeline* [38] presents an approach to recognize bottlenecks in the structure and run-time behavior of software by integrating energy measurement directly into CI/CD pipelines. The first part of the presented system uses static analysis of the source code, and the second part performs an analysis of the power consumption of the running application. For the measurement of energy consumption, the resource consumption is tracked based on the time frequency of the respective processes, similar to the power consumption management implemented in most operating systems. This paper uses CI/CD pipelines to create a closed-loop system for continuous analysis and optimization, but the energy consumption of the pipelines themselves is left out.
The *ECO CI* [39] project aims to make the energy usage of Github Action pipelines more transparent. They developed a small executable that writes the CPU usage of the hosted cloud runner into a text file. Afterward, they load a pre-trained gradient boosting model to map the recorded CPU values to power values. The user receives feedback about the consumed energy via the run report at the end of a pipeline run. This makes it relatively easy to instrument a pipeline without additional external tooling. On the downside, the instrumentation and the loading machine learning model consume a relatively large amount of additional resources, especially for smaller pipelines. The authors do not make any specific statement about the accuracy of this tool.

Chapter 4

Energy measurements of pipeline runners

This chapter focuses on answering the research question: *Which existing approaches can be used for transparent and granular energy measurements of CI/CD pipelines?* To determine the energy consumption of a CI/CD pipeline run, we need to calculate the power consumption of the runner used. Chapter 3 provides an overview of previous research on energy measurement methods at different levels. We divide these into hardware and software-based. For each of these classified methods, the following protocol is applied:

1. Theoretical analysis: explore how the measurements are created.
2. Practical analysis: identify necessary steps and existing tools.
3. Evaluation: Assess suitability to determine the power consumption of CI/CD runners, using the following criteria:
    a) Accuracy: how precise the energy measurements are compared to a reference measurement.
    b) Granularity: the level of detail of the measurements, such as whether they can be taken for individual processes or only for the device as a whole.
    c) Sampling frequency: the rate at which measurements can be taken.
    d) Overhead: the impact of the energy measurement method on the system under test.
    e) Applicability: the feasibility of integrating the method into CI/CD runners.

4.1 Hardware based metrics

4.1.1 External measurements devices and sensors

Theoretical analysis

Physical devices, like wattmeters or power supply sensors, measure the product of voltage (V) and current (I) across an electrical circuit and then multiply it by the power factor to obtain the real power (P) in watts ($P = VI \cdot \cos\theta$). The power factor $\cos\theta$ is a dimensionless number between -1 and 1 that represents the efficiency of an AC electrical circuit by quantifying the relationship between the actual power consumed for work and the apparent power oscillating between the source and the load [40]. Wattmeters are installed between the host machine's power supply and the electrical outlet to measure the system's energy consumption. In contrast, power supply sensors are integrated components of server hardware used to monitor power usage, input voltage, current, temperature, and other power-related metrics. The power supply sensor is typically reachable through a controller running on an independent chip, separate from the server's primary resources [41].

Practical analysis

One popular wattmeter is the Watt Up? Pro which has high reliability and accuracy [8]. Values are stored in the internal data logger and can be exported for analysis. Within the scope of this study, experiments are conducted with Dell PowerEdge servers. These servers are equipped with an Integrated Dell Remote Access Controller (iDRAC) offering a REST API that enables accessing the current power consumption of the server, allowing the retrieval of power consumption values in real-time. Other server manufacturers offer comparable solutions. HP offers Integrated Lights-Out, Lenovo XClarity Administrator, and Fujitsu ServerView Suite. Measuring the power consumption of a server during the execution of a CI/CD pipeline requires setting up a physical measurement system or a data logger for the power control chip. Consequently, the server's idle power consumption must be determined by collecting samples during a period without running any pipelines. During the pipeline execution, only a single runner should be running to avoid skewed measurements. After execution, the server's idle power is subtracted from the measurements, and the energy consumption of the pipeline run is calculated using either the weighted-average or interval-based method as presented in chapter 2.

Evaluation

The advantage of using dedicated hardware is the high accuracy. The Watt Up? Pro reports a measurement accuracy of ± 1.5%, a claim that is supported by research [8]. Similarly, Dell's PowerEdge servers boast a 1% accuracy for their power supply unit (PSU) power monitoring capabilities [42]. The main limitation of physical measurements is the coarse granularity. When multiple runners are concurrently operating on a server, physical measurement methods cannot break down the power consumption by individual runners. In terms of frequency, both the Watt Up? Pro and the Dell power chip provide measurements at 1 Hz. Note that querying power values from the integrated power sensors at high frequencies could overwhelm the dedicated chip, posing a risk to the server's regular operation. Hardware-based measurements do not introduce additional overhead on the system under test. The primary prerequisite for these methods is having access to the servers' power supply or sensors, making them suitable only for self-hosted runners. External measurements resemble an experimental approach, ideal for ascertaining or validating energy consumption but not applicable for larger systems or continuous monitoring.

4.1.2 Running Average Power Limit

Theoretical analysis

Intel's Running Average Power Limit (RAPL) is a hardware technology that delivers highly granular power samples at a rapid sampling rate. Incorporated into modern Intel processors, this technology allows users to track and regulate CPU, GPU, and DRAM power consumption. RAPL employs internal power counters and a proprietary power model to estimate the energy usage of the processor and its related components. Intel has not disclosed which counters are utilized and how the corresponding power value is computed [43]. While RAPL is essentially a hardware feature, it necessitates using specific tools or libraries to retrieve the calculated values [44].

Practical analysis

There are multiple ways to read RAPL values in Linux. The Linux principle *everything is a file* encapsulates the idea that aspects such as input/output, inter-process communication, and networking are managed as byte streams represented through the file system [45]. Consequently, RAPL values can be obtained from the powercap interface by reading files located within */sys/class/powercap/intel-rapl/*. Alternatively, these values can be accessed via the *perf_event* interface or through raw access to the underlying Model Specific Registers (MSRs) found under */dev/msr* [10]. In Windows environments, RAPL values can be accessed using the *Intel Power Gadget* or the *Windows Management Instrumentation*. Scaphandre is a powerful new monitoring agent tool written in Rust. Based on the RAPL measurements, representing the power used by the system, it calculates the power used by a single process based on the jiffies counter. Jiffies are small intervals of time where the processor is allocated to work on a given job. The energy can be calculated by checking for each jiffy and how much power is drawn at those moments in time [46].

Hypervisors provide virtual hardware to the VMs, which abstracts the underlying hardware. As a result, direct access to hardware-specific RAPL features is not available. Some hypervisors employ pass-through to expose specific hardware features to the VMs, granting direct access to the hardware and enabling the VMs to read RAPL values. Scaphrandre uses this approach to support kernel-based virtual machines [46]. While Docker containers share the same kernel as the host system, accessing RAPL values may require elevated permissions. Docker containers can be run in privileged mode to allow access, or the *sys/class/powercap* files can be mounted inside the container to access the RAPL values.

Evaluation

Several studies demonstrate the reasonable accuracy of RAPL values, adding a level of confidence in its utilization [10] [9] [23]. RAPL measurements offer a comprehensive view of the system's power usage and energy consumption or specific hardware components. Tools like Scaphandre further enrich this data by enabling the decomposition of energy values, providing more granular insight. The RAPL hardware feature delivers high-frequency sampling, up to 1000 Hz, which aids in precise energy consumption measurement. Being a hardware feature, RAPL does not induce overhead for generating the values. However, a computational cost is associated with querying these values via software running on the tested system. This overhead can become significant, primarily when operating at the maximum 1000 Hz frequency. Therefore, using an efficient tool like Scaphandre to minimize this impact is crucial. RAPL's use is restricted to specific CPUs, notably Intel (Sandy Bridge or newer) and a limited range of AMD processors. Moreover, its functionality is confined to specific power domains, including CPU and DRAM. Its utilization in most VMs and containers tends to be complex, with passthrough methods serving as a less-than-ideal solution. Furthermore, it is incompatible with contemporary cloud instances such as EC2, which utilize advanced virtualization approaches like the XEN hypervisor. As a result, the usefulness of this feature in the context of DevOps runners is typically restricted.

4.2 Software based metrics

The calculation of energy values using software-based metrics consists, on the one hand, of the collection of the relative resource consumption and, on the other hand, of using these values together with the energy model to estimate associated energy values. The related work presents various models to convert the resource consumption of VMs or containers into power values applicable to this use-case [25] [27] [28].

4.2.1 Agent based measurements

Theoretical analysis

This section considers additional measurements conducted by an agent. Operating systems employ different strategies to track and evaluate components' relative resource consumption, such as the CPU and memory. For CPU usage, the operating system maintains a record of the duration each process spends executing on the CPU. The relative CPU usage can be determined by calculating the fraction of the total CPU time a process consumes over a defined time interval. With regard to memory usage, every process has a corresponding address space, a portion of which is allocated for storing the process's data and code. The operating system creates a page table for each process, establishing a map between the process's address space and the physical memory. By traversing a process's page table, the operating system can ascertain the quantity of physical memory the process utilizes.

Practical analysis

Most operating systems come with native tools for monitoring resource usage. On Linux systems, information about the system-level and process-level information can be read from the virtual *proc* filesystem, providing an interface to access kernel data structures. This interface calculates the CPU time by examining the process's utime (user mode time) and stime (kernel mode time) values, representing the number of clock ticks the process has spent. The relative CPU usage for each process can then be calculated by the difference in CPU time between two successive samples divided by the total CPU time difference (including idle time) between the two samples. Windows offers the performance monitor and resource monitor as default tools and the Windows Management Instrumentation API. Agents can be added to run inside a VM or Docker image and collect these metrics.

Evaluation

The employed methodology offers a very good estimation of resource consumption. However, the accuracy of the derived energy values is predominantly dependent on the energy model utilized to convert resource usage into energy values. The actual usage may deviate due to influences like system overhead and potential inaccuracies in measurement. This approach offers high granularity at the process level. On Unix-based systems, information on individual processes can be obtained through the */proc/[pid]/stat* file dedicated to each process. The */proc/stat* file delivers real-time kernel statistics and does not adhere to a fixed update interval[47]. Accessing data from */proc* entails utilizing system resources, as the kernel must allocate CPU time to generate this data. This approach is independent of the underlying hardware and can easily be integrated into a VM or a container.

4.2.2 Exposed measurements

Theoretical analysis

This section considers measurements executed by the underlying technology exposed via API which can be used for energy measurements. Type 1 hypervisors run directly on the host's hardware and oversee the operation of VMs to maintain an intimate connection with the hardware. They deploy a CPU scheduler for the fair and efficient distribution of CPU and memory resources. Container solutions, like Docker, determine resource usage by inspecting the CPU and memory usage statistics supplied by cgroups. Docker computes the relative utilization by comparing the alteration in CPU consumption for a specific container over time against the overall change in CPU time on the host [48].

Practical analysis

Within the scope of this study, we took a closer look at the bare-metal hypervisor ESXi from VMware. ESXi runs on the individual server, while vSphere is one abstraction level higher and can monitor an entire data center. While ESXi allows for a higher sampling rate, vSphere allows an entire data center consisting of many ESXi instances to be queried via an API. VMware provides libraries simplifying working with the API. The docker stats command is a utility provided by Docker that allows users to fetch and display real-time statistics about running Docker containers. It operates as a continually running daemon, gathering, consolidating, processing, and disseminating information about active containers. When executing docker stats in the command line, it provides a live data stream for CPU usage, memory usage, network I/O, block I/O, and PIDs for all running containers. cAdvisor (container advisor) is a popular container monitoring solution from Google for multiple containers or Kubernetes clusters [49]. Hosted runners expose data collected by the underlying hypervisor, like Amazon's Elastic Cloud Compute (EC2), Google Compute Engine, or Azure Virtual machines. All three can be queried directly via a REST API, but also provide clients in different languages to facilitate queries.

Evaluation

Hypervisors and container solutions can provide accurate resource metrics values, with process-level granularity available for virtual machines and containers [48]. The ESXi hypervisor, samples CPU usage of virtual machines every 20 milliseconds, providing near real-time information [50]. Meanwhile, Docker's *stats* command provides real-time statistics. One of the key benefits of this approach is that it typically does not impose additional overhead on the system being measured, which preserves its original performance characteristics. This method is highly applicable for DevOps runners, which are typically executed as VMs or containers. Its flexibility and the absence of a need for additional instrumentation make it particularly suitable for instrumenting CI/CD runners.

4.3 Summary

The following table rates the criteria on a scale ranging from low to high. It is important to note that this is a comparative evaluation, not an absolute measurement.

Table 4.1: Relative comparison of energy measurement methods

Method	**Accuracy**	**Granularity**	**Frequency**	**Overhead**	**Applicability**
Physical devices	high	low	low-medium	low	medium
Intel RAPL	medium	high	high	medium-high	low
Agent	medium	high	high	medium-high	medium
Exposed	medium	medium	low-medium	low	high

Hardware-based solutions to measure the energy of CI/CD pipelines provide accurate results and have a low overhead. Software-based solutions allow higher granularity and are easier to integrate.

Chapter 5 Planetary framework

This chapter addresses the research question: *How can energy measurements be efficiently integrated into existing infrastructure used to run CI/CD pipelines?*. Thereby, there should be little to no overhead for the runners executing the pipelines no changes required on the pipelines to be monitored. As highlighted in Chapter 3, we can not rely on existing approaches in the scientific literature. To answer the research question, we designed and developed a generic framework called *Planetary* using the following steps:

1. Conceptual architecture of the framework: identifying the necessary components.
2. Implementation: developing the components.
3. Integration: understanding how the components can be incorporated into a state-of-the-art infrastructure.

The collected data and results are presented and analyzed in Chapter 6.

5.1 Architecture

To assess the energy consumption of CI/CD pipelines, gathering and integrating data from various sources is necessary. Figure 5.1 presents a high-level architecture of the components in this framework.

