

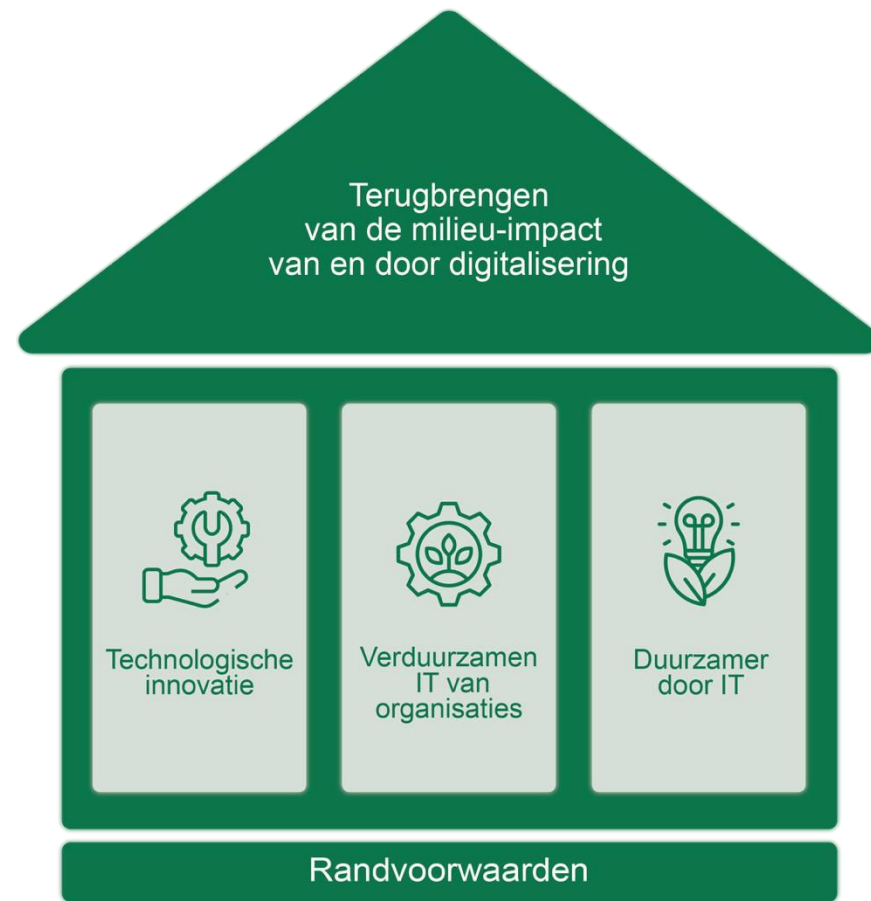
Webinar

**Duurzame software:
de impact die je niet mag onderschatten**





HUIS VAN DUURZAME DIGITALISERING





Duurzame Software:

De impact die je niet mag onderschatten

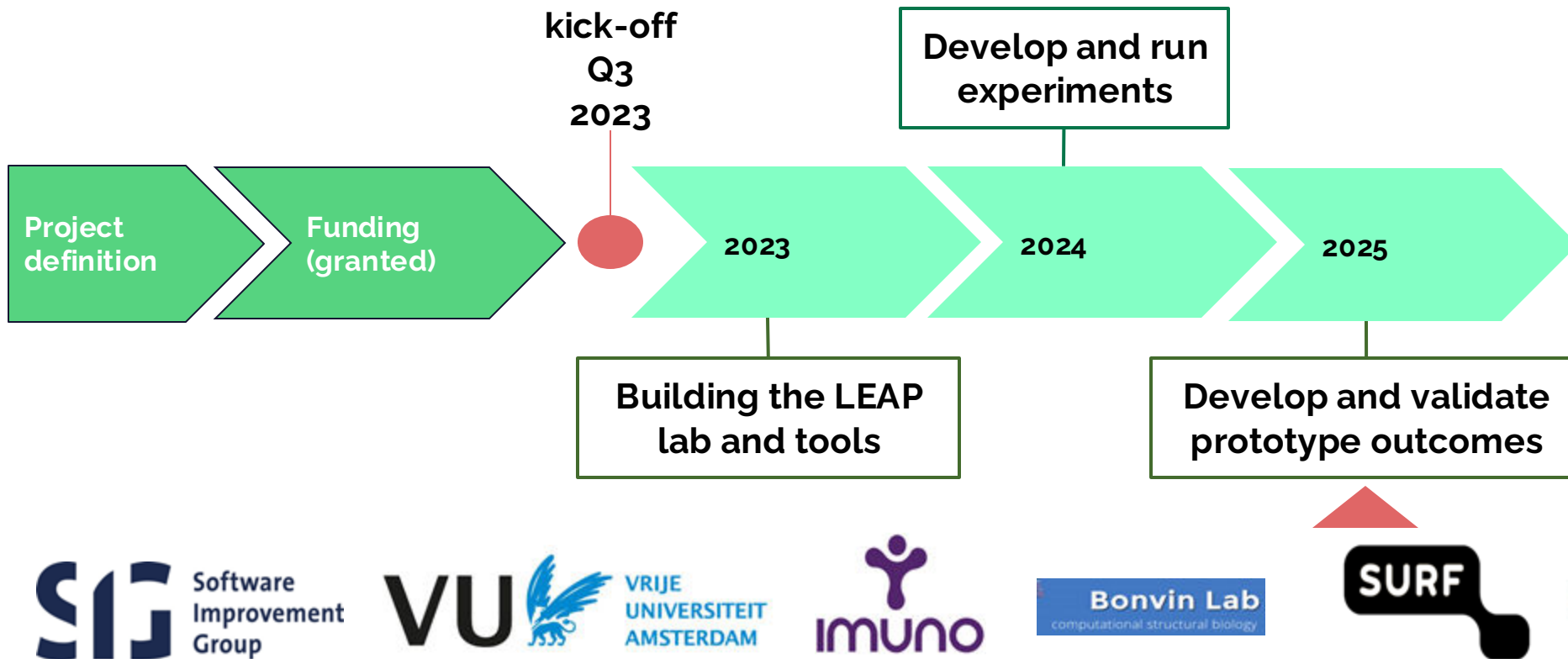
Pepijn van de Kamp

Director of Research & Innovation

LETSGO: Low Energy Acceleration Programme (LEAP)

