



GreenTech Maturity Assessment

Analysis Report

Assessment Details

Date of Assessment	27 December 2024
Project Name	Platform Engineering
Company Name	Tata Consultancy Services (TCS)
Version	1.0
Cloud Provider	AWS

Current Maturity Level



Processes are standardized, documented, well understood and reviewed.

GreenTech Maturity Levels

Leve l 1	1	Processes are unpredictable, poorly controlled, and reactive at best.
Level 2	2	Per-project processes. Often still reactive.
Level 3	3	Processes are standardized, documented, well understood and reviewed.

Level 4	4	Processes are measured and controlled.
Level 5	5	Continuous improvement occurs based on quantitative feedback.

Areas Implemented

Development	4 of 24
Network	1 of 2
Deployment	7 of 19
Design	0 of 31
Storage	0 of 5
Quality	0 of 8
Operations	0 of 12

Recommendations

Development:

1. Optimize source code for energy and carbon emissions using static code analysis

Avoid usage of static collections.	87
Avoid multiple if-else statement	61
Do not call a function when declaring a for-type loop	54
Use ++i instead of i++	43
String Builder	29
Avoid getting the size of the collection in the loop	23
Avoid creating and starting threads directly	17
Avoid using Pattern.compile() in a non-static context	11
Free Resources	2

Use System. arraycopy to copy arrays	1
--------------------------------------	---

2. Memory and energy utilization of docker and multi-stage docker files

Energy Usage	High	Low
Build Time	5 minutes	6 minutes
Layer Count	12 layers	8 layers
Image Size	2.06 GB	2.01 GB
Cache Efficiency	Low	High
CI/CD Impact	High resource usage	Optimized resource usage

Adopt Multi-Stage Docker Files:

- Reduce memory and energy consumption during build and runtime.
- Smaller image sizes lead to faster deployments and lower storage needs.

Optimize Base Images:

- Use official, slim, or alpine versions of base images to minimize size.
- Periodically review and update base images to leverage newer, more efficient versions.

Minimize Dependency Installation:

- Only install necessary dependencies to reduce build and runtime resource usage.
- Use tools like pip-compile to manage dependencies efficiently.

Leverage Caching:

- Utilize Docker's build cache to skip unchanged layers during rebuilds.
- Implement caching for dependencies (e.g., using a proxy server for pip dependencies).

Monitor and Analyze Resource Usage:

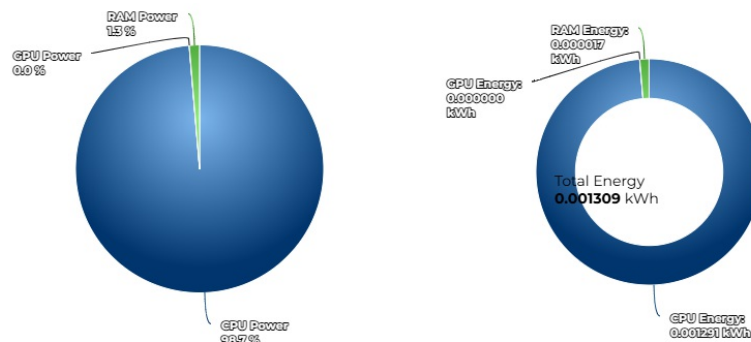
- Employ tools (e.g., Docker Stats, Prometheus, Grafana) to monitor container resource utilization.
- Analyze findings to identify optimization opportunities.

3. Energy metrics for application using Intel RAPL, Kepler, Schaphandre

CPU Model	Intel(R) Xeon(R) Platinum 8259CL CPU @ 2.50GHz

CPU Count	2
GPU Model	N/A
GPU Count	N/A
RAM	3.74 GB
OS	Linux
Country	United States
Region	Virginia

44.28 Seconds	0.001309 Kwh	0.000483 kg	0.00001091 kg/s



Monitor and Optimize: Regularly collect energy metrics to identify optimization opportunities in your application.