5.1.1 Log analyser

DevOps platforms allow to create, store, and trigger pipelines, which are then executed on runners. The platform saves information about the pipeline's

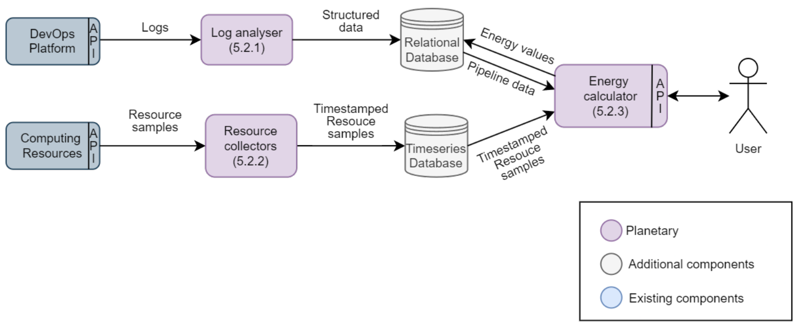


Figure 5.1: High level framework architecture

execution duration, the jobs and steps performed, and the used runner in logs. The logs are accessible through its API. The log analyzer is the first component responsible for communicating with the DevOps platform. It fetches the required data on-demand to evaluate specific runs or periodically, to obtain an overview and enable continuous monitoring. After retrieving the data, it is parsed and stored in a database. Both relational and NoSQL databases are valid options. NoSQL offers greater flexibility since no data model needs to be defined in advance, while relational databases enable more efficient querying and improved analysis by utilizing schemas and optimized data organization.

5.1.2 Resource collector

The resource collector is the second component and is responsible for providing data on the resources used by the runners. It collects power samples from physical devices or sensors and resource metrics, including, but not limited to, CPU, memory, and network usage from the runners. The collected data is stored in a database to subsequently determine the resource consumption for specific time windows. A time-series database is best suited for handling metrics and events with timestamps.

5.1.3 Energy calculator

After extracting pipeline run data and resource metrics of their runners, the energy calculator is responsible for combining this data. Given the run-time data of a pipeline on a particular runner and the average power the runner has consumed during this time, we can calculate the energy consumption consumed by that runner. Therefore, the energy calculator queries the log analyser database to retrieve the start and end time and the used runner and the time-series database to retrieve resource usage or power samples. Power samples can be used directly while we apply an energy model for relative resource samples. Additionally, this component is tasked with making the calculated values available. This could be achieved by writing the values back into a database, presenting them visually, or exposing them through an API.

5.2 Implementation

5.2.1 Log analyser

The log analyzer collects information about CI/CD pipelines by accessing the pipeline logs via the API of the DevOps platform.

Data model

To store the information contained in to the logs, we derive a common data model applicable to common DevOps platforms (like Github, Gitlab, and Azure DevOps) by analyzing pipeline definitions. The relations of the data model are shown in Figure 5.2. In CI/CD pipelines, workflows are specified by the pipeline definitions stored within the repository, which outline the sequence of tasks to be performed. Each execution instance of a given workflow is referred to as a run. Within a workflow, multiple jobs are composed of several steps. Jobs represent distinct work units executed on runners, the computing resources responsible for carrying out the tasks specified in the pipeline. Different jobs of one run can be executed by different runners. On some platforms, jobs are further categorized into stages. However, this aspect is neglected in our data model as GitHub, in its current version, does not provide support for stages. An exemplary Github Actions pipeline used for this project can be found in Listing A.1 (Appendix).

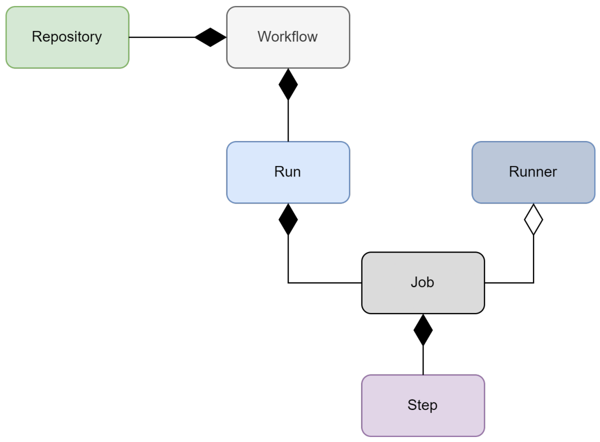


Figure 5.2: Data model of CI/CD pipeline components

Data collection

Data collection methods from the API can be categorized as either pull-based, which involves querying at specified intervals, or push-based, which leverages webhooks to trigger data collection following certain events. We engineered a pull-based microservice, to avoid the additional configuration of the webhooks in the individual repositories. Algorithm 1 shows the main steps executed by this service. It recurrently queries the respective API for each recorded repository to detect if new pipeline runs have been executed (Line 5). We record when the repository was last analyzed to prevent redundant work (Line 7). In addition, we store the platform for each repository to use platform-specific API and parser. We conduct further requests for each newly detected pipeline run to gather more detailed information about each run, such as jobs and steps (Line 9). The response from API is parsed and then stored in the relational database (Line 11).

This tool is designed generically, facilitating easy integration of platforms by implementing the abstract API and parser classes. In the context of this study, we developed API and parser implementations for Azure DevOps and GitHub Actions, since these two are used in the environment of this thesis. We use the *PostgreSQL*, a powerful open-source relational database management system popular due to its wide range of features, robustness, scalability, and compatibility. The code of this service is available on Github*.

*`https://github.com/nikolim/log_analyser`

Algorithm 1 Log analyzer main loop

```
Output: Pipeline information stored in database
1  Initialize: connect to relational database db
   for True do
2      repositories ← db.getRepositories()
3      for repo in repositories do
4          api, parser ← create objects based on platform of the repository
5          runsRaw ← api.getRunsSinceLastCrawl(repo)
6          runs ← parser.parseLogs(runsRaw)
7          sqlDb.setLastCrawl(repo, now)
8          for run in runs do
9              jobsRaw, stepsRaw, runnersRaw ← api.getDetails(run)
10             jobs, steps, runners ← parser.parse(jobsRaw, stepsRaw, runner-
                 sRaw)
11             sqlDb.store(jobs, steps, runnners)
12         end
13         sleep($n$ seconds)
14     end
15 end
```

5.2.2 Resource collectors

The task of this component is to collect power samples and resource metrics from the runners used for the CI/CD pipelines. This allows us to calculate how much energy a runner consumes in a given time frame. As discussed in Chapter 4, different strategies are applicable. The main focus of this framework is to create a scaleable measurement system with as little overhead as possible, where no change is required in the monitored pipelines or the used runners. VMs running on an on-site data center are used as runners in the test environment for this work, as shown in Figure 5.3. Therefore, we rely on exposed measurements, collecting metrics at the virtualization and hardware/server layers. This allows us to retrieve power samples of the server and use the relative resource consumption to decompose the power consumption into the different VMs.

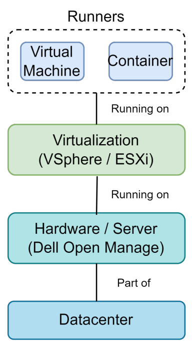


Figure 5.3: Layers of datacenter computing resources

Pollect is an open-source daemon for collecting system and application metrics in periodical intervals. After collecting, it exposes the metrics in them in a *Prometheus*-compatible format (introduced in Chapter 2) [*]. We use it to periodically collect information from the hardware/server and virtualization layer. To collect the required data, we extended the data sources of *pollect* to support the technologies in our test environment, namely VMware vSphere and Dell Open Manage. Using the vSphere API, we gather CPU, memory, network, and disk data from the hosts and VMs. vSphere also allows to capture the power of hosts, and we are using a more recent version which also provides power values for VMs. As the software is proprietary, it is unclear which model is used to determine the power of single VMs. Samples from the power supply chip for each server are collected using the Dell Open Manage API, which can be used for monitoring and management of multiple Dell servers. We found that the power values of vSphere and Dell Open Manage match. Thus vSphere can access the hardware sensors of the respective servers, making the measurements from Dell Open Manage obsolete.

The respective implementations can be found on GitHub [†‡]. Using a different hypervisor or another server manufacturer requires adding an additional data source to *pollect*, while the general approach is identical.
We observe that the data collection of all servers and virtual machines simultaneously may not be feasible, particularly for larger IT infrastructures. Most APIs are rate limited and, therefore, may not be able to provide all information for all hosts within a reasonable time. Consequently, we have

[*] `https://github.com/davidgiga1993/pollect`

[†] `https://github.com/nikolim/pollect/blob/main/pollect/libs/vmware/vsphere.py`

[‡] `https://github.com/nikolim/pollect/blob/main/pollect/libs/dell/openmanage.py`

chosen to implement selective monitoring of servers by performing an inner join between runners already used to run pipelines and all available machines. This allows the selected machines to be queried at a higher sampling rate without reaching the limits of the API limiting. The tradeoff is that when new VMs are started up that do not yet appear in any pipeline logs, they are not monitored, and therefore no statements can be made about the power consumption.
Prometheus is responsible for managing the collection and storage of data. The instance running *pollect* must be added as a scrape target, enabling *Prometheus* to periodically query and store the results in its internal time-series database. It is important to note that Prometheus, by default, saves values with the scrape timestamp in the database, making it necessary to expose measurement timestamps in order to prevent a time delay.

5.2.3 Energy calculator

This component is responsible for calculating the energy consumption of pipelines and relies on the data procured from two introduced components (Section 5.1.1 and Section 5.2.2). By default, energy calculations are performed on a per-job basis, requiring the determination of the job's duration and the runner's average power employed during the job execution. The duration of the job can be directly obtained from the parsed job stored in the database, whereas acquiring the average power requires querying the time series database within the time frame of the job's start and end time. If the obtained samples are power samples, they can be utilized directly. Conversely, if the samples are resource samples (like CPU, memory, disk, and network usage), a power model is employed to convert these resource samples into power samples. The power model is dependent on the available data and the underlying technology. Chapter 3 presents different solutions for power estimations for VMs [27] [28], containers[30] as well as cloud environments [34] [35]. In the scope of this thesis, we explore different machine-learning algorithms for power estimation and decomposition. We use proven machine learning algorithms from *Scikit-learn* for this purpose. *Scikit-learn* is a popular Python library for machine learning, offering a wide range of algorithms and tools for data preprocessing, model training, evaluation, and optimization*. They make this framework applicable if the virtualization or container solution exposes only relative resource samples instead of power samples. Results for power predictions using a *Random Forest Regressor* and

*`https://scikit-learn.org/`

CPU and memory as input used can be found in the Figure C.5 (Appendix). Subsequently, the exposed or estimated power samples are utilized to compute a weighted power consumption average and total energy consumption using Equation 5.1 and 5.2.

$$P_{avg} = \sum \frac{(P_{sample} \times \Delta T)}{T_{total}} \tag{5.1}$$

$$\mathrm{E} = \frac{(\mathrm{P} \times T_{total})}{3600} \tag{5.2}$$

P_{avg}	= weighted average power (watts)
P_{sample}	= weighted average power (watts)
ΔT	= Time between the samples (s)
T_{total}	= Total time span of the run (s)
E	= Energy consumed by the run (Wh)

The same methodology, as presented in algorithm 2 for jobs, can be applied to steps to calculate even more fine-granular results. Instead of the jobs, the start and end time of the step is determined. Algorithm 3 summarizes how to determine the energy consumption of a specific pipeline run with a given ID. Initially, the run is retrieved from the relational database (Line 29). The cumulative energy consumption of the run is then derived from the summation of all individual jobs (Line 34).

Planetary makes energy values accessible through a REST API. We used *FastAPI*, a high-performance web framework used to create reliable and production-ready APIs *. The emissions endpoint, denoted as */emissions/{repositoryId}*, enables the determination of energy and emission values associated with a specific repository within one of the predefined time frames (last hour, last day, last week). For a more comprehensive analysis, the details endpoint, represented as */emissions/details/{runId}*, can be employed, which also includes the steps of each job, offering more fine grained results. The badge endpoint, indicated by */badge/{repositoryId}*, yields a support vector graphic (SVG). By incorporating these SVGs into a README file, emission values can be effectively reported back to both developers and users of a repository, as illustrated in Figure 5.4. Additional to the API, we run a service in a given interval, calculating the energy values for jobs and writing the results to the relational database. This prevents the repeated calculation

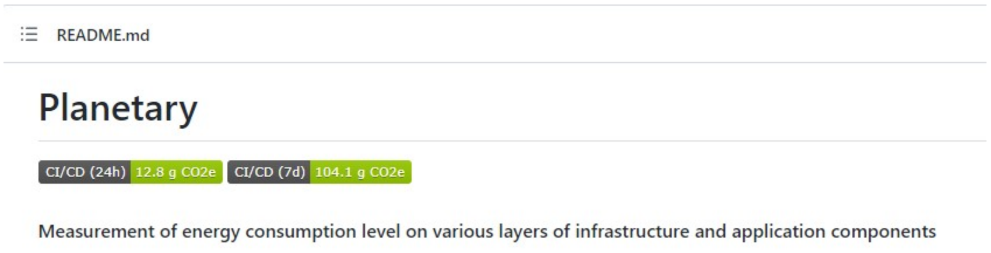
*https://github.com/tiangolo/fastapi

Algorithm 2 Calculate energy for job

```
Input: job
Output: energy, runTime, averagePower
Initialize: connect to timeseries database tsDb
if vmPowerAvailable then
    powerSamples $\leftarrow$ tsDb.getPowerSamples(job.runner, job.start, job.end)
end
else
    // use model to predict power consumption
    resourceSamples $\leftarrow$ tsDb.getResourceSamples(job.runner, job.start,
      job.end)
    powerSamples $\leftarrow$ powerModel.predict(job.runner, resourceSamples)
end
averagePower $\leftarrow$ calcWeightedAverage(powerSamples, job.start, job.end)
runTime $\leftarrow$ job.end $-$ job.start
energy $\leftarrow$ $\times \cdot$ runTime
```

Algorithm 3 Calculate energy for run

```
Input: runId
Output: energy, runTime, averagePower
Initialize: connect to relational database sqlDb
run $\leftarrow$ sqlDb.getRun(runId)
foreach job in run do
    energy, runTime, averagePower $\leftarrow$ calculateEnergyForJob(job)
    store(energy, runTime, averagePower)
end
averagePower $\leftarrow$ calculateWeightedAveragePower(runTimes, averagePowers)
runTime $\leftarrow$ sum(runTimes)
```


README.md

Planetary

CI/CD (24h) 12.8 g CO2e CI/CD (7d) 104.1 g CO2e

Measurement of energy consumption level on various layers of infrastructure and application components

Figure 5.4: Example Badge embedded inside a README file

of energy consumption values and enables energy values to be queried and organized more effectively.