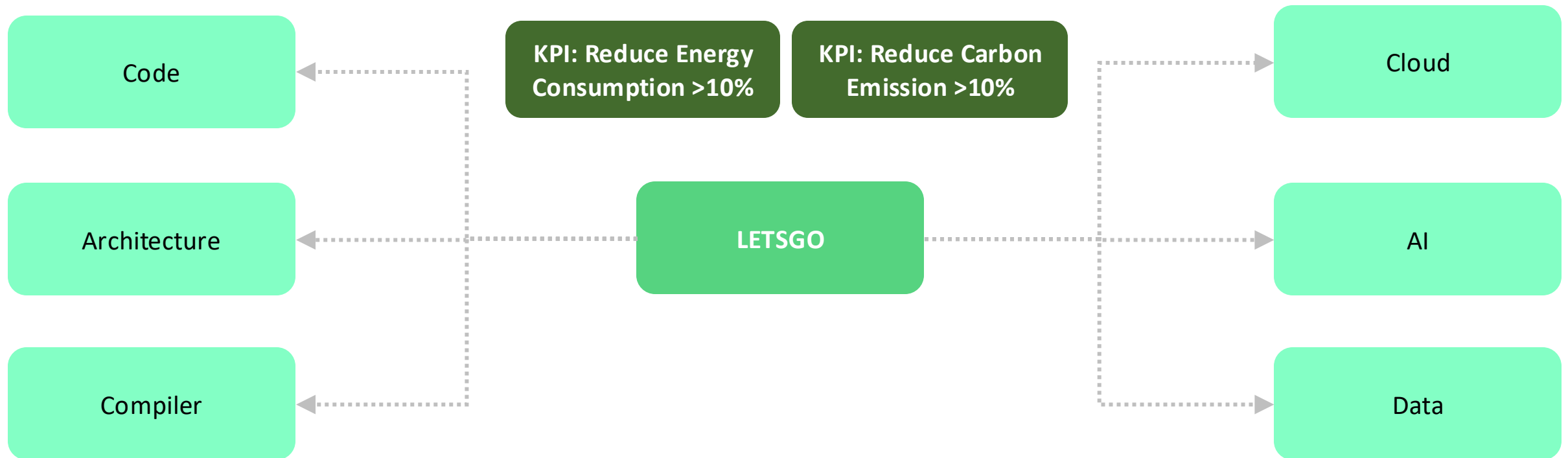
Experiments on Tuning Software-systems for Greener Operations

The LETSGO project aims to develop tools, and showcase tactics and best practices for reducing the energy consumption and carbon footprint of software applications that are migrated to - or running in cloud environments



LETSGO: Overview of Deliverables

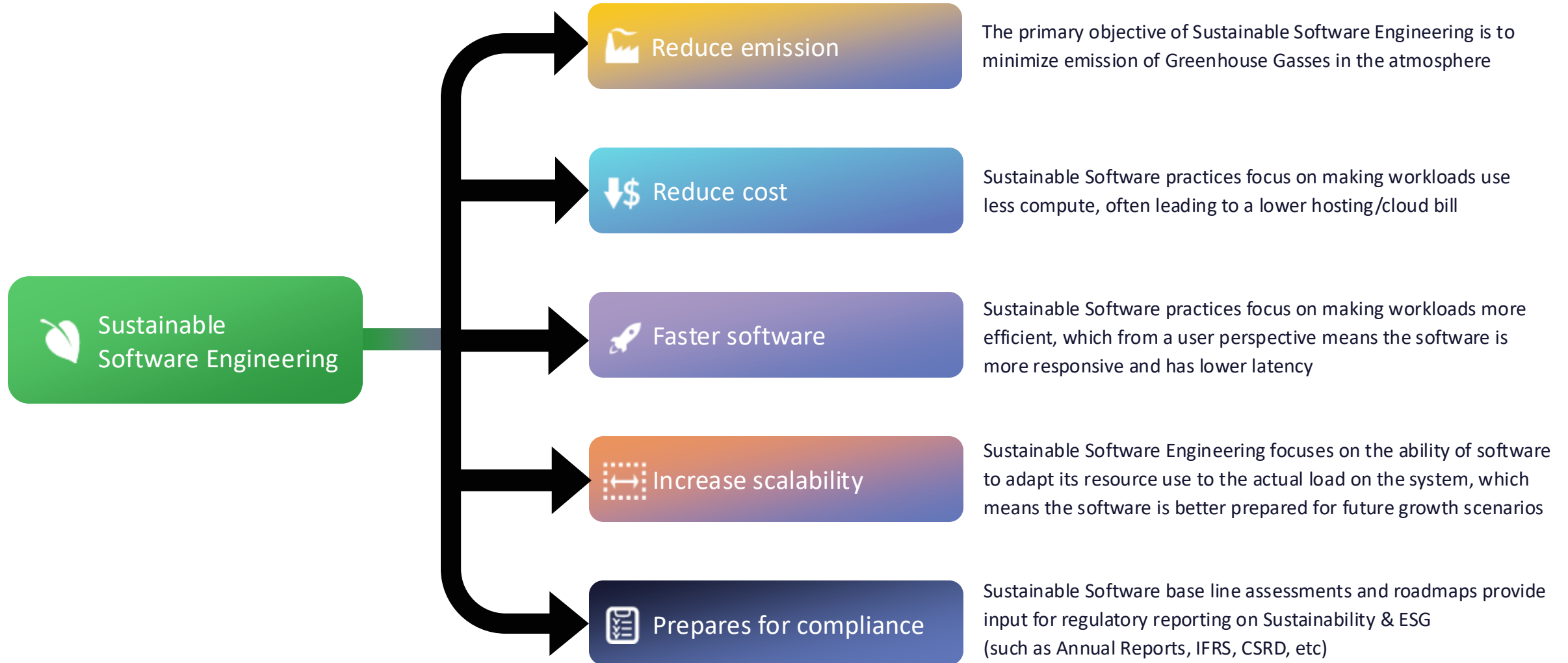
- Scientific contributions: 8 academic publications, 7 master thesis projects
- SIG: Green Software Model, Green Code Module in Sigrid Software Governance Platform (this presentation)
- **Final presentation: NCDD Conference on Sept 29**