Set Sustainability Goals: Establish targets for your Sustainability Score (SS) or individual energy metrics to drive improvement.

Energy-Aware Scheduling: Utilize Kepler to schedule workflow tasks during periods of low energy demand or when renewable energy sources are available.

Hardware/Software Co-Optimization: Collaborate with hardware teams to optimize system configurations for improved energy efficiency.

Continuously Update and Refine: As new energy-efficient technologies and methodologies emerge, incorporate them into your application and workflows.

4. Energy consumption of ML job, Model training

Optimize Model Architecture:

- Use efficient neural network architectures (e.g., MobileNet, SqueezeNet, EfficientNet) that require less computational resources.
- Apply model pruning, knowledge distillation, or quantization to reduce model size and computational requirements.

Select Energy-Efficient Hardware:

- Utilize GPU accelerators with high performance-per-watt ratios (e.g., NVIDIA Ampere or AMD CDNA).
- Consider TPU (Tensor Processing Unit)-based solutions for large-scale ML workloads.

- Explore FPGA (Field-Programmable Gate Array)-based accelerators for customized, energy-efficient computations.

Efficient Training Methodologies:

- Employ transfer learning to fine-tune pre-trained models, reducing training time and energy.
- Use early stopping techniques to halt training when satisfactory performance is achieved.
- Apply distributed training with optimized parallelization to minimize overall training time.

Data Efficiency:

- Use smaller, representative datasets for training, reducing computational requirements.
- Apply data augmentation to artificially increase dataset size without adding new samples.

Hyperparameter Tuning and Automation:

- Utilize hyperparameter tuning tools (e.g., Hyperopt, Optuna) to quickly identify optimal configurations.
- Implement automated ML (AutoML) pipelines to streamline the training process and minimize unnecessary computations.

Network:

1. **Energy efficiency for synchronous (REST vs gRPC) and Asynchronous (Kafka vs RabbitMQ) messaging between microservices**

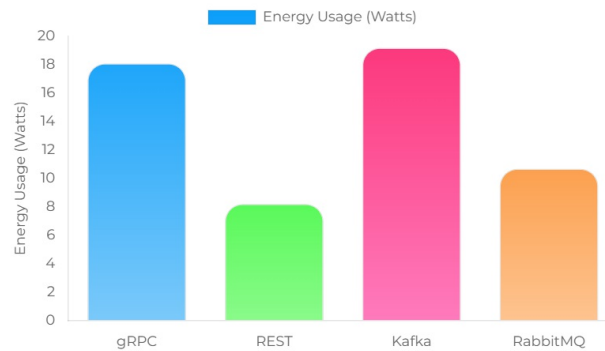
Synchronous Communication Protocols

Energy Usage	18.1 W	8.2 W
Peak Memory	4.45 MB	8 MB
Time Taken	3 Minutes	4 Min 50 Sec
Requests	500K	500K
Peak CPU	10%	4%

Asynchronous Communication Protocols

Energy Usage	19.2 W	10.68 W
Peak Memory	1024 MB	512 MB
	1M msg/s	50K msg/s

Message Rate		
Latency	10ms	5ms
CPU Usage	55%	35%



For Synchronous Messaging (REST/gRPC)

Use gRPC for New Developments:

- Leverage its inherent efficiency advantages, especially for microservices with high inter-service communication.

Migrate REST to HTTP/2:

- If moving to gRPC isn't feasible, ensure REST services use HTTP/2 for some efficiency gains.

Optimize Payloads:

- Use efficient serialization formats (e.g., protobuf for gRPC, consider alternatives for REST).
- Implement compression (if not already done).

Service Discovery and Load Balancing:

- Ensure efficient routing to reduce unnecessary network hops.

For Asynchronous Messaging (Kafka/RabbitMQ)

Choose Kafka for High-Volume Scenarios:

- Prefer Kafka when dealing with high throughput and low-latency requirements.