5.2.4 Emission calculation

Based on the power consumption and the runtime of pipelines, we can calculate CO_2 emissions. Apart from energy consumption, the emissions depend on the Power Usage Effectiveness (PUE) and the carbon density. The PUE metric quantifies the extra energy required for ancillary functions such as cooling and network devices. A PUE of 1.0 is optimal, indicating that all consumed energy is used for computation, while older and less efficient data center have PUE values over 2.0. In Germany, the average PUE for data centers was reported at 1.56 in the year 2021 [51]. Google reported an average PUE of only 1.1[52].
The carbon density specifies the CO_2 emissions per kilowatt-hour (kWh). This value is dependent on the used electricity mix. For instance in Germany, the carbon density in 2021 was, in average 349 grams of carbon dioxide per kilowatt-hour (gCO_2/KWh) [53]. Together, the emissions can be calculated as follows:

$$E = \frac{P \cdot t}{3600 \cdot 1000} \cdot PUE \cdot d \tag{5.3}$$

where:

E	$=$ $C0_2$ emissions in gram
P	$=$ be the power consumed in watts
t	$=$ be the runtime in seconds
PUE	$=$ be the unit less Power usage effectiveness ratio
d	$=$ be the CO2 density in gram per kWh

5.2.5 Integration into an existing infrastructure

This section describes the integration of the presented components into Siemens' IT infrastructure. This configuration can be adapted and replicated with minor modifications across most contemporary infrastructures. The architecture of all components integrated into the existing IT infrastructure is shown in Fig. 5.6. We focused on self-hosted runners operating within an on-site data center. Owing to network policies, we deployed the resource collectors as virtual machines (VMs) in the same data center.

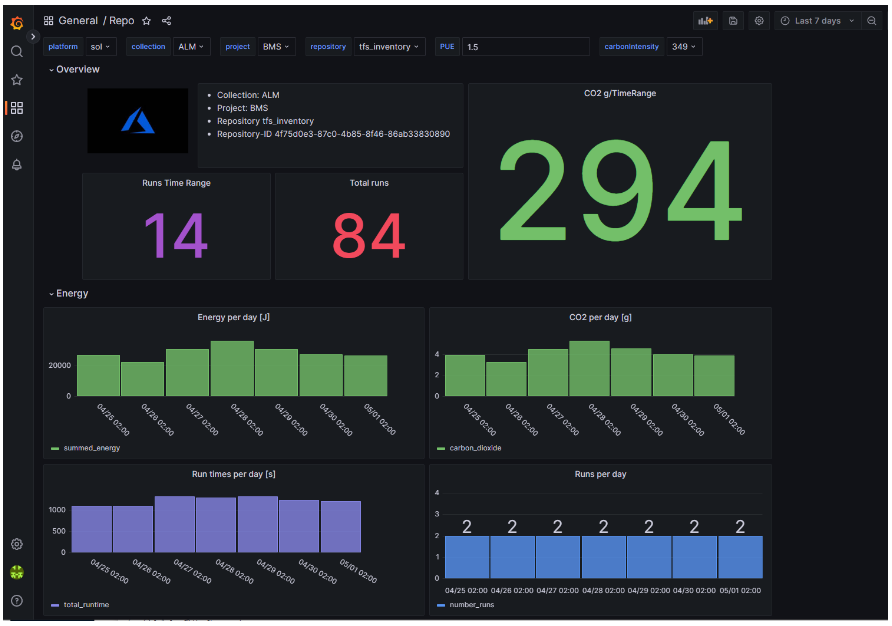
All remaining components (log analyser, energy calculator as well as the database) are deployed in the cloud in a managed Kubernetes cluster. For the log analyzer and energy calculator, we create docker containers, publish them to a container registry, and then deploy them to the cluster. Moreover, a PostgreSQL database is provisioned to store the results generated by the log analyzer. We automated the process using CI/CD pipelines*. This approach facilitates the effortless upgrading and scaling of the application as needed.

Furthermore, a Prometheus and Grafana instance is deployed within the same cluster. Prometheus extracts resource consumption data from the resource collectors and stores these values in its internal time series database. Note that, by default, Prometheus preserves this data in memory for two hours and on disk only for 15 days [54]. Grafana is used for monitoring with Prometheus and PostgreSQL as data sources. Grafana enables the selection of a specific time frame for which the values are to be displayed. An example dashboard for monitoring the total quantity of CO2 emissions generated by all pipelines for a selected repository within the selected time frame is depicted in Figure 5.5. This dashboard also presents the number of pipeline runs, the aggregate runtime, and the associated consumed energy.

5.3 Summary

We created a flexible framework by utilizing existing logs from the DevOps platforms and leveraging the API provided by the runner's underlying technology. It seamlessly integrates with contemporary infrastructure, making it simple to scale and deploy without requiring changes to the runners or CI/CD pipelines.

*`https://github.com/nikolim/planetary/tree/main/.github/workflows`



General / Repo

platform: sol · collection: ALM · project: BMS · repository: tfs_inventory · PUE: 1.5 · carbonIntensity: 349 · Last 7 days

Overview

- Collection: ALM
- Project: BMS
- Repository tfs_inventory
- Repository-ID 4f75d0e3-87c0-4b85-8f46-86ab33830890

Runs Time Range: 14

Total runs: 84

CO2 g/TimeRange: 294

Energy

Figure 5.5: Planetary dashboard implemented with Grafana: Energy consumption of all CI/CD pipeline runs for a selected repository

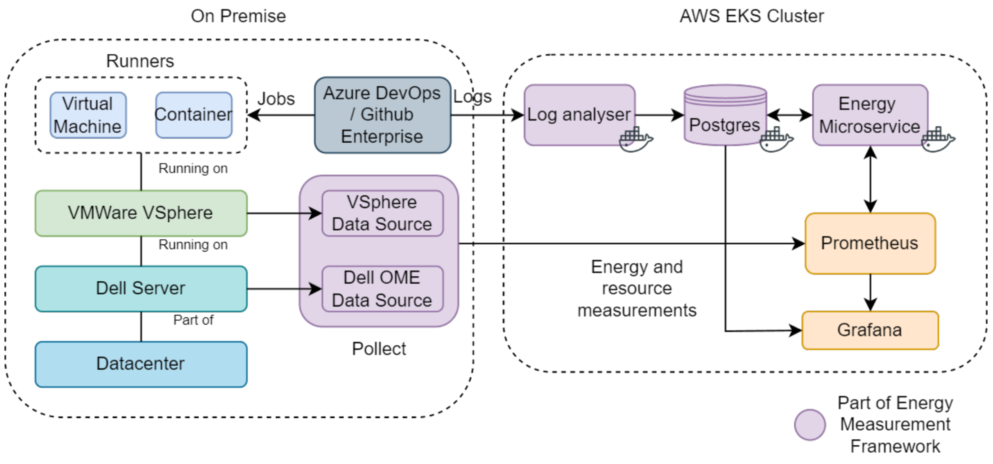


Figure 5.6: Architecture overview in the Siemens infrastructure

Chapter 6

Experiments

This chapter applies Planetary to a modern IT infrastructure to record and analyze the energy consumption of CI/CD pipelines. The goal is to answer the question: *Into which general categories can jobs and steps executed in CI/CD pipelines be divided, and how do these categories contribute to energy consumption?* We take the following steps to record data:

1. Selecting an appropriate test environment: We choose a real-world test environment where CI/CD pipelines play an essential role in the software development process.

2. Configuration of Planetary: The framework must be adapted and configured based on the given technologies and infrastructure.

3. Collecting data: Record all CI/CD pipeline runs for a specified period and determine their energy consumption.

The recorded data provides the basis for answering the research question, which we divide as follows:

- **RQ 3.1** *What are the general categories into which jobs and steps that are executed in CI/CD pipelines can be grouped?* We apply post-processing on the recorded data to divide the jobs and steps into subcategories. DevOps pipelines can execute arbitrary commands, scripts, and binaries or use predefined workflow actions. This can lead to a remarkable diversity in their designations, making it challenging to analyze their exact responsibilities. We first form categories by extracting all the names assigned to the jobs and steps, calculate their frequency, and then form categories based on the most commonly occurring words. Consequently, each job and step is assigned a category based on its name using natural language processing metrics.

- **RQ 3.2** *What is the energy consumed by jobs and steps in these different categories?* Planetary calculates energy values for jobs and steps in a specific interval and writes them into the relational database. We group the jobs and steps based on their categories and then analyze the energy consumption per category.

6.1 Test environment

We analyze the CI/CD pipelines of two Azure DevOps instances. The first instance is primarily employed for tooling, internal services, and blueprint pipelines for various projects and hosts 942 repositories. The second instance, in contrast, is utilized for developing large software projects and services and hosts 1565 repositories. The pipelines from the first instance are often simpler and characterized by their fast execution (157 seconds on average). In contrast, the pipelines associated with the second instance exhibit more complexity, involving more steps and a considerably longer run time (885 seconds on average). Cumulatively, these two platforms have a total of 2,507 repositories. The runners of these pipelines are running on servers listed in table 6.1. These VMs on these servers are allocated resources ranging from 1 to 32 CPUs and between 1 to 256 GB of RAM. The operating system of these VMs used as runners include versions of Windows (7, 8, 10, Server 2016, and Server 2019) and a selection of Linux distributions (Debian, CentOS, SUSE, and Ubuntu).

Model	**CPU**
Dell PowerEdge R630	Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz
Dell PowerEdge R640	Intel(R) Xeon(R) Gold 6248R CPU @ 3.00GHz
Dell PowerEdge R750	Intel(R) Xeon(R) Platinum 8362 CPU @ 2.80GHz
Dell VxRail P570F	Intel(R) Xeon(R) Gold 6248 CPU @ 3.00GHz

Table 6.1: Servers used in the experiment test environment

6.2 Configuration of Planetary

Log analyser configuration

The Log Analyzer is designed to inspect repositories' CI/CD runs. Therefore, all repositories to be analyzed are initially added to the relational database. New repositories can also be added during the operation of the service. The variable *repo_scan_frequency* determines the frequency of querying new repositories from the database, in our case 6 hours (21600 seconds). New pipeline runs are requested for every repository at intervals specified by the *run_scan_frequency* variable, which we set to 1 hour (3600 seconds). We employ a thread pool to parallelize the service to expedite this process. The thread pool size can be adjusted using the *max_threads* variable, enabling simultaneous querying of multiple repositories and concurrent writing into the database. We noted a substantial performance enhancement when using eight threads instead of a single thread in our testing environment. However, increasing the thread count beyond this yielded little to no further gain in performance. The values for these parameters are summarized in Table 6.2.

Key	**Value**	**Description**
max_threads	8	Threads determining the level of parallization
repo_scan_frequency	$21600s$	Interval in seconds to check for new repositories
run_scan_frequency	$3600s$	Interval in seconds to check new runs

Table 6.2: Test environment: Log analyser configuration

Pollect resource collector configuration

We utilize *pollect* (introduced in 5.2.2) to gather power samples from the virtualization and hardware layers. The frequency of data retrieval can be customized using the *tick_time* parameter. In our case, *pollect* requests and exposes new data points from its targets every 60 seconds. Prometheus is then responsible for collecting these values. Setting a higher sampling frequency (lower *tick_time*) requires adjusting the scrape interval of the Prometheus instance. Additionally, *pollect* needs the specification of *scrape_targets* to identify the sources that will be queried. These targets are characterized by their type (in our case, Vsphere and Dell Open Manage), the address where they can be accessed, and the credentials required for authentication. In our case, we monitor one vSphere and one Open Manage instance each for data centers in Erlangen and Karlsruhe.

Key	Value	Description
tick_time	$60s$	Intervall to query APIs for data
scrape_target 1	VSphere API Endpoint 1	Data center Erlangen (DE)
scrape_target 2	VSphere API Endpoint 2	Data center Karlsruhe (DE)
scrape_target 3	Open Manage API Endpoint 1	Data center Erlangen (DE)
scrape_target 4	Open Manage API Endpoint 2	Data center Karlsruhe (DE)

Table 6.3: Test environment: Pollect resource collector configuration

Energy calculator configuration

The energy calculator uses the start and end time of jobs and steps and power samples to calculate energy values and does not require additional configuration (see Equation 5.2). To determine the CO2 emissions, the PUE factor and carbon density must be configured (see Equation 5.3). Since the used data centers are located in Germany, we assume a PUE value of 1.56 carbon intensity of $349\frac{gCO_2}{KWh}$, corresponding to the German average. Note that we have chosen these values to generate generally valid values. They do not provide exact information about the emission values at Siemens. Siemens data centers are state-of-the-art and therefore have a significantly lower PUE and, due to the use of renewable energies, also a significantly lower carbon intensity. The service interval determines how often the database is checked for new runs. After the calculation, the energy values for the new runs are automatically written back to the database (see Algorithm 3).

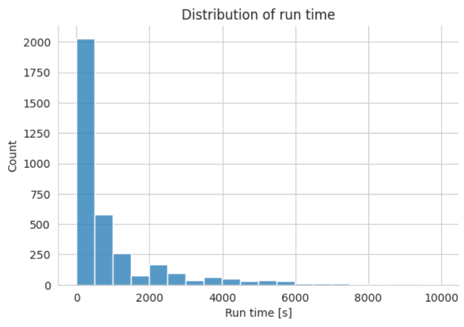
Key	Value	Description
PUE	1.56	Power Usage Efficiency Factor
carbon_density	$349\frac{gCO_2}{KWh}$	CO_2 emissions [g] per kilowatt-hour
service_interval	$3600s$	Write energy values back to the database

Table 6.4: Test environment: Energy calculator configuration

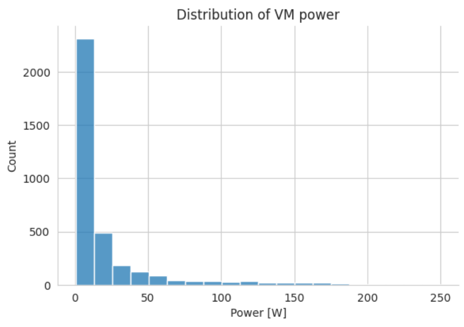
6.3 Results

6.3.1 Overview

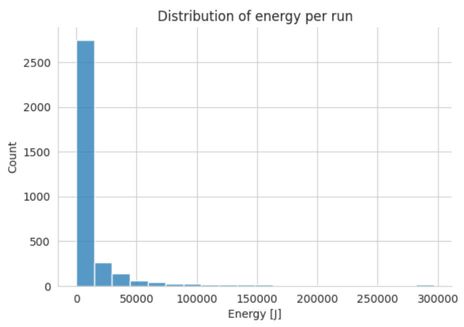
With this described setup, we tracked real-world operations for two weeks. During this period, we logged 3.571 pipeline runs, which comprised 11.791 jobs and 173.250 steps. We monitored 93 servers with a total of 1330 VMs. The distribution plots of the run time, power, energy consumption and the