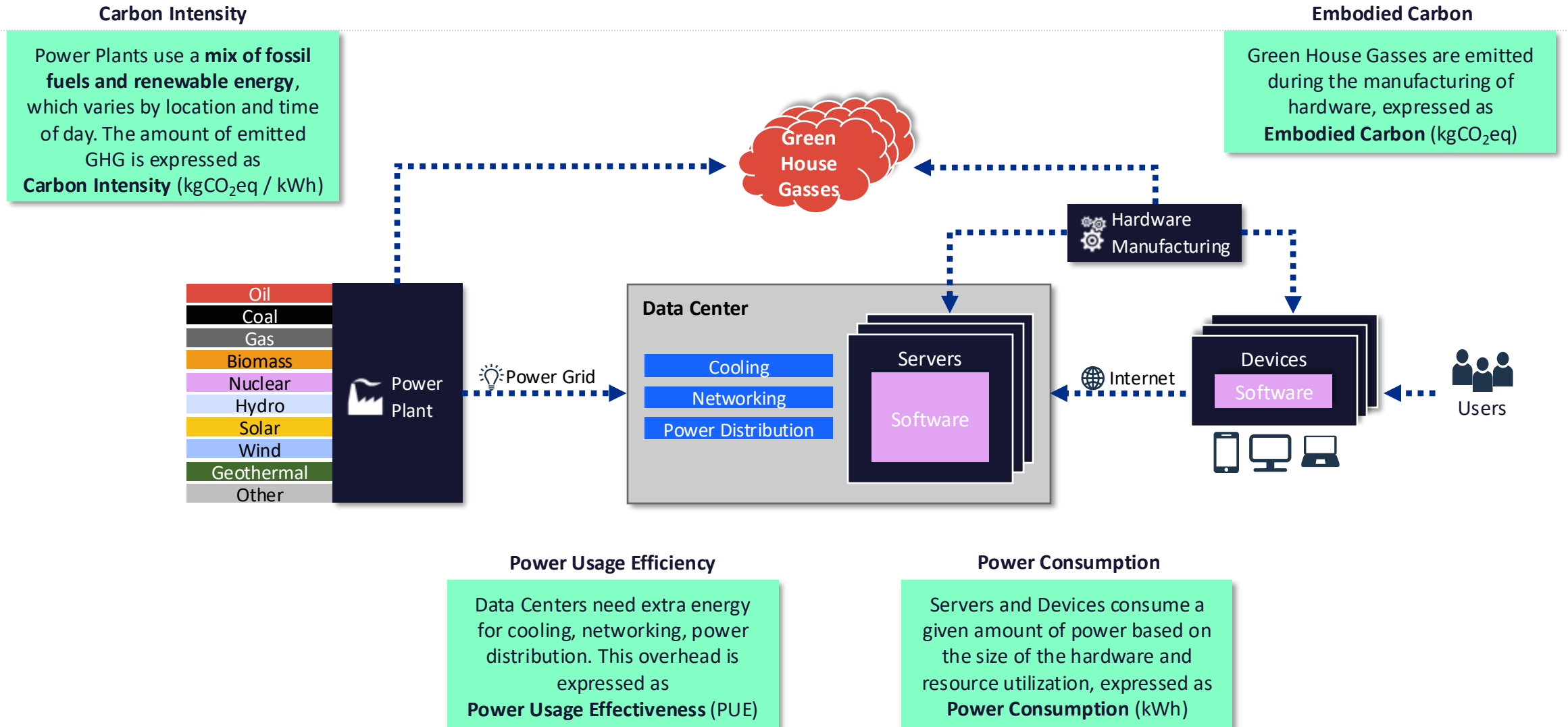
Statistics on the State of Data Center Carbon Emissions

- **Today, 40% of global carbon emissions originate from power plants**
- **Data centers consume 4% of global electricity, estimated to grow to 14% in 2030**
- **Global carbon emissions from data centers surpass that of the entire aviation industry**
- **Digital Services companies report that >50% of their carbon emissions come from IT**
- **Datadog: 83% of container spend is wasted on idle resources**

Sustainable Software Engineering is aligned with other Business Objectives



How does software emit Green House Gasses?



The ISO/IEC 21031 for Software Carbon Intensity has 3 sub-characteristics



Energy Efficiency

The extent to which a system is optimized to minimize energy consumption

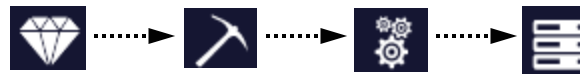


“Optimize code and architecture and increase scalability to reduce energy consumption”



Hardware Efficiency

The extent to which the hardware setup is designed to have the smallest possible carbon footprint during the life cycle of the system



“Right-size hardware and consider the amount of carbon emitted during both its production and operation”



Carbon Awareness

Ability for the system to time-shift or region-shift computations to take advantage of cleaner or renewable sources of electricity



“Use more compute when renewable energy is abundant and less when energy is generated from fossil fuels”

Reduce emissions by up to 43% in NL by applying Time Shifting of Batch Jobs

Statistics of the Dutch Energy Grid

The average carbon intensity of the energy grid in **The Netherlands** in 2024 was 274 gCO₂eq/kWh

