Full Stack Emissions Reporting

A systematic approach

Introduction

The NADIKI project's main objective is to develop a comprehensive approach to emissions reporting for software developers. The aim is to collect all emission's input, both static and dynamic and aggregate them for software developers to influence their choices and change their behavior.

Software is complex. And the software development practice is a continuous process of delivering value for the end user, whether human or machine. If we can inform the software developer (*developer* can be changed to any other role in the domain of software development) about the environmental impact of their choices they will factor that in.

Our approach is to collect as much data as we can, and make reasonable assumptions for the data we lack. The static data is generally of the type of embedded emissions like buildings and other necessary physical infrastructure. These will be amortized, with a depreciation depending on the expected (not advertised) lifetime of these assets.

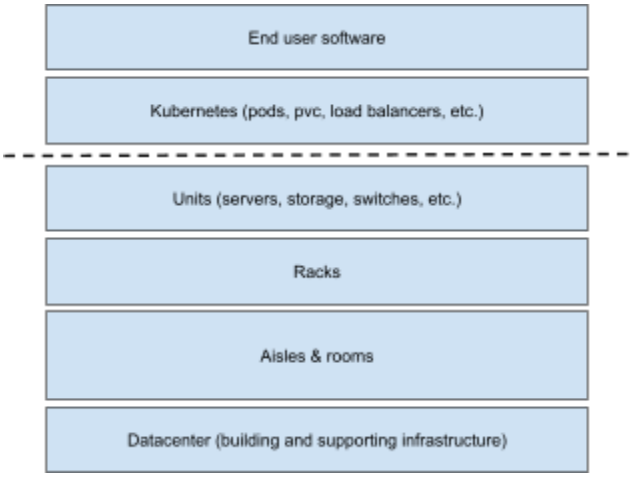
Dynamic data are the continuous inputs necessary to make 'the place run'. These include energy to power servers and supporting equipment like networking and air conditioning. But also water in case of liquid cooling.

Lastly, the energy re-use and/or renewable energy locally generated will be factored into the emissions.

As we expect certain emissions to be hard to measure (or hard to expose) the most important is to get the whole stack (Full Stack) represented in our approach. It is not important to get everything right, but we have to have everything in.

In the rest of this document we will describe the Full Stack, Emissions Reporting, the Full Stack Model and the Full Stack Architecture. To ensure a pragmatic approach to developing this solution we will let the Data model be the API, and only define the (REST) API when we have a full working set-up.

The Full Stack

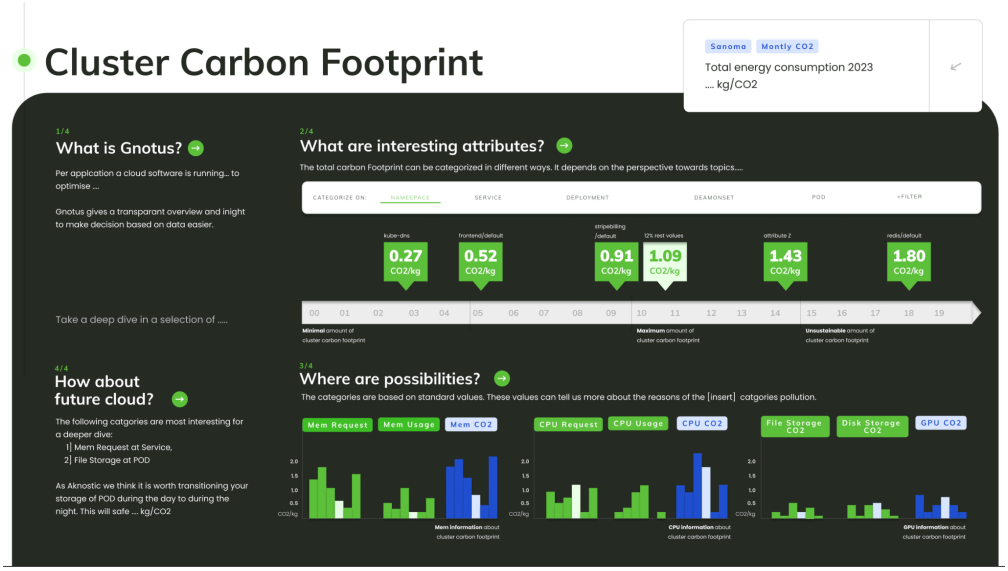


As you can see in this schematic there is a lot going into getting end users their software. For sake of simplicity we have taken the most common datacenter layout with aisles, rooms and racks. There are other configurations, but they are outside the scope of our work.

Working from the top we see ‘End user software’. This is the domain for the software developer. A software developer needs to understand emissions in their own context. This means they require reporting on the level of individual components, both aggregated (namespace, cluster, etc.) and at commodity level (pods, pvc, load balancer, etc.) The Kubernetes level is responsible for aggregating all underlying emissions inputs, attributing them to the commodities that make up the end user software. We assume that both these layers do not generate additional emissions, static or dynamic.

The rest is relatively straightforward to understand. But, not all of these different assets or containers have their own measurements so they might have to be derived. The best way to get a realistic emissions collection is to use measuring tools/devices.

Emissions Reporting



Software developers think logically. Their product consists of a collection of tools and components that interact to bring value to the end user (remember, both human and machine). Tools can be databases, caching, streaming, serverless, etc. Components can be APIs, CFAs, LLMs, etc. These are the things that make up the world of a software developer.

To change their behavior and influence their decisions we need to stack, rank and suggest. If we can show that a MySQL database is the nr. 1 emitter they will look at query optimization, caching strategies, tools choice (other database) and/or architecture changes (move to no-SQL).

If the nr. 1 emitter is the LLM the software developer needs to understand if this is energy or the embedded emissions of their servers. If it is energy the workload can be migrated to an area where the energy mix is more favorable to renewable energy. If the embedded emissions are causing the most harm, a move to refurbished hardware or a commodities provider with a longer lifespan of their hardware is warranted.

The Full Stack Model

The Full Stack Model is a direct representation of the bottom four layers in our Full Stack. The companion PDF is a definition of this model. The description of the objects in the model is the following

datacenter the physical building or space containing one or more aisles, in optional rooms
aisle a row of zero or more racks
room a closed space with zero or more racks

rack a cabinet with zero or more units
unit a physical enclosure that has zero or more switches, compute units or storage units
switch networking equipment
compute compute (server, blade or board)
storage disk (jbod, raid, tape, etc.)

The Full Stack Architecture

The first phase in this project is collecting all the data from the bottom 4 models. The Full Stack Model is a good definition to start with. But we should treat it as work in progress and not hesitate to adapt to reality.

For the first phase we need a component to collect the data. We propose to use prometheus for this. The interface to prometheus is well defined and this is a standard and scalable way to collect huge amounts of data.