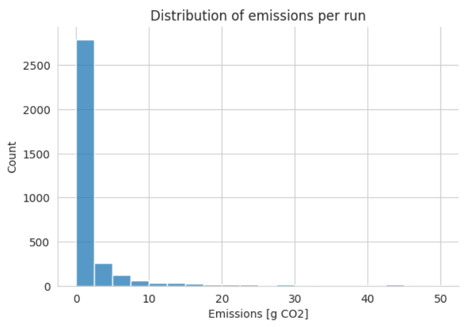
(a) Run time distribution



(b) Power distribution



(c) Energy distribution



(d) Emission distribution

Figure 6.1: Distribution plots of pipelines: Run time, power, energy, emissions

emissions are displayed in Figure 6.1. We recorded the following average values:

- Average pipeline run time: $863s$
- Average power-consumption of the used runners: $30.98W$
- Average energy for a pipeline run: $26736.95J$
- Average emissions for a pipeline run: $4.04gCO_2$

6.3.2 RQ 3.1 Categories

This section answers the research question *What are the general categories into which jobs and steps that are executed in CI/CD pipelines can be grouped?* We first extract all the names assigned to the jobs and steps, calculate their frequency, and then form categories based on the most commonly occurring words. Table 6.5 shows the resulting categories. It is important to note

that some words are inherently interconnected. For example, *container* and *docker* are frequently associated, and we consequently group them into a single category.
Afterward, each job and step is assigned a category based on its name. We utilize the concept of *Levenshtein* distance to calculate the minimal number of edit operations (insertion, deletion, or substitution of a character) required to transform one string into another [55]. We compute the *Levenshtein* distance for each of the predefined categories (Table 6.5). The used implementation* outputs values in the range 0 to 100, where higher values indicate a better matching. Subsequently, we attribute each job or step to the category with the highest matching score, if it surpasses the predetermined threshold of 50. Otherwise, we place the job or step in the *other* category.
We notice that the classification often fails to capture the semantic meaning. To counteract this problem, we use *BLEURT*, a machine-learning-based metric for evaluating text developed by Google [56]. It utilizes a pre-trained transformer model to generate embeddings for the reference and generated text. The embeddings are then used to calculate a score, quantifying the similarity between the two texts, focusing on semantics and context rather than just similarities based on the characters. We use the pre-trained *TinyBERT* model for the similarity matching, as it provides a good tradeoff between accuracy and inference speed [57]. The model does not give a fixed range of output values. We record values between -2 and 1, where higher values represent a better matching. Analogous, if the score does not surpass the predetermined threshold -0.5, we place the job or step into the *other* category.
Table 6.6 shows the number of jobs in each category, and its relative share, based on the *TinyBERT* classification. Figure 6.2 visualizes this data. Most jobs belong to the other category, followed by release, test, setup and clean-up, scan and building. Table 6.7 and Figure 6.3 show the ratio of the different step categories. When assigning steps to the same categories, we observe a different distribution. Here, the setup and clean-up categories are represented most, with test in second place, followed by release, container, and other.

*`https://github.com/ztane/python-Levenshtein`

Table 6.5: Common Categories of DevOps Jobs and Steps

Category	**Description**
Build	Compiling and constructing the application or software from source code.
Container	Working with containerization technologies such as Docker to package and deploy applications.
Linting	Reviewing source code to flag programming errors, bugs, stylistic errors, and suspicious constructs.
Other	Miscellaneous tasks or responsibilities not falling into other specific categories.
Release	Managing and deploying software releases to various environments or production systems.
Scan	Conducting security scans, vulnerability assessments, or license violations on software or infrastructure to identify potential risks.
Setup and clean-up	Provide a clean, consistent environment and ensure resources are released after a job completes.
Test	Designing, implementing, and executing tests to ensure the quality and functionality of the software.

Category	**Count**	**Relative (%)**
Build	608	5.19 %
Container	450	3.84 %
Linting	104	0.89 %
Other	4326	36.91 %
Release	2847	24.29 %
Scan	944	8.05 %
Setup and Clean-up	1096	9.35 %
Test	1346	11.48 %

Table 6.6: Count and relative ratio of job categories based on *Bleurt*

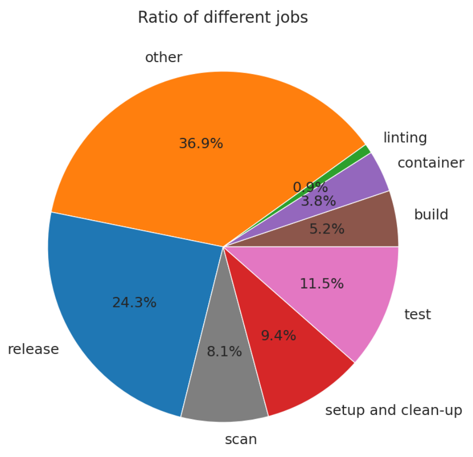


Figure 6.2: Ratio of job categories based on *Bleurt*

Category	**Count**	**Relative (%)**
Build	12716	7.45 %
Container	15670	9.18 %
Linting	2302	1.35 %
Other	15450	9.05 %
Release	26331	15.43 %
Scan	6773	3.97 %
Setup and Clean-up	59472	34.84 %
Test	31984	18.74 %

Table 6.7: Count and relative ratio of step categories based on *Bleurt*

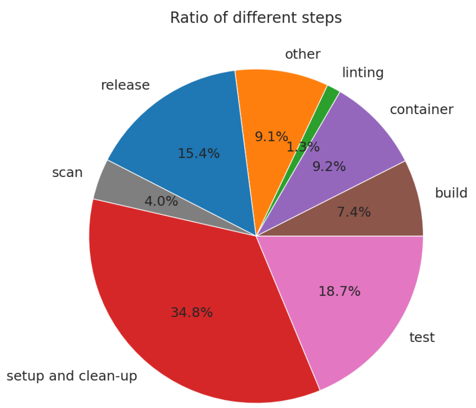


Figure 6.3: Ratio of step categories based on *Bleurt*

6.3.3 RQ 3.2 Energy

This section answers the research question 3.2 *What is the energy consumed by jobs and steps in these different categories?* Table 6.8 shows the mean energy consumption in Joules for different jobs and steps within the CI/CD pipelines. The column *Job Energy* represents the total energy in Joules required to complete a particular job. This includes all steps within that job. The *Step Energy* column refers to the energy consumption of individual steps within a job.

The average energy consumptions for the individual categories are shown in Figure 6.4 for jobs and Figure 6.5 for steps. *Release* jobs, consuming $45582.03J$, are the most energy-intensive jobs. They typically involve packaging the software, managing configurations, and deploying the software to the servers. The corresponding *release* steps consume significantly less energy at $170.32J$. The *build* job consumes the second most energy, with a total of $37184.21J$. Build processes involve compiling code and dependencies, which are typically resource-intensive. The *build* step consumes far more energy than all other steps, with a total of $3903.80J$. *Test* jobs, used to validate the software's functionality, consume $6064.07J$. The corresponding *test* steps consume $657.43J$. The *container* jobs consume, on average $1585.45J$, and the corresponding steps $107.01J$. The *setup and clean-up* jobs used to prepare the pipeline environment consume $1314.15J$, and the corresponding steps consume $423.89J$. The *scan* job consumes the least energy overall, at $274.67J$, and the *scan* steps consume $178.29J$. Finally, the *other* category

comprises various other tasks and steps in the DevOps pipeline which could not be assigned confidently to one of the categories. This category consumed $10258.90J$ for jobs and $175.25J$ for steps.
We compute the proportional contribution of each category to the total energy consumption. This is done by multiplying the count for each category by its corresponding median energy. The outcome of this computation for jobs is outlined in Table 6.9 and depicted in Figure 6.6. The data shows that the *release* jobs account for over 60% of the total energy expenditure for the observed pipeline runs. Conversely, the findings for steps are presented in Table 6.10 and Figure 6.7. We observe that the *build* steps make up the largest share, accounting for over 45% of the energy consumption of all the analyzed steps.
Finally, we explored the connection between runtime and energy consumption, discovering a correlation of 0.661 for jobs and a correlation of 0.689 for steps. Hence, while runtime has a substantial impact, there are also notable variations in the average power consumption, explaining the different energy values.

Table 6.8: Energy Requirements for Jobs and Steps

Category	**Job Energy (J)**	**Step Energy (J)**
Build	37184.21	3903.80
Container	1585.45	107.01
Linting	6028.30	161.95
Other	10258.90	175.25
Release	45582.03	170.32
Scan	274.67	178.29
Setup and clean-up	1314.15	423.89
Test	6064.07	657.43

6.4 Summary

We quantify the average energy consumption of pipeline runs, jobs, and steps. To be able to make more precise statements, we classify jobs and steps into distinct categories and analyze their energy consumption. Our experiments found that jobs categorized as *release* have the highest energy consumption, while the *build* category has the highest energy consumption among steps.

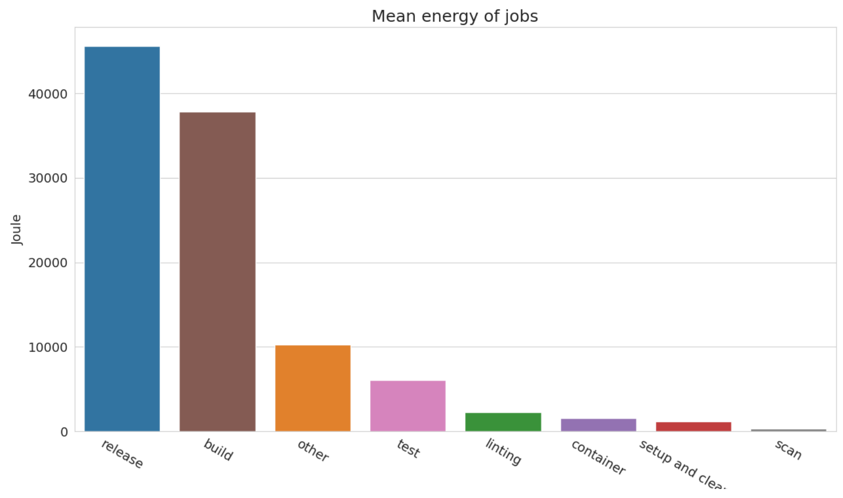


Figure 6.4: Mean energy for jobs for different categories

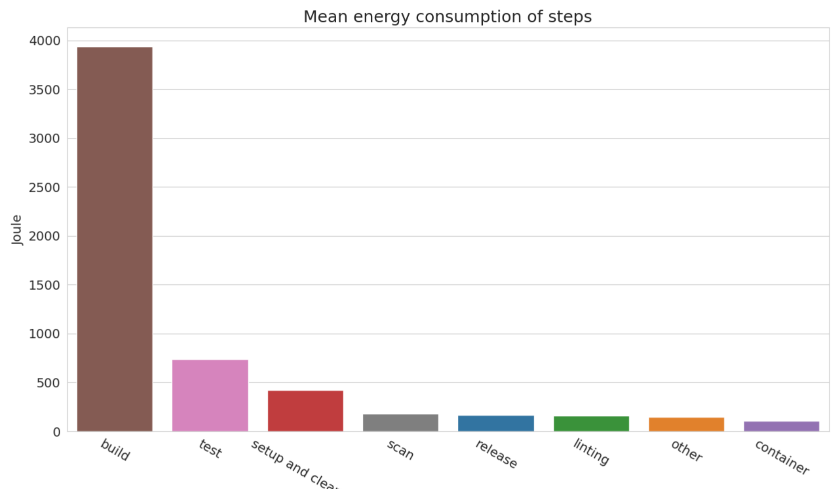


Figure 6.5: Mean energy for steps for different categories

Category	Total Energy (J)	Relative (%)
Release	1.30×10^8	62.52
Other	4.42×10^7	21.29
Build	2.30×10^7	11.08
Test	8.11×10^6	3.91
Setup and Clean-up	1.28×10^6	0.62
Container	7.17×10^5	0.35
Scan	2.60×10^5	0.13
Linting	2.34×10^5	0.11

Table 6.9: Total and relative energy consumption of job categories

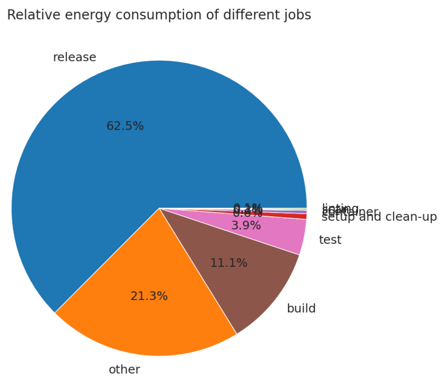


Figure 6.6: Relative energy consumption of the job categories in relation to the total energy consumption of all jobs

Category	Total Energy (J)	Relative (%)
Build	5.00×10^7	45.94
Setup and Clean-up	2.53×10^7	23.21
Test	2.37×10^7	21.73
Release	4.40×10^6	4.04
Other	2.26×10^6	2.07
Container	1.68×10^6	1.55
Scan	1.22×10^6	1.12
Linting	3.73×10^5	0.34

Table 6.10: Total and relative energy consumption of step categories

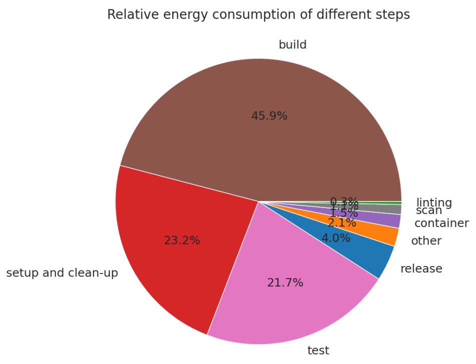


Figure 6.7: Relative energy consumption of the step categories in relation to the total energy consumption of all steps

Chapter 7

Discussion

7.1 Energy measurements of pipeline runners

While examining existing methods for energy measurements in the context of CI/CD pipelines, we faced various challenges. There is no one-fits-all solution since power measurements depend significantly on the underlying technology. A higher level of accuracy, reached by hardware-based measurements, typically leads to more coarse-grained granularity and limited applicability for CI/CD runners. Additionally, a high measurement frequency can often only be achieved by agent-based solutions, requiring instrumentation of the used runner and creating overhead. Based on these findings, a layered approach was the best solution for the on-premise runners in our test environment. We collect high-accuracy data from the server's power supply sensors. The runners in our test environment are VMs. Therefore, the power values of the individual runners on this device are queried from the hypervisor to reach more fine-grained results. This way, we validate that the sum of all runners on a device plus the IDLE power equals the total device power. This approach might not be applicable for cloud runners since they usually do not provide access to the underlying hardware and are limited to exposed measurements or agent-based solutions.