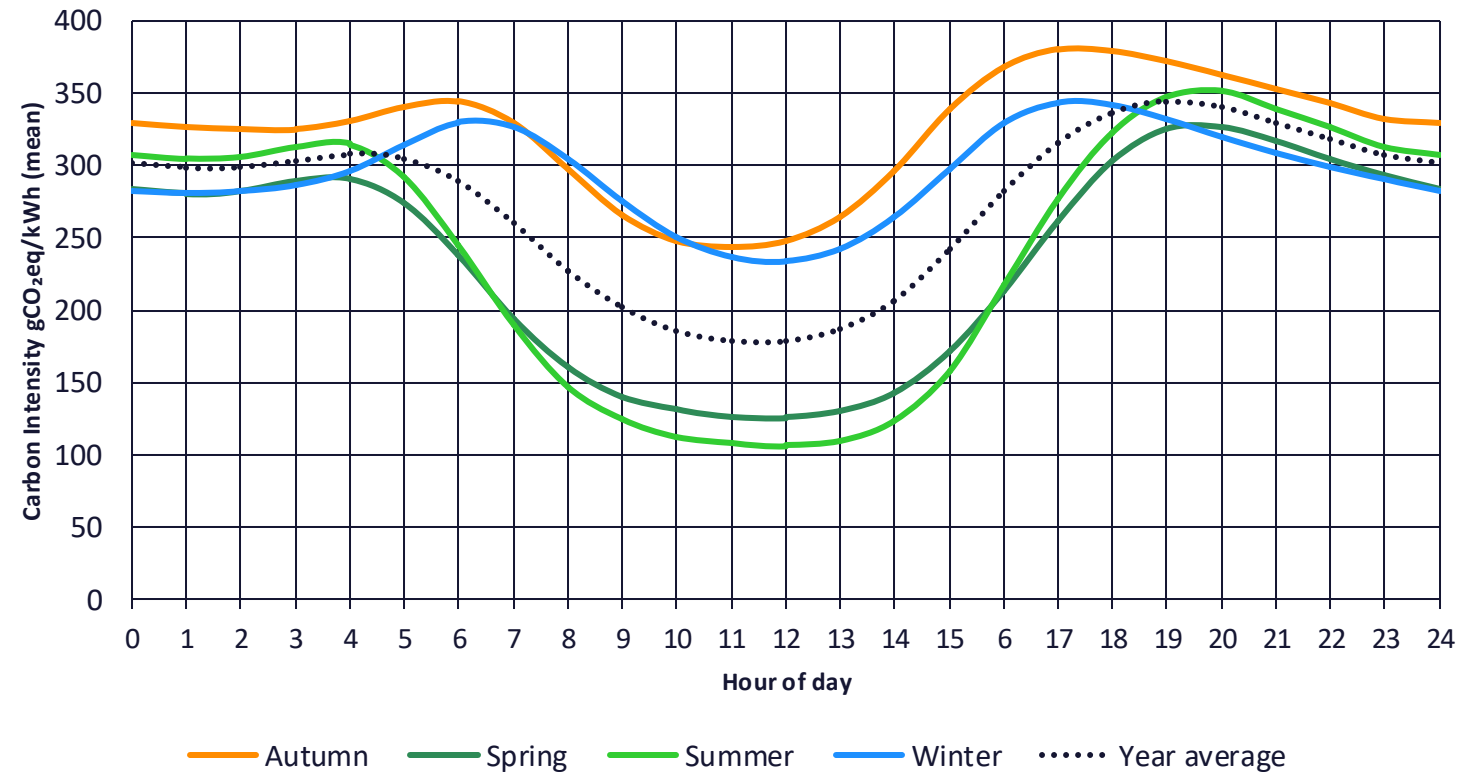
On average, the Dutch energy grid has the **lowest carbon intensity** between **11:00** and **12:00**

Simulating Time Shifting Strategies

Scheduling batch jobs in a **static time window** (11:00-12:00) can **reduce carbon emissions** by **36%** compared to the average grid intensity

Scheduling batch jobs dynamically based on carbon intensity **forecasting** (e.g. WattTime API) can **reduce carbon emissions** by **43%**

Grid Carbon Intensity by Hour of Day in The Netherlands (2024)



Source: ⚡ ELECTRICITY MAPS

Choosing a Hosting Region with the lowest possible Carbon Intensity



Location awareness

Where in the world software is hosted has a significant impact on the carbon emission of your software

Recommendation: Choose a hosting location based on both business requirements and sustainability requirements