Optimize RabbitMQ Configurations:

- For existing RabbitMQ setups, review and optimize configurations for batching, compression, and efficient queue management.

Leverage Distributed Capabilities:

- For both Kafka and RabbitMQ, ensure distributed setups are optimized to minimize energy consumption.

Deployment:

1. Tagging of resources to track usage

Mandatory Tags:

- Owner (e.g., team, department, individual)
- Environment (e.g., dev, staging, prod)
- Project/Service (e.g., project name, service identifier)
- Cost Center (e.g., budget code, department ID)

Optional Tags (as needed):

- Application
- Component
- Lifecycle (e.g., temporary, permanent)
- Compliance (e.g., PCI, HIPAA)

Tagging Best Practices:

- Use Meaningful Names: Clearly indicate the tag's purpose.
- Keep it Concise: Short tag names (< 20 characters) for easier management.
- Avoid Duplication: Use a single tag for a specific attribute (e.g., don't use both Env and Environment).
- Use Consistent Formatting: Establish a standard for tag values (e.g., all lowercase, separated by hyphens).

Tagging Structure (Hierarchy):

- Flat Structure: Simple, straightforward (e.g., Owner: JohnDoe, Environment: Prod)
- Hierarchical Structure: Organized with categories (e.g., Project:MyApp/Environment:Dev, CostCenter:IT/Department:DevOps)

Instances without Tags:

marvel-windows-jumbotest	i-07a7a6794367ded49	t3.medium
marvel-esha-windows-jumpbox-01	i-0378700bcf2ef68e7	t3.large
marvel-rapl-amd	i-0c43a18a046c87984	t3a.large
marvel-sustain	i-0571933e06349edd6	t2.medium
marvel-autoscaler-3	i-0c86e5cbdc6ef48e9	t2.medium
marvel-docker-registry	i-0121edc6357368099	t3.micro
marvel-git-perforce	i-0361799692abc3dc4	t2.medium
marvel-esha-windows-jumpbox-03	i-01b006bb7ceefd989	t3.large
marvel-Prometheus	i-00f4b18e0c7615802	t2.micro

marvel-slurm-testing-slurmctl	i-02028574274244586	t3a.small
marvel-slurm-testing-slurm-00	i-0c10cab3a673568a5	t3a.small
marvel-slurm-testing-gw	i-01528621d1601b22d	t3a.small
marvel-slurm-testing-slurmdb	i-04cf6faf1252ba6a7	t3a.small

2. Calculate energy utilization for provisioned resources and show recommendations to deployment teams

(Legend: R denotes Recommendation Rank)

Under-Provisioned:

i-0011eee5f9bfa37d5	t3.large	m7i-flex.large	m6i.large	m7i.large
i-01e2bba24c61139a5	t3.large	m7i-flex.large	m6i.large	m7i.large
i-0378700bcf2ef68e7	t3.large	m7i-flex.large	m6i.large	m7i.large
i-061783c19995cf461	t3.large	m7i-flex.large	m6i.large	m7i.large
i-06b0d1885c8bfb73a	t2.medium	c7i-flex.large	c6i.large	c5.large
i-07d971daf49d26ba6	t3.large	m7i-flex.large	m6i.large	m7i.large
i-07fc1239a8432b1ff	t2.nano	t3.small	t3.medium	t3.large
i-0803db063f7902be8	t2.nano	t3.small	t3.medium	t3.large
i-0a30bd8f427568294	t2.large	m7i.large	r7i.large	c7i-flex.xlarge
i-0bcac5f25c482c094	t3.large	m7i-flex.large	m6i.large	m7i.large
i-0e7f58282fba77a57	t3.xlarge	m7i-flex.xlarge	m6i.xlarge	m7i.xlarge