7.2 Planetary

The proposed framework leverages existing pipeline logs, available on most modern DevOps platforms, and combines them with resource data without making any assumptions. This makes it flexible and applicable to different CI/CD setups. At the same time, it is the responsibility of the individual components to ensure that the values generated are accurate.

Log analyzer

Our log analyzer implementation examines existing pipeline run logs and stores the data in a PostgreSQL database. We opted for an interval-based implementation that regularly retrieves and processes all data from the Azure DevOps API. Although this method introduces additional computational and storage requirements, a small Kubernetes pod (0.5 CPU and $512MB$ memory) was sufficient to process the new runs of the 2507 repositories within an average of 2 minutes when executed once an hour. The total storage was less than $400MB$ for over 1 million steps and 100000 jobs and 25000 runs. Storing CI/CD pipeline data in a relational database has proven valuable. Additional to energy values, this data also allows to create pipeline statistics, for example, for failure rates.

Resource collectors

To collect power and resource data, we intensively studied the technologies used in our test environment. Our experiences with vSphere found its powerful API a valuable resource, providing comprehensive data, including power samples for VMs. Overall, it has proven to be reliable. Occasionally, it could not provide values for servers or VMs indicated by a -1. We discovered that the Dell Open Manage API is notably slow, probably due to its internal rate-limiting. However, it was an essential validation tool for the power measurements acquired from vSphere. Figure C.3 (Appendix) compares vSphere and Dell Open manage power samples and shows their high correlation. Incorporating both technologies into *pollect* as a data source was straightforward, facilitating regular data collection. This setup demonstrated great dependability, permitting uninterrupted data recording over weeks without encountering any issues.

Energy calculator

The energy calculator component combines log data, providing information pipelines run duration, and the used runner, with time-series data, providing the resource consumptions. If power samples are already available, as in our test environment, calculating the energy consumption of a pipeline is straightforward. Transforming these energy values into emissions requires the *Power Usage Effectiveness* factor and the *carbon intensity*. Given that deriving these values can be complicated and is not a core part of this work, we resorted to the German average values. Finally, discussions with stakeholders within Siemens revealed that it is essential to make the calculated energy consumption values as easily accessible as possible to those responsible. Creating Grafana dashboards to enable live monitoring and an easy-to-use API for querying more fine-grained energy values proved a good solution.

7.3 Experiments

Our study revealed that a typical pipeline execution duration is approximately 14 minutes, consuming a mean power of about 30 watts, resulting in an estimated carbon dioxide emission of $4g$. Though these emissions seem small, we recorded over 250 pipeline executions per day, contributing to $1kg$ of emissions per day.

Job and step categories

Since absolute values may not be relevant to external entities or individuals outside our organizational context, we conducted a comparative analysis of different categories of CI/CD steps and jobs. An initial categorization approach was implemented utilizing the *Levenshtein* distance metric. This metric did not yield satisfactory results. For example, the words *Linux* and *Linter* have no semantic commonality but a low Levensthein distance as they share the first three letters and have the same number of characters. We transitioned to *BLEURT*, which calculates the similarity based on the semantic meaning. In our organizational context, the naming of steps and jobs generally adheres to a specific naming convention, yielding good results. For jobs, the *release* category was the most represented with 24.3%, while for steps the *setup and clean-up* category had the highest share with 34.8%. This discrepancy observed between the distribution of jobs and steps to categories can be explained by the compositional multi-step structure of the jobs. For instance, a release job usually encompasses several steps, including a build phase and a release phase responsible for either publishing the resultant artifact or deploying the built container. Additionally, most jobs include setup and clean-up steps.

Job and step energy consumption

The high energy consumption of build jobs and steps was expected. The high values for release jobs can be primarily attributed to the inclusion of build steps within these jobs. In most instances, release jobs comprise one or even multiple build steps, supplemented by an additional step responsible for releasing the build binaries or artifacts. Test steps and jobs in software development can be energy-intensive due to the numerous test cases required to cover the functionality of large software projects. Additionally, they often require resource intensive simulations and emulations. We recorded low energy values for scanning jobs, which typically involve computationally intensive tasks like checking the code for errors or vulnerabilities. However, in our test environment, these tasks are outsourced to external services, explaining the low energy consumption on the monitored pipeline runner, which is only responsible for triggering this service.

7.4 Pipeline energy efficiency improvements

Based on the experiments and experience we have gained during the analysis, we derive some potential measures to optimize the energy consumption of CI/CD pipelines. Energy optimization techniques can be classified into three broad categories: focusing on power reduction, time reduction, or balance between power and time trade-off [58]. Since power reduction is mainly achieved by hardware optimization, we focus on reducing the run time of the CI/CD pipelines.

7.4.1 Category based improvements

For the categories identified as the most energy-intensive in Chapter 6, we utilize existing research to suggest and discuss energy efficiency enhancements for jobs and steps in the respective category.

Build optimization

In the analyzed pipelines, we identify build steps as the most energy-consuming (Section 6.3.3). One strategy to optimize builds is to use caching. Caching in CI/CD pipelines is a strategy that stores results from runs, allowing for their reuse in subsequent runs to reduce build time and resource usage. Dependency caching can reduce download times by reusing cached libraries, SDKs, and plugins in subsequent builds [59]. Build Artifacts Caching stores artifacts that allow skipping steps in subsequent builds if the source code has not changed. Build artifact caching is enabled by incremental compilation, allowing build systems to update prior build artifacts efficiently. This can be achieved by leveraging dependency information between build steps, identifying those impacted by source updates, and running only those necessary for correct artifact generation. Implementing incremental builds faces various challenges. These include compromised correctness in existing build systems and high environmental dependency of build systems impacting product consistency. The issue of linearity in continuous integration often necessitates complete rebuilds due to worker dependencies on previous build products. Maudoux et al. experimented on a large open-source browser, revealing that approximately 90% of CPU time on CI workers is wasted, highlighting that incremental builds could offer substantial resource savings[60].
Galaba et al. [61] propose a framework to interfere build accelerations decisions automatically. The two main strategies are to use caching of build environment and skipping unaffected build steps, In their evaluations, over 87% of the builds could be accelerated, and 74% percent of accelerated builds achieve a speed-up of two-fold with respect to their non-accelerated counterparts.
Dependency caching is easy to implement. However, invalidating and updating the cache takes time and requires additional storage, and therefore is not beneficial in every case. While incremental builds offer much potential to save runtime and energy, they can add complexity and the risk of incorrect builds.

Setup and Clean-up optimization

Chapter 6 shows that 34.84% of the steps can be assigned to the *setup and cleanup* category. They are responsible for getting the environment ready to run the actual steps and jobs and cleaning up after running. They can be accelerated by using a container solution such as Docker, and in some cases may even be unnecessary. The use of Docker enables the rapid creation of isolated environments. While containers can be created and destroyed much faster than virtual machines, they start in a known state, reducing the required setup. By using the correct container, with an optimized base image and all the tools already installed, can significantly reduce pipelines execution duration [62].

Test optimization

Test prioritization can optimize tests in CI pipelines by sequencing tests based on their importance and potential impact on the overall application. By running high-priority tests first, the application's critical functionality and key components are tested early in the process. This approach allows for faster feedback and can reduce the total energy consumed if the pipeline can be terminated early in case of failures. Marijan et al. present a systematic approach to optimize the run-time of CI testing cycles [63]. The goal is to minimize the run time while maintaining good fault detection and coverage. To solve this multiobjective optimization problem, they propose history-based and risk-based test optimization to compute an optimized set of tests. They found that if tests are selected to test only the most relevant and risky functionality related to the changes, substantial savings can be made in the duration of testing. The same authors [64] developed an algorithm for identifying and eliminating test redundancy in regressions tests. Regression tests ensure that previously developed and tested software still performs correctly after changes or modifications have been made to it. The main issue addressed is that software functionality is being tested multiple times across different test cases. The proposed algorithm employs coverage metrics on historical data to categorize tests into three groups: unique, completely redundant, or partially redundant. When evaluated, this approach demonstrated the potential to reduce the execution time of regression tests by 31% when skipping redundant tests.

7.4.2 General best practices

This section discusses general recommendations and best practices for augmenting pipeline energy efficiency. Besides strictly technical optimizations, the development and organization of CI/CD pipelines offer opportunities for optimization.

Programming language

The choice of programming language of a project and the associated toolchain greatly influence the energy consumption of the respective CI/CD pipeline. We generally differentiate programming languages by the execution type between compiled, virtual machine and interpreted. While compiled and virtual machine languages have a compiling step, interpreted languages do not. Modern interpreted languages, like Javascript, use Just-In-Time (JIT) compilation to compile JavaScript to bytecode or machine code just before or during its execution. As a result, the build step in an interpreted language is often much more energy efficient or even completely unnecessary, significantly reducing the entire pipeline's energy consumption. At the same, interpreted languages often consume many times more energy during the execution [65] [66].

Runner efficiency

Choosing the right type and size of agents or runners for a pipeline can lead to power reduction. The configuration of a VM can impact energy consumption, even if two VMs are executing the same job. The primary factors influencing this are the VM's resource allocation (CPU, memory, and storage) and its utilization efficiency. Suppose a job is placed on a larger VM and can leverage additional resources to complete the task more quickly. In that case, it may result in higher power usage but lead to the same overall energy consumption. If the job can not make use of these extra resources, a larger VM might lead to higher energy consumption due to inefficiencies and overhead. Idle or underutilized resources still consume power, leading to unnecessary energy use [67].

Trigger and chain optimization

Trigger and chain optimization can effectively enhance energy efficiency in CI/CD pipelines when specific expensive steps can be skipped under known conditions. The authors of [68] analyze commits where CI runs are triggered unnecessarily. These include commits where the changes only affect non-source code files or modifications that involve formatting or commenting. They propose an automated process based on a set of rules to determine which commits can be skipped in the CI process. The method found that about 18% of the build commits can be CI skipped in their test environment.
Path filters allow for the configuration of pipelines to trigger only when

changes occur in specific file paths. This is particularly beneficial when dealing with monorepos or repositories that house multiple projects. By skipping pipelines without significant changes, unnecessary executions can be avoided, resulting in significant energy savings. For instance, GitHub Actions provides functionality allowing to use *paths* and *paths-ignore* keywords to filter pipeline triggers based on the altered file paths. Consequently, if the changes do not affect specific components, pipelines associated with these components will not be initiated [69]. Conditional statements are another powerful method for avoiding needless pipeline runs, thus conserving energy. Many CI/CD systems allow conditional statements to control whether specific steps or jobs in the pipeline should be executed based on the outcome of previous steps or the nature of the triggering event [70]. This kind of selective execution is applicable to most pipelines and can significantly reduce the number of unnecessary jobs, resulting in faster pipeline completion times and reduced energy consumption.

Pipeline development and debugging

One common strategy developers adopt when configuring CI/CD pipelines is the trial and error method, which often entails pushing changes to the remote repository multiple times and initiating pipeline runs until they function as expected. While it may eventually yield functional pipelines, this approach results in unnecessary pipeline executions and, thus, increased energy consumption. The development and deployment of enhanced linting tools can significantly mitigate this issue. Linting tools are instrumental in catching logical and syntactic errors at the preliminary stage of writing and editing configurations in the online editor. By incorporating these tools within popular development environments, developers can identify and rectify simple errors before committing changes, thereby circumventing needless pipeline runs.

Vassallo et al. [71] introduced a semantic linter capable of automatically identifying four distinct types of *smells* in pipeline configuration files. These smells, or sub-optimal characteristics, include *fake success*, wherein every executed job should possess the capability to fail the build, and *Retry Failure*, which signifies non-deterministic behavior prompting developers to address failed builds by rerunning the pipeline.

As previously discussed in Chapter 2, specific platforms like GitHub and GitLab enable local pipeline execution. This feature prevents unnecessarily pushing code to remote repositories. This consumes energy locally but can often prevent a cascade of pipelines or jobs. However, this method may not be suitable for larger jobs, which may not be executable locally. In summary, improving the tools in configuring and debugging CI/CD pipelines can streamline establishing error-free pipelines. This saves developers' time and can lead to significant energy savings by reducing the need for unnecessary pipeline runs.

7.4.3 Applicability of improvements

While these strategies provide general ideas for improving energy efficiency in CI/CD pipelines, their implementation is contingent upon the unique circumstances of each pipeline. Some energy improvements mentioned are already being applied in some of the pipelines analyzed. For example, Docker is used, but the base images could be further improved. Energy optimization may not always align with other performance goals. Build caching can improve run time, but it might also introduce latency and additional storage costs. Similarly, using interpreted languages to reduce build time may have downstream impacts on execution time and overall performance. We note that the practicality of the strategies, such as optimizing triggers and chains, relies heavily on the existing architecture and complexity of the CI/CD pipelines and the used platform. For example, path filters and conditional statements, to only trigger pipelines if required, are available on both Azure DevOps and Github and are already used in many of the pipelines analyzed. This thesis's main goal is to measure the energy consumption of CI/CD pipelines. The implementation and verification of the suggested improvements exceed the scope of this work and are left for future work. Lastly, the potential for direct power reduction, for instance, through hardware optimization, remains unexplored.

7.5 Threats to validity

7.5.1 Internal validity

In order to evaluate existing approaches to measure energy consumption in CI/CD pipelines, we have conducted rigorous investigations through academic publications and practical tools. Despite our best efforts, there may still be some methods that have not been adequately analyzed or have been overlooked. We believe that the information in Table 4.1 offers a

valuable guideline for selecting an appropriate energy measurement approach. However, we note that its basis largely stems from our experiences and findings within a specific testing setting.
To determine the energy consumption of CI/CD pipelines, it is crucial to have access to the resource data of the pipeline runner. Suppose this data is unavailable because the runner was not monitored, the runner's name can not be unambiguously mapped to a machine, or the server or monitoring service has restarted. In this case, the computation of energy consumption for these pipelines is impossible. The pipeline runs missing runner data were filtered out in the presented data. However, this situation is problematic if the objective is to observe energy consumption not from isolated pipeline runs but on a larger scale, such as all pipeline executions within the last 24 hours. This could result in a substantial underestimation of energy usage.
The sampling rate of resource data can significantly influence the accuracy of results. If a pipeline experiences considerable power consumption variations and a low sampling rate, estimated power consumption may deviate from the actual consumption, particularly in short-duration pipelines and jobs. Therefore, maximizing the sampling frequency within the limits of the technology used is advisable but is not enforced by the framework.
It is important to recognize that the calculated energy values do not include the energy consumed by the external services. Typical examples of external services used in CI/CD pipelines include platforms for managing code quality, security services, and hosting and managing artifacts, binaries, and containers. The CI/CD pipelines only trigger the respective service by API calls or predefined actions. The energy consumption occurs at the respective service providers, which can only be measured if the service reports the energy value. As a result, the energy values we measure are lower than the total energy consumption.
Moreover, the classification of different categories of jobs and steps is based on their naming. Within Siemens, certain naming conventions exist, which is why this classification works well, but this approach might not be transferable to other environments. Lastly, we acknowledge the potential for errors in our implementation of the services. We have conducted extensive testing of our code to minimize these risks.