Anti-pattern: Only consider hosting location based on the location of users

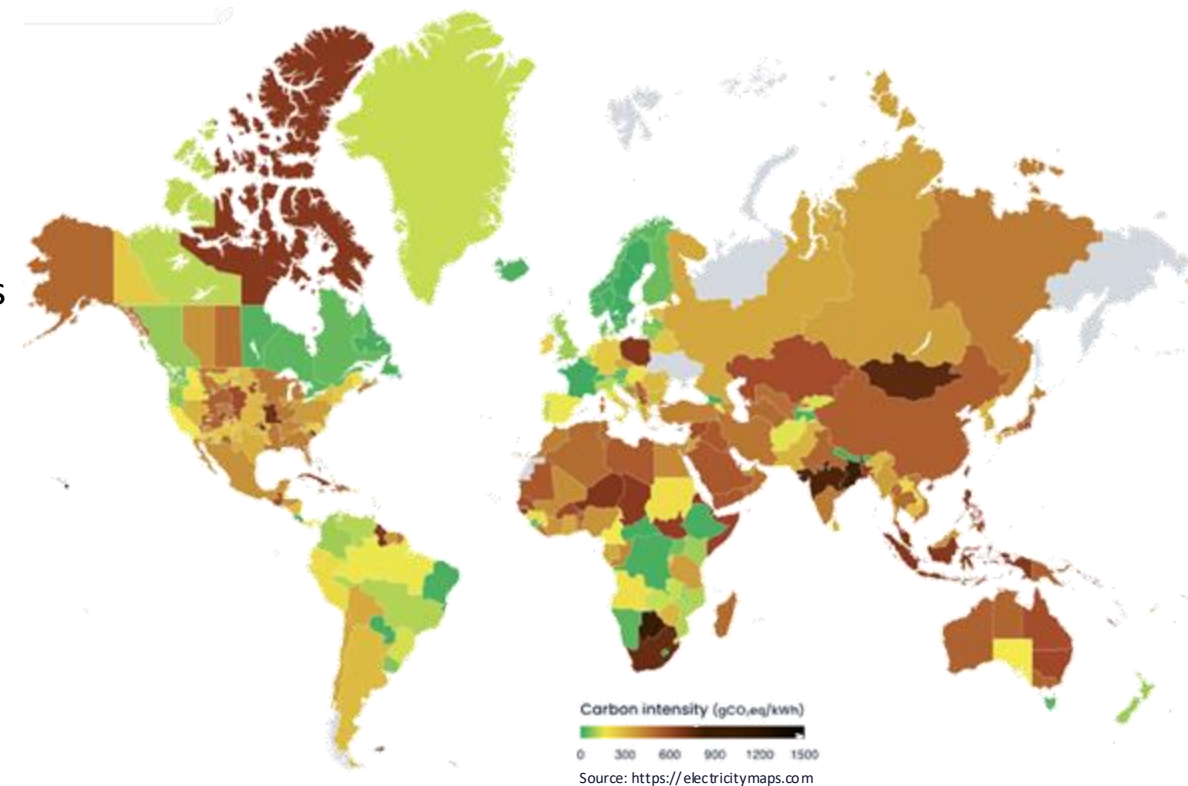
Potential limitations: data residency requirements (e.g. data must reside in the EU or US)

Examples

- Electricity consumed in EU North (Sweden) has a **12x lower carbon intensity** than electricity consumed in EU Central (Germany)
- Electricity consumed in US West (Oregon) has an almost **6x lower carbon intensity** than electricity consumed in US East (Virginia)

Average carbon intensity per region world-wide (2024)

(carbon intensity = amount of CO₂ emitted per generated kWh of electricity)



Green Cloud Practices can significantly reduce both Operational Carbon Emissions and Cost

Example calculation of Emission and Cost reduction

Migrating a single server from on-premise to public cloud (lift & shift)
 Additional Green Cloud Practices can be applied depending on context

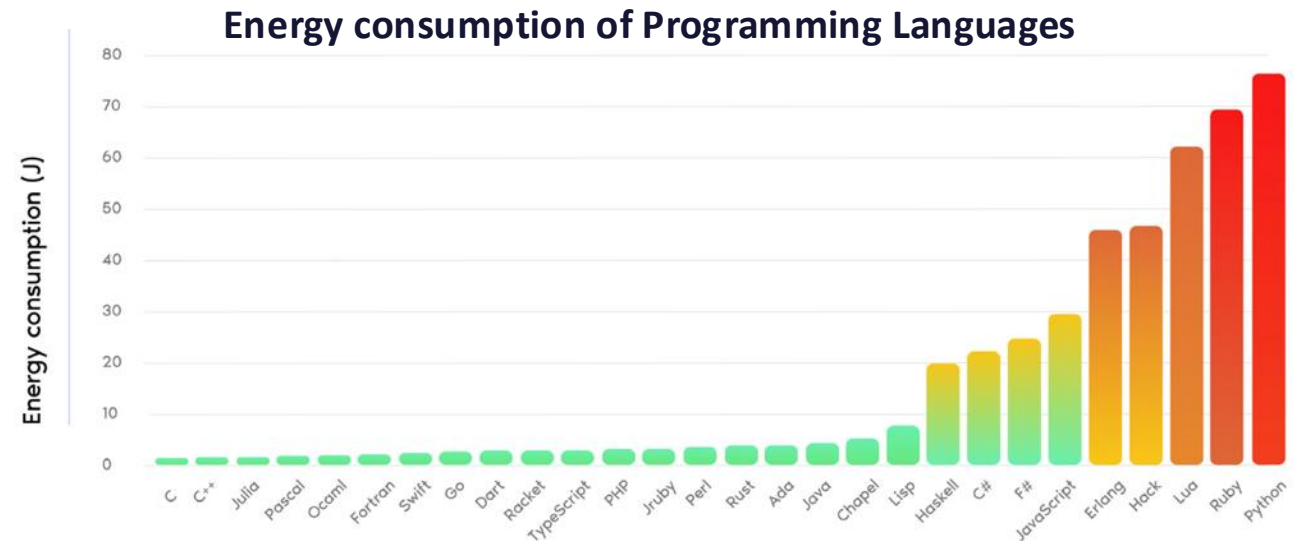
		Operational Carbon Emission		Cost Per Server	
		Yearly emission (kgCO ₂ eq)	CO ₂ Reduction (%)	Yearly Cost (USD)	Cost Savings (%)
Green Cloud Practices ↓	On-prem dedicated server (DELL 4CPU/16GB on-prem in Netherlands, 5 year lifespan)	3,300 kg	-	\$ 2,635	-
	Lift and shift to public cloud VM (XL 4vCPU/16GB in AWS Frankfurt)	415 kg	- 87%	\$ 1,282	- 51%
	Move to green cloud region (XL 4vCPU/16GB in AWS Paris)	31 kg	- 92%	\$ 1,244	- 3%
	Optimize Software and Right-size VM (XL to L: 2 vCPU/8GB in AWS Paris)	16 kg	- 50%	\$ 622	- 50%
	Use Efficient Chipset (ARM) (Graviton2 2 vCPU/8GB in AWS Paris)	10 kg	- 38%	\$ 477	- 23%
	Power Scheduling (shut down in weekends, Graviton2 2 vCPU/8GB in AWS Paris)	7 kg	- 28%	\$ 341	- 29%
Total reduction compared to On-prem		- 3,293 kg	- 99.8%	- \$ 2,294	- 87%
Total reduction compared to Lift and Shift		- 408 kg	-98.3%	- \$ 941	- 73%

More and more research becomes available on the Energy Efficiency of Programming Languages

- The combination of **programming language and runtime** has a large impact on the energy consumption of a system
- **Recommendation:** Choose the set of programming languages to be used for a software system consciously, taking multiple factors into account, including Sustainability

Benchmark data shows interpreted **Python** consumes **38x** more energy than **Java**, but...