Over-Provisioned:

i-07a7a6794367ded49	t3.xlarge	r7i.large	t3.xlarge	-
i-0f6c16ceb8ea482c2	t3.2xlarge	r6i.xlarge	r7i.xlarge	t3.2xlarge
i-0f97772fef03977df	t3.xlarge	r6i.large	r7i.large	t3.xlarge

i-0fda6069f43c06df1	t3.xlarge	r7i.large	t3.xlarge	-
---------------------	-----------	-----------	-----------	---

3. Identify Unused Resources

Unused EC2 Resources:

marvel-GenAI-partha	i-0a1748bca3199a63f	t3.medium	274
marvel-LLM-test	i-0b43ba541b65fcd8f	t3.2xlarge	175
marvel-mddp-devsecops	i-0c73c29f8e5d01576	t3.xlarge	162
marvel-EEaaS-Kubernetes	i-0ced2e371b3425fe5	t3.large	142
marvel-TEaaS-openstack-kolla	i-0842262e5bb3e2915	t3.xlarge	141
marvel-awx-ansible	i-0d5da5ba883efcf5b	t3.xlarge	127
marvel-Neureda-registry	i-0dc340b462af4ed74	t3.medium	125
marvel-windows-jumboxtest	i-07a7a6794367ded49	t3.medium	122
marvel-PlatformEng-Keycloak	i-0161cf03311390911	t3.medium	121
marvel-PE-mddp-k8s-1	i-0bfe201057f0db5f9	t3.2xlarge	121
marvel-Prometheus	i-00f4b18e0c7615802	t2.micro	120
marvel-slurm-testing-slurmctl	i-02028574274244586	t3a.small	120
marvel-slurm-testing-slurm-00	i-0c10cab3a673568a5	t3a.small	120
marvel-slurm-testing-gw	i-01528621d1601b22d	t3a.small	120
marvel-slurm-testing-slurmdb	i-04cf6faf1252ba6a7	t3a.small	120
mfdm_qual_dd_pg	i-03b6cc42f6ee8679a	t2.medium	116
tcsnxgnmfdmins5	i-054d01dc2082a9968	t3.large	114
tcsnxgnpemonobs01	i-0a30bd8f427568294	t2.large	114
tcsnxgnpemonobs02	i-0644a1225a61cfad0	t3.medium	114
tcsnxgnpemonobs03	i-0f4a538834c7f8379	t2.large	114
tcsnxgnmfdmins1	i-06f35320739d13110	t3.medium	113
tcsnxgnmfdmins3	i-0dd30488163388001	t3.medium	113

tcsnxgnmfdmins2	i-030fdebbe195c6d05	t3.medium	113
-----------------	---------------------	-----------	-----

4. Turn off workloads and node pools outside of business hours

Automate Shutdown/Startup

- What It Is: Scripted automatic shutdown/startup of workloads outside business hours.

Sustainability Benefits:

- Reduced energy consumption
- Lower greenhouse gas emissions
- Decreased e-waste from prolonged hardware lifespan

Identify Non-Essential Workloads

- What It Is: Analyzing workloads to determine which can be safely turned off during non-business hours.

Sustainability Benefits:

- Targeted energy reduction
- Minimized unnecessary resource utilization
- Enhanced overall efficiency

Autoscaling Configuration (Node Pools)

- What It Is: Configuring node pools to autoscale down to 0 nodes during non-business hours.

Sustainability Benefits:

- Dynamic energy consumption adjustment
- Significant reduction in idle resource energy waste
- Enhanced environmental responsiveness

Scheduled Node Pool Management

- What It Is: Scheduling node pool shutdowns/startups using platform-specific features or tools.

Sustainability Benefits:

- Predictable energy savings
- Reduced operational carbon emissions
- Improved resource utilization efficiency

Node Pool Sizing Optimization

- What It Is: Regular review to optimize node pool sizing.

Sustainability Benefits :

- Continuous energy efficiency improvement

- Reduced e-waste through minimized node replacements
- Lower environmental impact

5. Show energy efficient resource provisioning options during provisioning

Please refer to this [link](#) for a guide on how to implement this.