7.5.2 External validity

Planetary is designed as a universal framework suitable for most contemporary IT infrastructures. However, its experimental system is devised and assessed within the unique software/hardware environment of Siemens IT, limiting the external validity of the experiments to these specific environments. Firstly, the experiments are premised on specific pipelines used mainly for constructing large-scale software applications. While we have endeavored to produce results with broad validity by incorporating projects of diverse responsibilities, the generalizability of these pipelines remains questionable. The industrial context is a commendable example of industry practice, yet it could vary significantly across different companies. Secondly, our implementation focuses on hosted runners, even though the framework is designed to incorporate cloud runners easily. Using hosted runners has the advantage of determining the server on which a pipeline runner operates and accessing the server's power sensors, which is often not cumbersome or not possible at all in a cloud environment. Using different hardware and software to run CI/CD pipelines may result in different energy values.

7.6 Sustainability and Ethics

Sustainability is the primary motivation behind this project. Siemens as a large and international company, is committed to sustainability by providing technologies for sustainable infrastructure, developing digital solutions to measure consumption, and ensuring data transparency [72]. This project seeks to complement these efforts by working towards greening IT processes. In the future, computing will become even more ingrained in our lives, and it is a fundamental duty of engineers and developers to mitigate the environmental impact of these technologies. Optimizing the energy efficiency of CI/CD pipelines is one of many possible steps. To further this cause, the code developed through this project is open-source, hoping to inspire and facilitate the adoption of energy measurement of CI/CD pipelines in others.

Chapter 8

Conclusion

In this work, we successfully combined the domain of energy measurements with CI/CD pipelines. We analyzed different energy measurement strategies and their applicability in CI/CD pipelines. Hardware-based methods provide accurate results and have low overhead on pipeline runners. Software-based solutions allow higher granularity and are easier to integrate. These insights served as a foundation for developing the generic and adaptable *Planetary* framework. An implementation of this framework was integrated into the modern IT infrastructure at Siemens. We monitored CI/CD pipelines for two weeks in a real-world production environment. In addition to the calculated energy values, we have identified and categorized different types of jobs and steps within these pipelines. This subdivision allowed us to the most energy-intensive categories. For the categories *build*, *test*, and *setup*, we have proposed generic improvement strategies and general best practices to improve the energy efficiency of CI/CD pipelines.
While our work has yielded promising results, it has also highlighted areas for future research. One such area is the analysis of CI/CD pipelines executed on cloud runners. Our framework could be extended by adding a new data source to the resource collector and a suitable energy model. Another aspect involves integrating energy measurements of external services, providing a more holistic view of the total energy use in CI/CD pipelines. We hope that more API providers will provide transparency and clarity on how much energy is required for specific calls. Finally, it would be interesting to implement the suggested improvements and measure their effects on their energy consumption.

References

[1] C. Freitag, M. Berners-Lee, K. Widdicks, B. Knowles, G. S. Blair, and A. Friday, “The real climate and transformative impact of ICT: A critique of estimates, trends, and regulations,” vol. 2, no. 9, p. 100340. doi: 10.1016/j.patter.2021.100340. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S2666389921001884 [Page 1.]

[2] Statista. Energy demand in data centers worldwide. https://www.statista.com/statistics/186992/global-derived-electricity-consumption-in-data-centers-and-telecoms/. Accessed: 2023-06-15. [Page 1.]

[3] L. Ardito, R. Coppola, M. Morisio, and M. Torchiano, “Methodological guidelines for measuring energy consumption of software applications,” vol. 2019, pp. 1–16. doi: 10.1155/2019/5284645. [Online]. Available: https://www.hindawi.com/journals/sp/2019/5284645/ [Pages 3 and 14.]

[4] M. A. Lopez-Pena, J. Diaz, J. E. Perez, and H. Humanes, “DevOps for IoT systems: Fast and continuous monitoring feedback of system availability,” vol. 7, no. 10, pp. 10 695–10 707. doi: 10.1109/JIOT.2020.3012763. [Online]. Available: https://ieeexplore.ieee.org/document/9151987/ [Page 3.]

[5] R. Merritt. What is green computing? Accessed: 2023-06-15. [Online]. Available: https://blogs.nvidia.com/blog/2022/10/12/what-is-green-computing/ [Page 5.]

[6] S. Asadi, A. R. C. Hussin, and H. M. Dahlan, “Organizational research in the field of green IT: A systematic literature review from 2007 to 2016,” vol. 34, no. 7, pp. 1191–1249. doi: 10.1016/j.tele.2017.05.009. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0736585316305408 [Page 5.]

[7] O. Zakaria, "Software eco-design: Investigating and reducing the energy consumption of software." [Online]. Available: https://theses.hal.science/tel-03429300/document [Page 6.]

[8] J. M. Hirst, J. R. Miller, B. A. Kaplan, and D. D. Reed, "Watts up? pro AC power meter for automated energy recording: A product review," vol. 6, no. 1, pp. 82–95. doi: 10.1007/BF03391795. [Online]. Available: http://link.springer.com/10.1007/BF03391795 [Pages 6, 21, and 22.]

[9] K. N. Khan, M. Hirki, T. Niemi, J. K. Nurminen, and Z. Ou, "RAPL in action: Experiences in using RAPL for power measurements," vol. 3, no. 2, pp. 1–26. doi: 10.1145/3177754. [Online]. Available: https://dl.acm.org/doi/10.1145/3177754 [Pages 7, 15, 18, and 23.]

[10] S. Desrochers, C. Paradis, and V. M. Weaver, "A validation of DRAM RAPL power measurements," in *Proceedings of the Second International Symposium on Memory Systems*. ACM. doi: 10.1145/2989081.2989088. ISBN 9781450343053 pp. 455–470. [Online]. Available: https://dl.acm.org/doi/10.1145/2989081.2989088 [Pages 7, 15, and 23.]

[11] T. Li, Institute of Electrical and Electronics Engineers, and IEEE Computer Society, "BFEPM: Best fit energy prediction modeling based on CPU utilization." [Page 7.]

[12] P. Rattanatamrong, Y. Boonpalit, S. Suwanjinda, A. Mangmeesap, K. Subraties, V. Daneshmand, S. Smallen, and J. Haga, "Overhead study of telegraf as a real-time monitoring agent," in *2020 17th International Joint Conference on Computer Science and Software Engineering (JCSSE)*. IEEE. doi: 10.1109/JCSSE49651.2020.9268333. ISBN 9781728161679 pp. 42–46. [Online]. Available: https://ieeexplore.ieee.org/document/9268333/ [Page 7.]

[13] P. Heller, *Automating Workflows with GitHub Actions*, 1st ed. Packt Publishing. ISBN 9781800560406 OCLC: 1285525493. [Page 8.]

[14] Github. Migrating from GitLab CI/CD to GitHub actions. Accessed: 2023-06-15. [Online]. Available: https://ghdocs-prod.azurewebsites.net/en/actions/migrating-to-github-actions/manual-migrations/migrating-from-gitlab-cicd-to-github-actions [Page 8.]

[15] Gitlab. Runner documentation. Accessed: 2023-05-03. [Online]. Available: https://docs.gitlab.com/runner/ [Page 9.]

[16] R. D. Caballar. These are the world's 12 largest hyperscalers. Accessed: 2023-05-03. [Online]. Available: https://www.datacenterknowledge.com/manage/2023-these-are-world-s-12-largest-hyperscalers [Page 11.]

[17] C. Anderson, "Docker [software engineering]," vol. 32, no. 3, pp. 102–c3. doi: 10.1109/MS.2015.62. [Online]. Available: http://ieeexplore.ieee.org/document/7093032/ [Page 12.]

[18] Kubernetes documentation. Accessed: 2023-05-03. [Online]. Available: https://kubernetes.io/docs/home/ [Page 13.]

[19] B. Burns, J. Beda, and K. Hightower, *Kubernetes up & running: dive into the future of infrastructure*, second edition ed. O'Reilly Media. ISBN 9781492046530 [Page 13.]

[20] L. Cruz. Green software engineering done right: a scientific guide to set up energy efficiency experiments. Accessed: 2023-06-15. [Online]. Available: https://luiscruz.github.io/2021/10/10/scientific-guide.html [Pages 14 and 15.]

[21] M. Martinez, S. Martínez-Fernández, and X. Franch, "Energy consumption of automated program repair." doi: 10.48550/ARXIV.2211.12104. [Online]. Available: https://arxiv.org/abs/2211.12104 [Page 14.]

[22] S. Georgiou, S. Rizou, and D. Spinellis, "Software development lifecycle for energy efficiency: Techniques and tools," vol. 52, no. 4, pp. 1–33. doi: 10.1145/3337773. [Online]. Available: https://dl.acm.org/doi/10.1145/3337773 [Page 15.]

[23] L. Gerhorst, S. Reif, B. Herzog, and T. Honig, "EnergyBudgets: Integrating physical energy measurement devices into systems software," in *2020 X Brazilian Symposium on Computing Systems Engineering (SBESC)*. IEEE. doi: 10.1109/SBESC51047.2020.9277849. ISBN 9781728182865 pp. 1–8. [Online]. Available: https://ieeexplore.ieee.org/document/9277849/ [Pages 15 and 23.]

[24] R. Schone, T. Ilsche, M. Bielert, M. Velten, M. Schmidl, and D. Hackenberg, "Energy efficiency aspects of the AMD zen 2 architecture," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE. doi: 10.1109/Cluster48925.2021.00087. ISBN 9781728196664 pp. 562–571.

[Online]. Available: https://ieeexplore.ieee.org/document/9556102/ [Page 16.]

[25] A. Kansal, F. Zhao, J. Liu, N. Kothari, and A. Bhattacharya, "Joulemeter: Virtual machine power measurement and management," Tech. Rep. MSR-TR-2009-103, August 2009, mSR Tech Report. [Online]. Available: https://www.microsoft.com/en-us/research/publication/joulemeter-virtual-machine-power-measurement-and-management/ [Pages 16 and 24.]

[26] M. Colmant, M. Kurpicz, P. Felber, L. Huertas, R. Rouvoy, and A. Sobe, "Process-level power estimation in VM-based systems," in *Proceedings of the Tenth European Conference on Computer Systems*. ACM. doi: 10.1145/2741948.2741971. ISBN 9781450332385 pp. 1–14. [Online]. Available: https://dl.acm.org/doi/10.1145/2741948.2741971 [Page 16.]

[27] W. Wu, W. Lin, and Z. Peng, "An intelligent power consumption model for virtual machines under CPU-intensive workload in cloud environment," vol. 21, no. 19, pp. 5755–5764. doi: 10.1007/s00500-016-2154-6. [Online]. Available: http://link.springer.com/10.1007/s00500-016-2154-6 [Pages 16, 24, and 34.]

[28] X. Yu, G. Zhang, Z. Li, W. Liangs, and G. Xie, "Toward generalized neural model for VMs power consumption estimation in data centers," in *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*. IEEE. doi: 10.1109/ICC.2019.8762017. ISBN 9781538680889 pp. 1–7. [Online]. Available: https://ieeexplore.ieee.org/document/8762017/ [Pages 16, 24, and 34.]

[29] E. A. Santos, C. McLean, C. Solinas, and A. Hindle, "How does docker affect energy consumption? evaluating workloads in and out of docker containers." doi: 10.48550/ARXIV.1705.01176. [Online]. Available: https://arxiv.org/abs/1705.01176 [Page 17.]

[30] G. Fieni, R. Rouvoy, and L. Seinturier, "SmartWatts: Self-calibrating software-defined power meter for containers." doi: 10.48550/ARXIV.2001.02505. [Online]. Available: https://arxiv.org/abs/2001.02505 [Pages 17 and 34.]

[31] W. Silva-de Souza, A. Iranfar, A. Bráulio, M. Zapater, S. Xavier-de Souza, K. Olcoz, and D. Atienza, "Containergy—a container-based energy and performance profiling tool for next generation workloads,"

vol. 13, no. 9, p. 2162. doi: 10.3390/en13092162. [Online]. Available: https://www.mdpi.com/1996-1073/13/9/2162 [Page 17.]

[32] X. Zhang, Z. Shen, B. Xia, Z. Liu, and Y. Li, "Estimating power consumption of containers and virtual machines in data centers," in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE. doi: 10.1109/CLUSTER49012.2020.00039. ISBN 9781728166773 pp. 288–293. [Online]. Available: https://ieeexplore.ieee.org/document/9229581/ [Page 17.]

[33] T. Veni and S. M. S. Bhanu, "Prediction model for virtual machine power consumption in cloud environments," vol. 87, pp. 122–127. doi: 10.1016/j.procs.2016.05.137. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S1877050916304768 [Page 18.]

[34] W. Lin, H. Wang, Y. Zhang, D. Qi, J. Z. Wang, and V. Chang, "A cloud server energy consumption measurement system for heterogeneous cloud environments," vol. 468, pp. 47–62. doi: 10.1016/j.ins.2018.08.032. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0020025518306364 [Pages 18 and 34.]

[35] B. DAVY. Estimating AWS EC2 instances power consumption. Accessed: 2023-06-15. [Online]. Available: https://medium.com/teads-engineering/estimating-aws-ec2-instances-power-consumption-c9745e347959 [Pages 18 and 34.]

[36] S. P. E. Corporation. SPECpower results. Accessed: 2023-04-24. [Online]. Available: https://www.spec.org/power_ssj2008/results/ [Page 18.]

[37] L. Ismail and H. Materwala, "Computing server power modeling in a data center: Survey, taxonomy, and performance evaluation," vol. 53, no. 3, pp. 1–34. doi: 10.1145/3390605. [Online]. Available: https://dl.acm.org/doi/10.1145/3390605 [Page 18.]

[38] A. Kruglov, G. Succi, and X. Vasuez, "Incorporating energy efficiency measurement into CI\CD pipeline," in *2021 2nd European Symposium on Software Engineering*. ACM. doi: 10.1145/3501774.3501777. ISBN 9781450385060 pp. 14–20. [Online]. Available: https://dl.acm.org/doi/10.1145/3501774.3501777 [Page 19.]