Energy Efficiency of **Python** can be improved **19x** when **compiling Ahead of Time**



Simple Code Refactorings can significantly reduce Energy Consumption

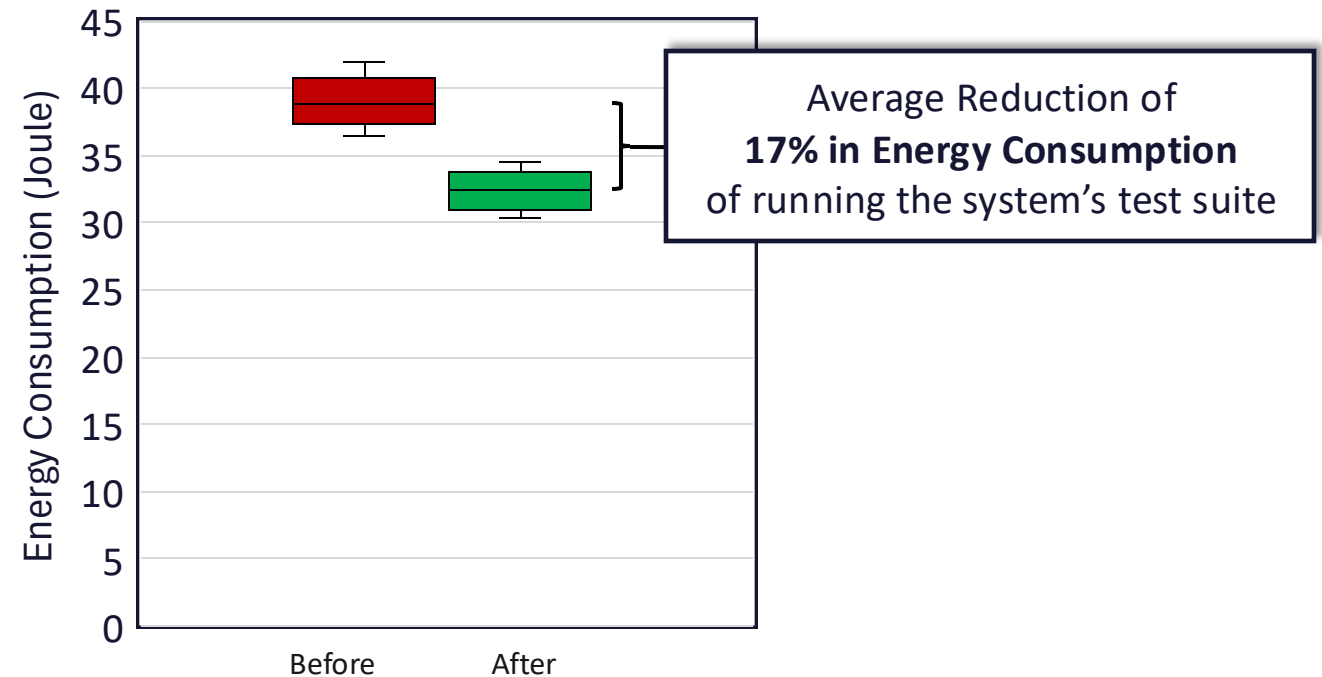
Refactoring

Replacing old standard Java Collections with new counter parts in an open-source system

```

8 public class WordCounter {
9     public static List<String> calculateNGrams(List<String> words, int n) {
10-        List<String> nGrams = new Vector<>();
10+        List<String> nGrams = new ArrayList<>();
11
12         for (int i = 0; i < words.size() - n + 1; i++) {
13             nGrams.add(words.subList(i, i + n).toString());
14         }
15
16         return nGrams;
17     }
18
19     public static Map<String, Integer> countFrequency(List<String> words) {
20-        Map<String, Integer> count = new Hashtable<>();
20+        Map<String, Integer> count = new HashMap<>();
21
22         for (String word : words) {
23             count.put(word, count.getDefault(word, 0) + 1);
24         }
25
26         return count;
27     }
28 }
  
```

Vector was replaced by *ArrayList*
Hashtable was replaced by *HashMap*



Real world case: Reducing execution time by 90% (from 10 hours to 1 hour)

- One of our graph processing algorithms took 10 hours to run on a very large graph
- After profiling the system our engineers found the culprit: an inefficient algorithm (n^2)
- Changing a few lines of code solved the problem (precompute values in a `HashMap<>`, which can lookup values in constant time)

Code Refactoring of an Algorithmic Inefficiency

```

27 + private Map<Pair<String, String>, Dependency> dependenciesBySourceAndTarget;
- private Optional<Dependency> findEdge(String fromComponent, String toComponent, List<Dependency> edges) {
-     return edges.stream()
-         .filter(edge -> edge.getSource().equals(fromComponent) && edge.getTarget().equals(toComponent))
-         .findFirst();
63 + private Optional<Dependency> findEdge(String fromComponent, String toComponent) {
64 +     return Optional.ofNullable(dependenciesBySourceAndTarget.getOrDefault(Pair.of(fromComponent, toComponent), null));
65 }
66
67 private Set<Dependency> getViolationGraph(DefaultDirectedWeightedGraph<String, DefaultWeightedEdge> ddwg,
69     return allPaths.stream()
70         .flatMap(path -> path.getEdgeList().stream())
-         .map(e -> findEdge(ddwg.getEdgeSource(e), ddwg.getEdgeTarget(e), dependencies))
71 +         .map(e -> findEdge(ddwg.getEdgeSource(e), ddwg.getEdgeTarget(e)))
73         .collect(toSet());
74 }
  
```

Calling
stream()...findFirst()
from another stream.

Time complexity = n^2

Introduce HashMap
with precomputed
values

Lookup value in map
in constant time

Refactored code

Original code

This is not just an optimization, but rather a paradigm shift in the efficiency of this algorithm!

Introducing Green Coding rules in Sigrid for benchmarking, analyzes 70,000 enterprise systems on a weekly basis

Algorithm with Inefficient Time Complexity found where List.contains() is called from an iteration construct

Done

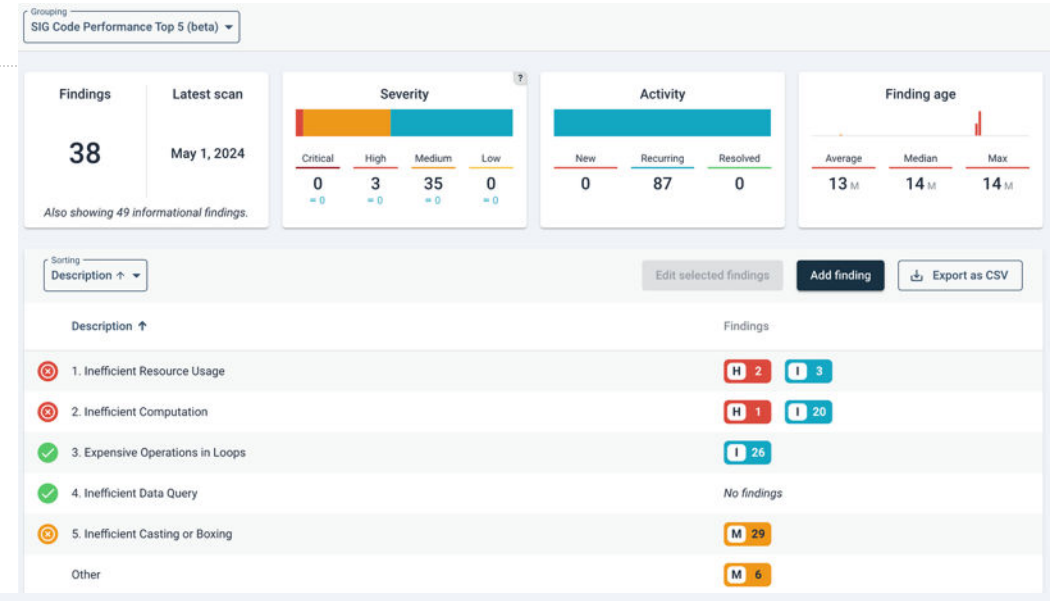
Problem Description

Calling `List.contains()` from an iteration construct such as a for loop, forEach loop, or while loop can lead to inefficient time complexity, especially for large lists. This can result in poor performance and excessive resource consumption. To address this problem, software engineers typically use a more efficient data structure such as `Set<>` or `Map<>` for look-up operations, which offer better time complexity for such operations.

Problem Example

```
1 import java.util.List;
2
3 public class ListContainsExample {
4     public boolean isValuePresent(List<String> list, String value) {
5         for (String item : list) {
6             if (list.contains(value)) {
7                 return true;
8             }
9         }
10        return false;
11    }
12 }
```

In this example, the `isValuePresent` method iterates through the list using a for-each loop and calls `list.contains(value)` within the loop. This can lead to inefficient time complexity as the `contains` method has to iterate through the list for each call, resulting in poor performance.



hadoop-hdfs-project/hadoop-hdfs/src/main/java/org/apache/hadoop/hdfs/server/balancer/Dispatcher.java

```
546 public void adjustIndices(List<Integer> adjustList) {
547     if (adjustList.isEmpty()) {
548         return;
549     }
550
551     byte[] newIndices = new byte[indices.length - adjustList.size()];
552     for (int i = 0, j = 0; i < indices.length; ++i) {
553         if (!adjustList.contains(i)) {
554             newIndices[j] = indices[i];
555             ++j;
556         }
557     }
558     this.indices = newIndices;
559 }
560 }
```