[39] G. C. Berlin. Eco CI. Accessed: 2023-06-15. [Online]. Available: https://www.green-coding.berlin/projects/eco-ci/ [Page 19.]

[40] A. Ahmad. Measuring power using AC wattmeters. Accessed: 2023-04-25. [Online]. Available: https://eepower.com/technical-articles/measuring-power-using-ac-wattmeters/# [Page 21.]

[41] G. Da Costa, J.-M. Pierson, and L. Fontoura-Cupertino, "Mastering system and power measures for servers in datacenter," vol. 15, pp. 28–38. doi: 10.1016/j.suscom.2017.05.003. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S2210537917300318 [Page 21.]

[42] Dell, "Dell VxRail v570f technical documentation." [Page 22.]

[43] E. Rotem, A. Naveh, A. Ananthakrishnan, E. Weissmann, and D. Rajwan, "Power-management architecture of the intel microarchitecture code-named sandy bridge," vol. 32, no. 2, pp. 20–27. doi: 10.1109/MM.2012.12. [Online]. Available: http://ieeexplore.ieee.org/document/6148200/ [Page 22.]

[44] V. M. Weaver, M. Johnson, K. Kasichayanula, J. Ralph, P. Luszczek, D. Terpstra, and S. Moore, "Measuring energy and power with PAPI," in *2012 41st International Conference on Parallel Processing Workshops*. doi: 10.1109/ICPPW.2012.39 pp. 262–268, ISSN: 2332-5690. [Page 22.]

[45] "Everything is a file," page Version ID: 1148432342. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Everything_is_a_file&oldid=1148432342 [Page 22.]

[46] Hubblo. Scaphandre documentation. Accessed: 2023-04-25. [Online]. Available: https://hubblo.org/scaphandre-documentation/explanations/how-scaph-computes-per-process-power-consumption.html [Page 23.]

[47] B. Ward, *How Linux works: what every superuser should know*, 3rd ed. No Starch Press. ISBN 9781718500402 OCLC: on1233311015. [Page 25.]

[48] Runtime metrics. Accessed: 2023-04-30. [Online]. Available: https://docs.docker.com/config/containers/runmetrics/ [Pages 25 and 26.]

[49] Google. cAdvisor. Accessed: 2023-06-02. [Online]. Available: https://github.com/google/cadvisor [Page 26.]

[50] VMware, "vSphere 6.7.x technical documentation." [Page 26.]

[51] R. Hintemann and S. Hinterholzer, "Cloud computing drives the growth of the data center industry and its energy consumption." [Online]. Available: https://www.borderstep.org/wp-content/uploads/2022/08/Borderstep_Rechenzentren_2021_eng.pdf [Page 37.]

[52] Google. Efficiency – data centers. Accessed: 2023-05-12. [Online]. Available: https://www.google.com/intl/de/about/datacenters/efficiency/ [Page 37.]

[53] Germany: power sector carbon intensity 2000-2021. Accessed: 2023-05-03. [Online]. Available: https://www.statista.com/statistics/1290224/carbon-intensity-power-sector-germany/ [Page 37.]

[54] Prometheus. Storage | prometheus documentation. Accessed: 2023-05-02. [Online]. Available: https://prometheus.io/docs/prometheus/latest/storage/ [Page 38.]

[55] M. Gilleland. Levenshtein distance. Accessed: 2023-05-11. [Online]. Available: https://people.cs.pitt.edu/~kirk/cs1501/Pruhs/Spring2006/assignments/editdistance/Levenshtein%20Distance.htm [Page 45.]

[56] T. Sellam, D. Das, and A. P. Parikh, "BLEURT: Learning robust metrics for text generation." [Online]. Available: http://arxiv.org/abs/2004.04696 [Page 45.]

[57] X. Jiao, Y. Yin, L. Shang, X. Jiang, X. Chen, L. Li, F. Wang, and Q. Liu, "TinyBERT: Distilling BERT for natural language understanding." [Online]. Available: http://arxiv.org/abs/1909.10351 [Page 45.]

[58] K. Liu, K. Mahmoud, J. Yoo, and Y. D. Liu, "Vincent: Green hot methods in the JVM," p. 102962. doi: 10.1016/j.scico.2023.102962. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0167642323000448 [Page 54.]

[59] C. Chandrasekara and P. Herath, *Artifacts and Caching Dependencies*. Apress, pp. 51–61. ISBN 9781484264638 9781484264645. [Online]. Available: http://link.springer.com/10.1007/978-1-4842-6464-5_5 [Page 55.]

[60] G. Maudoux and K. Mens, "Bringing incremental builds to continuous integration." [Online]. Available: https://dial.uclouvain.be/pr/boreal/object/boreal:189543 [Page 55.]

[61] K. Gallaba, J. Ewart, Y. Junqueira, and S. McIntosh, "Accelerating continuous integration by caching environments and inferring dependencies," vol. 48, no. 6, pp. 2040–2052. doi: 10.1109/TSE.2020.3048335. [Online]. Available: https://ieeexplore.ieee.org/document/9311876/ [Page 55.]

[62] A. K. Yadav, M. L. Garg, and Ritika, "Docker containers versus virtual machine-based virtualization," in *Emerging Technologies in Data Mining and Information Security*, A. Abraham, P. Dutta, J. K. Mandal, A. Bhattacharya, and S. Dutta, Eds. Springer Singapore, vol. 814, pp. 141–150. ISBN 9789811315008 9789811315015. [Online]. Available: http://link.springer.com/10.1007/978-981-13-1501-5_12 [Page 56.]

[63] D. Marijan, M. Liaaen, and S. Sen, "DevOps improvements for reduced cycle times with integrated test optimizations for continuous integration," in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*. IEEE. doi: 10.1109/COMPSAC.2018.00012. ISBN 9781538626665 pp. 22–27. [Online]. Available: https://ieeexplore.ieee.org/document/8377636/ [Page 56.]

[64] D. Marijan, A. Gotlieb, and M. Liaaen, "A learning algorithm for optimizing continuous integration development and testing practice: Learning algorithm for CI," vol. 49, no. 2, pp. 192–213. doi: 10.1002/spe.2661. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1002/spe.2661 [Page 56.]

[65] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, and J. Saraiva, "Ranking programming languages by energy efficiency," vol. 205, p. 102609. doi: 10.1016/j.scico.2021.102609. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0167642321000022 [Page 57.]

[66] ——, "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*. ACM. doi: 10.1145/3136014.3136031. ISBN 9781450355254 pp. 256–267. [Online]. Available: https://dl.acm.org/doi/10.1145/3136014.3136031 [Page 57.]

[67] R. Nasim, E. Zola, and A. J. Kassler, “Robust optimization for energy-efficient virtual machine consolidation in modern datacenters,” vol. 21, no. 3, pp. 1681–1709. doi: 10.1007/s10586-018-2718-6. [Online]. Available: http://link.springer.com/10.1007/s10586-018-2718-6 [Page 57.]

[68] R. Abdalkareem, S. Mujahid, E. Shihab, and J. Rilling, “Which commits can be CI skipped?” vol. 47, no. 3, pp. 448–463. doi: 10.1109/TSE.2019.2897300. [Online]. Available: https://ieeexplore.ieee.org/document/8633335/ [Page 57.]

[69] Github. Workflow syntax for GitHub actions. Accessed: 2023-05-23. [Online]. Available: https://ghdocs-prod.azurewebsites.net/en/actions/using-workflows/workflow-syntax-for-github-actions [Page 58.]

[70] ——. Expressions github actions. Accessed: 2023-05-23. [Online]. Available: https://ghdocs-prod.azurewebsites.net/en/actions/learn-github-actions/expressions [Page 58.]

[71] C. Vassallo, S. Proksch, A. Jancso, H. C. Gall, and M. Di Penta, “Configuration smells in continuous delivery pipelines: a linter and a six-month study on GitLab,” in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM. doi: 10.1145/3368089.3409709. ISBN 9781450370431 pp. 327–337. [Online]. Available: https://dl.acm.org/doi/10.1145/3368089.3409709 [Page 58.]

[72] Siemens. Sustainability report 2022. https://assets.new.siemens.com/siemens/assets/api/uuid:c1088e4f-4d7f-4fa5-8e8e-33398ecf5361/sustainability-report-fy2022.pdf. Accessed: 2023-06-15. [Page 61.]

Appendix A

Supporting materials

```
name: Deploy Postgres to EKS

on:
  push:
    branches: [ 'main' ]

  workflow_dispatch:

jobs:
  deploy:
    name: Deploy to EKS
    runs-on: [ Linux ]
    container: alpine/k8s:1.24.12
    steps:
      - name: Check out the repo
        uses: actions/checkout@v3

      - name: Set up Kubeconfig
        env:
          KUBECONFIG_SECRET: ${{ secrets.KUBECONFIG }}
        run: |
          echo "$KUBECONFIG_SECRET" > kubeconfig.yaml
          echo "KUBECONFIG=$(pwd)/kubeconfig.yaml" >>
   $GITHUB_ENV

      - name: Create config map with init script
        run: |
          kubectl create configmap postgres-init-scripts --
   from-file=postgres/init.sql --namespace alm-energymonitor
   --dry-run=client -o yaml | kubectl apply -f -

```

```
    - name: Add helm repo
      run: |
        helm repo add bitnami https://charts.bitnami.com/bitnami
        helm repo update

    - name: Deploy postgres helm chart to kubernetes cluster
      run: |
        helm upgrade --install --kubeconfig=kubeconfig.yaml postgres bitnami/postgresql --namespace alm-energymonitor -f postgres/values.yaml
```

Listing A.1: Github Action Pipeline: Deploy Postgres to a Kubernetes cluster using Helm Chart

Appendix B

Log analyser

Figure B.1 exhibits a Grafana dashboard that offers a comprehensive view of all monitored repositories. It presents the aggregate count of runs, steps, and jobs along with the number of runners deployed. The bar plots indicate the repositories with the highest number of pipeline runs and the total time spent running CI/CD pipelines for each repository.

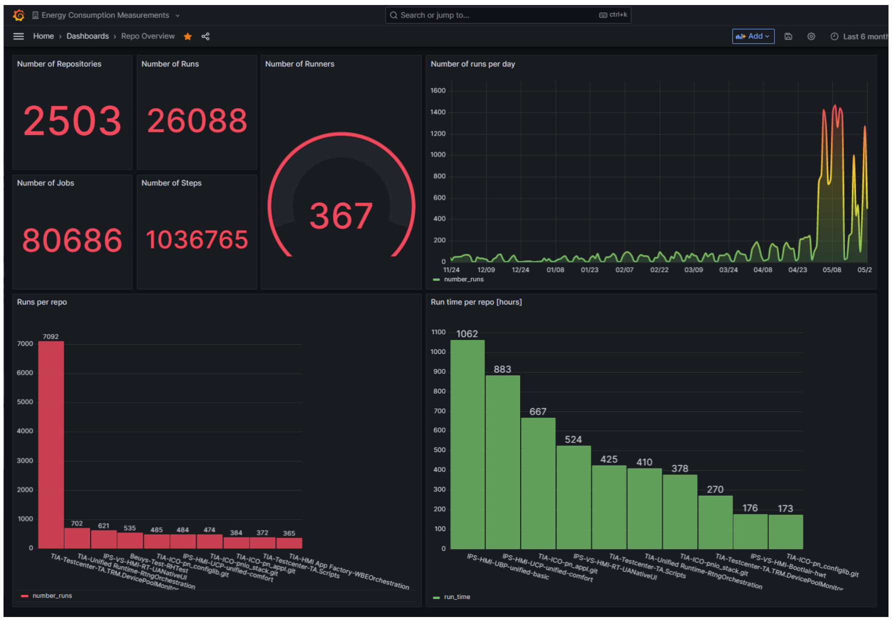


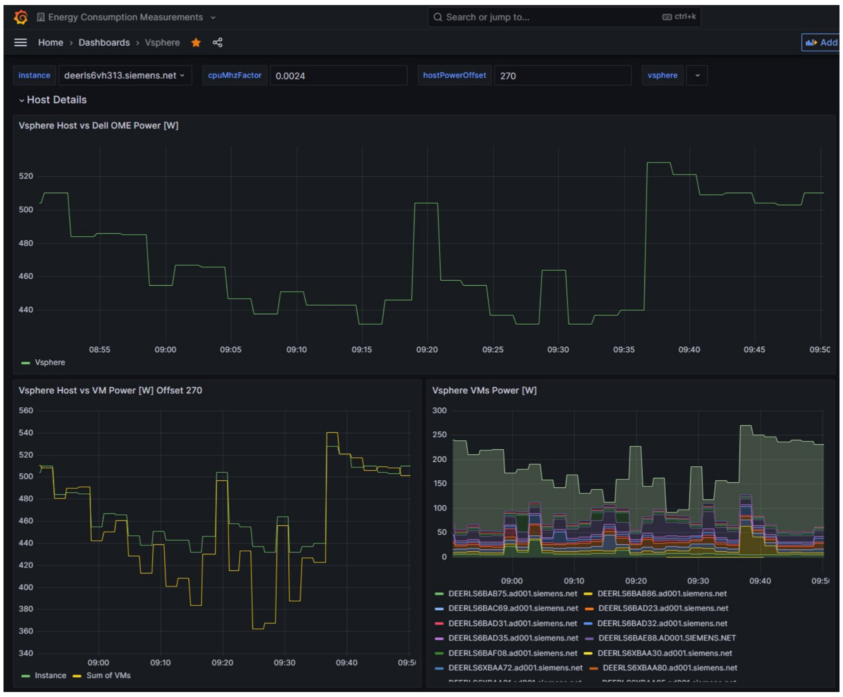
Figure B.1: Planetary dashboard implemented with Grafana: Real-time monitoring of the repositories

Appendix C

Resource collectors

C.1 Resource monitoring

Figure C.1 and Figure C.2 illustrate the real-time tracking of power values gathered from vSphere and Dell Open Manage. Moreover, they display the proportional values for resources (CPU, Memory, Network) used by all VMs operating on the server.