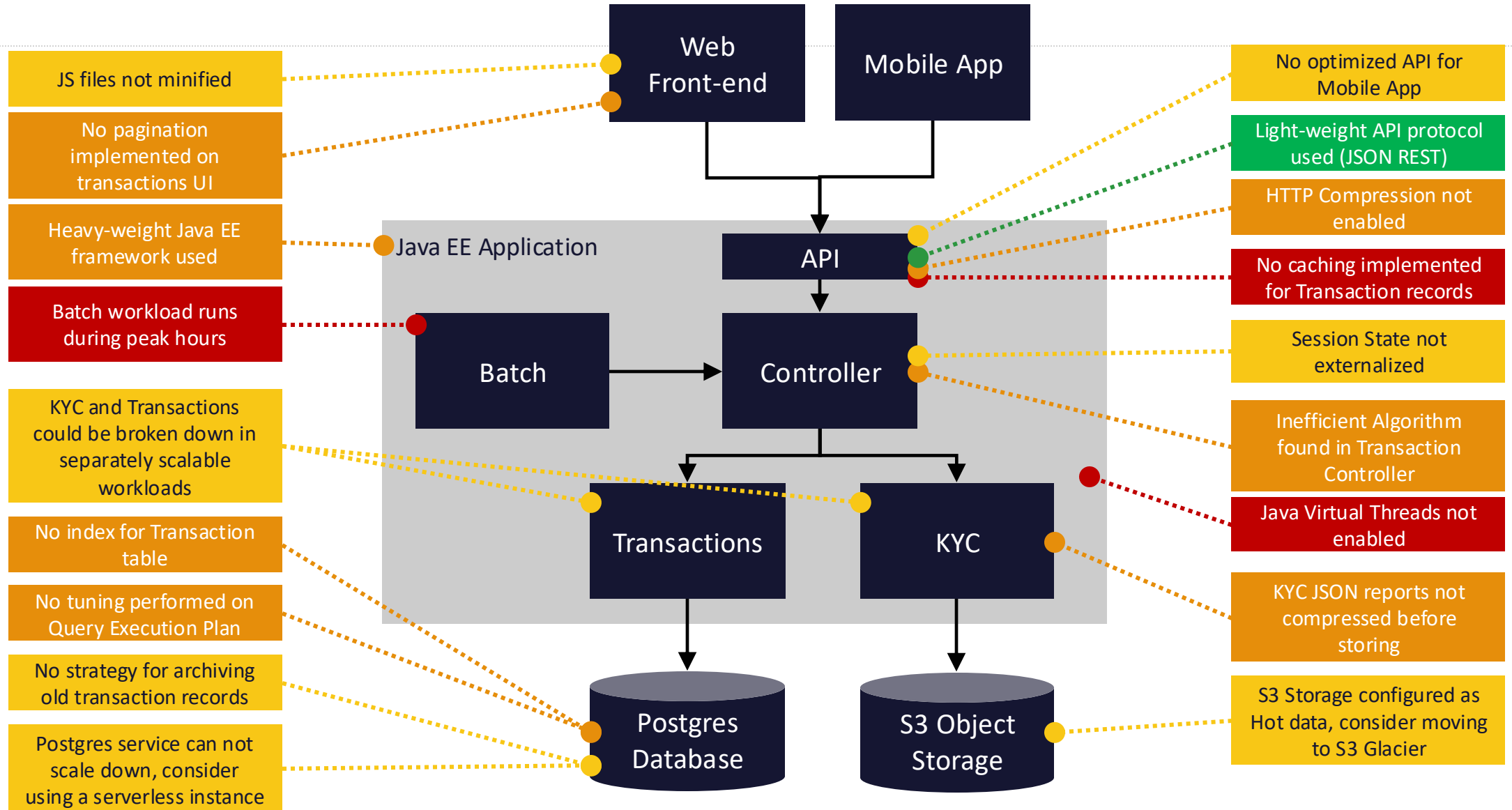
The SIG Green Software Model maps 3 sub-characteristics of ISO 21031 to 8 system properties

 ISO/IEC 21031 Software Carbon Intensity								
	 Code efficiency	 Technology efficiency	 Location awareness	 Resource elasticity	 Hosting strategy	 Architecture efficiency	 Embodied carbon strategy	 Observability
 Energy Efficiency	■	■		■		■		■
 Hardware Efficiency	■			■	■		■	■
 Carbon Awareness			■	■		■		■

Overview of Green Software Patterns and Practices for Enterprise Web Applications

Code	Architecture	Network	Deployment
Choose an energy-efficient language	Nonfunctional requirements for Sustainability	Apply caching	Shift to cloud (away from dedicated hardware)
Use multi-threading(async/parallel execution)	Define Green Architecture Blueprints	Enable (HTTP) compression	Use energy-efficient / high-density servers
Refactor inefficient algorithms (big O n ² to n)	Revisit Availability requirements (5 nines?)	Reduce payload size	Use energy-efficient chipset (e.g. ARM)
Use appropriate data structures	Simplify topology / Reduce layers	Avoid chattiness between services	Right-sizing (reduce over-provisioning)
Remove unused libraries/features/dead code	Break down large workloads into smaller parts	Use light-weight protocol (e.g. REST)	Migrate idle machines to on-demand
Use a light-weight framework	Stateless / Interruptible / Resumable	Avoid polling	Apply Power Scheduling (e.g. business hours)
Use homogenous tech stack (not fragmented)	Favor Multi tenancy	Request Bundling	Use workload orchestration (e.g. Kubernetes)
Use recursion / nested loops with care	Optimize API contract for Consumers (BFF)	Deploy services in Proximity (e.g. Pod Affinity)	Elastic scaling (Enable auto-scaling)
Avoid memory leaks	Externalize session state	Observability	Use cloud-native capabilities (e.g. Serverless)
Minimize data transformations	Replace self-managed components with SaaS	Use profiling to find inefficiency hotspots	Use Spare Cloud Capacity (e.g. Spot Instances)
Lazy loading of in-memory objects	Data	Monitor Memory/CPU/Energy Consumption	Use redundancy sparsely (use SRE patterns)
UX / Front-End	Index database tables	Set and track KPI for yearly CO ₂ emission	Use Service Mesh for High Load use cases
Dark Mode UI	Optimize and tune queries	Life Cycle Management	Carbon awareness
Pagination	Purge / archive old data	Ensure API backwards compatibility	Deploy in region with carbon free electricity
Minify digital resources (js/css)	Compress data at rest	Modernize software to extend its lifespan	Run jobs during Green time windows
Limit server roundtrips per user journey	Use appropriate data storage tiers (hot/cold)	Manage longevity of hardware (±5 years)	Schedule based on Carbon Intensity Forecasts

Sustainability By Design Analysis Example



How do we get started?

Activity	Operations	Development	Architecture	Leadership
Set direction and KPIs (Reduce carbon emission by x% by 2030)				
Embed <i>Sustainability by Design</i> principles in architecture blue prints for all Green Field development				
Make inventory and report on carbon emissions of infrastructure per initiative (Start simple with back of the envelope calculations)				
Assess brown field initiatives with the largest footprint				
Awareness and Knowledge sharing of Green Software & Green Hosting tactics				
Monitor energy consumption and spot idle resources (Proactively work with business to minimize footprint)				

Green software is future-proof software



 Pepijn van de Kamp

**Low emission software is fast
and scalable**

**>90% emission reduction means
significantly lower cloud costs**

Green Software entails so much more than just lowering carbon emissions

DANK

Nationale Coalitie Duurzame Digitalisering
Software Improvement Group