Energy Consumption Measurements
Search or jump to...
ctrl+k
Home › Dashboards › Vsphere
Add
Host Details
Vsphere Host vs Dell OME Power [W]
Vsphere
Vsphere Host vs VM Power [W] Offset 270
Instance
Sum of VMs
Vsphere VMs Power [W]
DEERLS6BAB75.ad001.siemens.net
DEERLS6BAB86.ad001.siemens.net
DEERLS6BAC69.ad001.siemens.net
DEERLS6BAD23.ad001.siemens.net
DEERLS6BAD31.ad001.siemens.net
DEERLS6BAD32.ad001.siemens.net
DEERLS6BAD35.ad001.siemens.net
DEERLS6BAE88.AD001.SIEMENS.NET
DEERLS6BAF08.ad001.siemens.net
DEERLS6XBAA30.ad001.siemens.net
DEERLS6XBAA72.ad001.siemens.net
DEERLS6XBAA80.ad001.siemens.net

instance deerls6vh313.siemens.net
cpuMhzFactor 0.0024
hostPowerOffset 270
vsphere

Figure C.1: Planetary dashboard implemented with Grafana: Real-time power measurements of a Dell PowerEdge server and the VMs running on it.

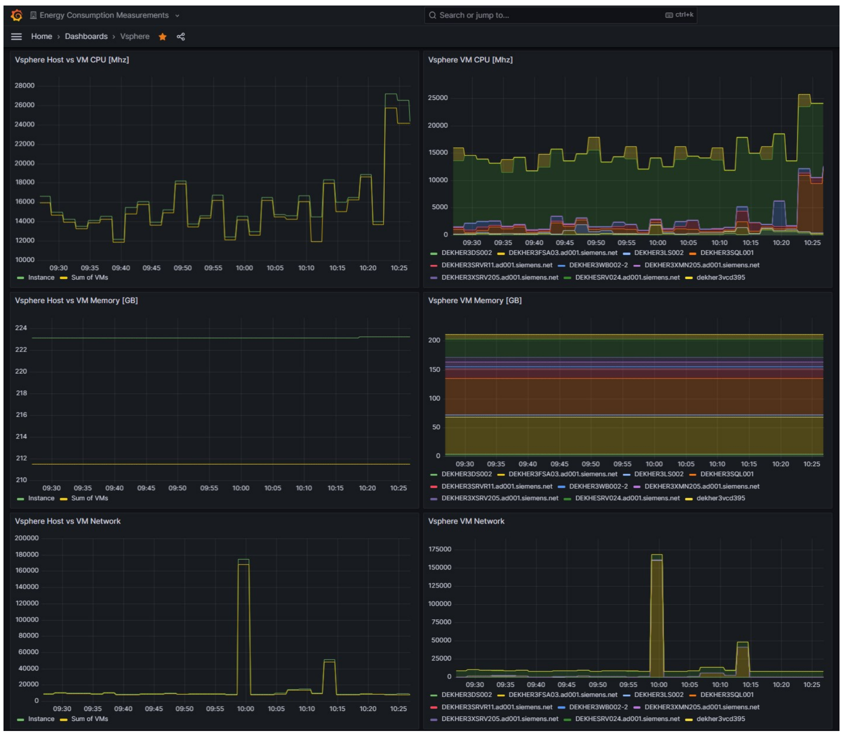

Energy Consumption Measurements
Search or jump to...
Home › Dashboards › Vsphere
Vsphere Host vs VM CPU [Mhz]
Vsphere VM CPU [Mhz]
Vsphere Host vs VM Memory [GB]
Vsphere VM Memory [GB]
Vsphere Host vs VM Network
Vsphere VM Network
Instance
Sum of VMs
DEKHER3DS002
DEKHER3FSA03.ad001.siemens.net
DEKHER3LS002
DEKHER3SQL001
DEKHER3SRVR11.ad001.siemens.net
DEKHER3WB002-2
DEKHER3XMN205.ad001.siemens.net
DEKHER3XSRV205.ad001.siemens.net
DEKHESRV024.ad001.siemens.net
dekher3vcd395

Figure C.2: Planetary dashboard implemented with Grafana: Real-time CPU, Memory and Network usage of the host and the VMs running on it

C.2 Resource analysis

Figure C.3 demonstrates the power metrics recorded from vSphere (illustrated in blue) and those gathered from Dell Open Manage (represented in orange) for a representative server. The values showcase high correlation exceeding 0.95. In some cases, however, we were able to detect a slight delay in the values (between 0 and 20 seconds).

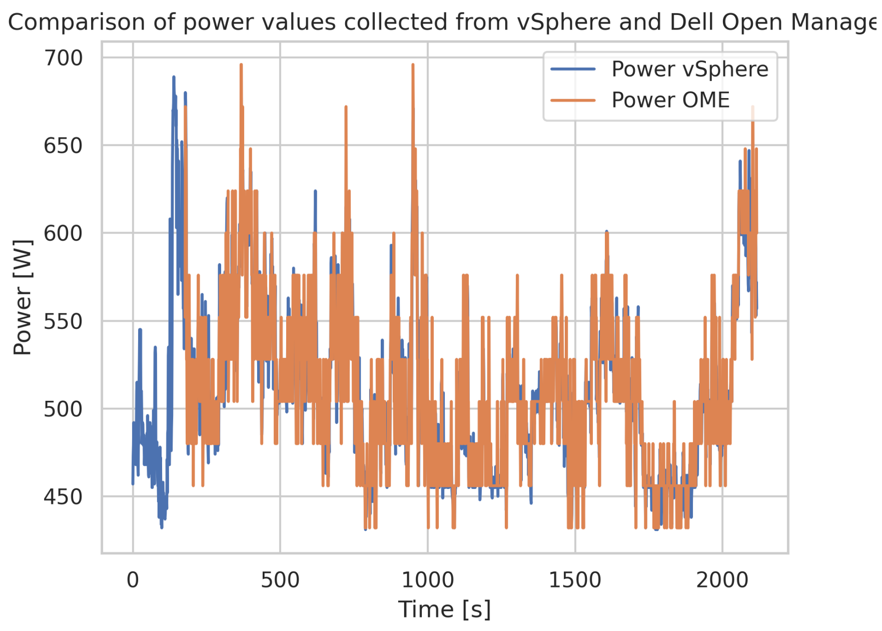


Figure C.3: Comparison of power values collected with vSphere and Dell Open Manage

Figure C.4 illustrates the correlation between relevant metrics, obtainable from individual VMs via the vSphere API, and their respective power values. The highest correlation (by far) is to CPU usage, which is given as a relative and absolute value.

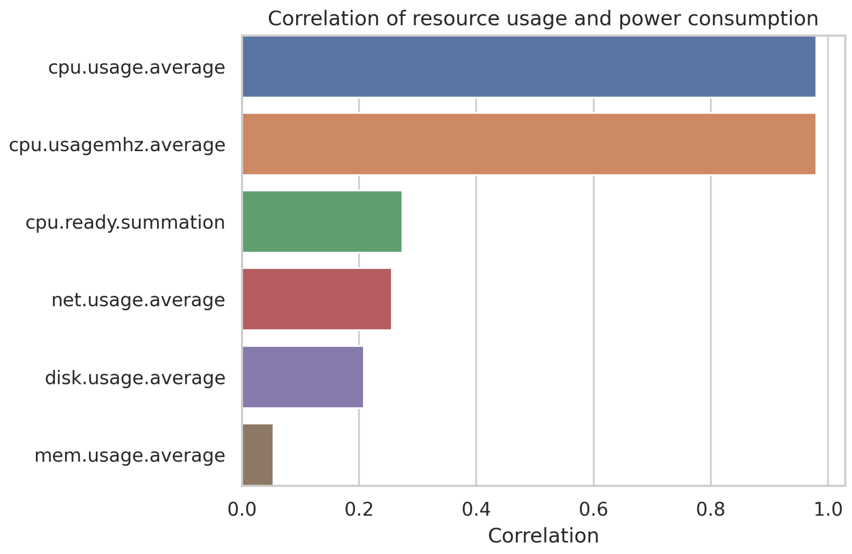


Figure C.4: Correlation of relative resource metrics and power consumption of VMs

Employing a Random Forest Regressor from Scikit Learn, we estimated the power values of individual VMs with an average error of 5% , utilizing CPU, network, and memory utilization metrics compared to the exposed power values from vSphere. The predicted values vs. the values exposed by vSphere are shown in Figure C.5. This model is strictly for validation purposes.

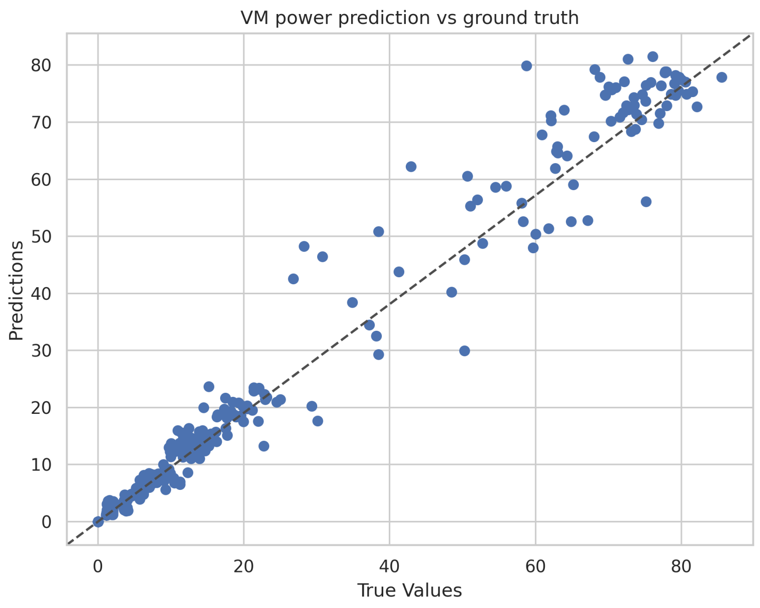


Figure C.5: Comparison of predicted VM power values using Random Forest Regressor and CPU, network and memory usage as input against power values provided by vSphere.

TRITA-EECS-EX-2023:521

€€€€ For DIVA €€€€

{
”Author1”: { ”Last name”: ”Limbrunner”,
”First name”: ”Nikolai”,
”E-mail”: ”ntli@kth.se”,
”organisation”: {”L1”: ”School of Electrical Engineering and Computer Science”,
}
},
”Cycle”: ”2”,
”Course code”: ”DA231X”,
”Credits”: ”30.0”,
”Degree1”: {”Educational program”: ”Master’s Programme, Computer Science, 120 credits”
,”programcode”: ”TCSCM”
,”Degree”: ”Degree of Master of Science in Engineering”
,”subjectArea”: ”Computer Science and Engineering”
},
”Title”: {
”Main title”: ”Dynamic macro to micro scale calculation of energy consumption in CI/CD pipelines”,
”Language”: ”eng” },
”Alternative title”: {
”Main title”: ”Dynamisk beräkning av energiförbrukning i CI/CD-pipelines från makro- till mikroskala”,
”Language”: ”swe”
},
”Supervisor1”: { ”Last name”: ”Kugler”,
”First name”: ”Christopher”,
”E-mail”: ”christopher.kugler@siemens.com”,
”Other organisation”: ”Siemens”
},
”Supervisor2”: { ”Last name”: ”Martinez”,
”First name”: ”Matias”,
”E-mail”: ”matias.martinez@uphf.fr”,
”organisation”: {”L1”: ”School of Electrical Engineering and Computer Science”,
”L2”: ”Computer Science” }
},
”Examiner1”: { ”Last name”: ”Monperrus”,
”First name”: ”Martin”,
”E-mail”: ”monperrus@kth.se”,
”organisation”: {”L1”: ”School of Electrical Engineering and Computer Science”,
”L2”: ”Computer Science” }
},
”Cooperation”: { ”Partner_name”: ”Siemens”},
"National Subject Categories": "10201, 10206",
”Other information”: {”Year”: ”2023”, ”Number of pages”: ”1,??”},
”Copyrightleft”: ”None”,
”Series”: { ”Title of series”: ”TRITA-EECS-EX” , ”No. in series”: ”2023:0000” },
”Opponents”: { ”Name”: ”Johanna Nilsen”},
”Presentation”: { ”Date”: ”2023-06-14 09:00”
,”Language”:”eng”
,”Room”: ”via Zoom https://kth-se.zoom.us/j/68775837061”
,”City”: ”Stockholm” },
”Number of lang instances”: ”2”,
”Abstract[eng]”: €€€€

This thesis applies energy measurements to the domain of continuous integration (CI) and continuous delivery (CD) pipelines. The goal is to conduct transparent and fine-granular energy measurements of these pipelines, increasing awareness and allowing optimizations regarding their energy efficiency. CI and CD automate processes like compilation, running tests, and code analysis tools and can improve the software quality and developer experience and enable more frequent releases. Initially, the applicability of existing energy measurement approaches for these tasks is analyzed. Afterward, a generic framework consisting of a pipeline run analyzer, a resource consumption collector, and an energy calculator is proposed. A representative implementation for a state-of-the-art infrastructure is devised to demonstrate its functionality, enabling the collection, analysis, and interpretation of data from real-world examples. Finally, it is examined whether this data aligns with the theoretical considerations and can be used to optimize the pipelines. The overall goal is to contribute to the sustainability of DevOps processes and therefore counteract the disastrous consequences of unrestrained climate change.

€€€€,
”Keywords[eng]”: €€€€
GreenIT, Sustainability, Energy measurement, Software Engineering, CI/CD pipelines, DevOps €€€€,
”Abstract[swe]”: €€€€

Denna avhandling tillämpar energimätningar på området kontinuerlig integration (CI) och kontinuerlig leverans (CD).
Målet är att genomföra transparenta och finkorniga energimätningar av dessa pipelines, vilket ökar medvetenheten och möjliggör optimeringar av deras energieffektivitet. CI och CD automatiserar processer som kompilering, testkörning och kodanalysverktyg och kan förbättra programvarukvaliteten och utvecklarens upplevelse samt möjliggöra tätare lanseringar.

Inledningsvis analyseras tillämpligheten av befintliga metoder för energimätning för dessa uppgifter. Därefter föreslås ett generiskt ramverk som består av en analysator för pipelinekörning, en insamlare av resursförbrukning och en energikalkylator.
För att demonstrera dess funktionalitet utarbetas en representativ implementering för en modern infrastruktur som möjliggör insamling, analys och tolkning av data från verkliga exempel. Slutligen undersöks om dessa uppgifter stämmer överens med de teoretiska övervägandena och kan användas för att optimera rörledningarna. Det övergripande målet är att bidra till hållbarheten i DevOps-processer och därmed motverka de katastrofala konsekvenserna av ohämmade klimatförändringar.

€€€€,
”Keywords[swe]”: €€€€
GreenIT, hållbarhet, energimätning, programvaruteknik, CI/CD-pipelines, DevOps €€€€,
}