6. Implement pre-checks for common issues in CI/CD pipeline to avoid failures in different stages of pipeline

Please refer to this [link](#) for a guide on how to implement this.

Details

1. Development (4/24)

Areas Implemented:

- Energy metrics for application using Intel RAPL, Kepler, Schaphandre
- Energy consumption of ML job, Model training
- Optimize source code for energy and carbon emissions using static code analysis
- Compare memory and energy utilization of docker and multi-stage docker files

Areas to be Implemented:

- Simplicity and Efficiency Trade-offs while developing algorithms
- Efficient Software Algorithms and data structures
- Efficient integration and delivery pipelines (Dev, Test, CI env)
- On demand development environments
- Minimizing unnecessary code execution
- Utilizing lazy loading
- Optimize energy consumption for Python and Java applications
- Leverage LLMs to suggest energy efficient algorithms for Java / Python applications in IDE
- Provide configurable deployment of applications with selected features to minimize resource utilization
- Calculate energy utilization at function level
- Optimize energy utilization of applications using AI/ML
- Memory optimization for applications
- Rightsizing containers and VM's based on resource utilization
- Compare and suggest open-source tools which are energy efficient

- Train models with custom rules for optimization of code
- Suggest optimized version of built-in functions which are energy efficient in IDE / static analyzer
- Code splitting - Convert code to smaller files which can be loaded on-demand
- Tree shaking: This technique helps reduce the overall application size by removing unused code from the final build, thus optimizing the web product and minimizing resource consumption
- Split large code repositories
- Remove dead code and unused data from application

2. Network (1/2)

Areas Implemented:

- Compare energy efficiency for synchronous (REST vs gRPC) and Asynchronous (Kafka vs RabbitMQ) messaging between microservices

Areas to be Implemented:

- Optimize data transfer (Implement local data processing, aggregation and compression techniques to avoid transfer of frequent, large amounts data)

3. Deployment (7/19)

Areas Implemented:

- Tagging of resources to track usage
- Calculate energy utilization for provisioned resources and show recommendations to deployment teams
- Identify and share report on unused resources
- Turn off workloads and node pools outside of business hours
- Show energy efficient resource provisioning options during provisioning
- Implement pre-checks for common issues in CI/CD pipeline to avoid failures in different stages of pipeline
- Workload allocation on energy efficient servers, VMs

Areas to be Implemented:

- Deploy application in a region that is closest to users
- Deploy in regions that are powered by renewables or energy efficient resources
- Energy efficient workload scheduling
- CPU frequency optimization for non-critical workloads
- Alerts during resource provisioning (to avoid over provisioning) about energy consumption and energy efficient options

- Calculate and minimize energy utilization per transaction
- Implement automated decision between horizontal vs vertical scaling of applications
- Set up minimal artifacts - Reduce disk, memory and processing demands by considering the software for a given purpose
- Schedule resource intensive tasks to execute when renewable energy source is available
- Consolidate application workloads to maximize server utilization and move idle servers to low power mode
- Measure (Idle/Normal/Peak) energy consumption of resources and identify scope for optimizations (Get power consumption per CPU core information from processor type)
- Identify unused resources in K8s using KOR tool

4. Design (0/31)

Areas Implemented:

- No Areas Implemented

Areas to be Implemented:

- Sustainability as NFR
- Minimal Architecture (i.e. Developing minimal software for value)
- Replace long-running service with a simple Function as a Service
- Asynchronous processing and event-driven architectures
- API Design: protocol (TCP, UDP, custom wire protocols)
- API Design: data format used for request/response of your APIs: Protobuf/gRPC is better compared to JSON, XML
- API Design: Avro schemas instead of JSON for data serialization
- Reusable APIs vs P2P Integrations
- Choice of programming language: Go, Rust over Python, Java; Java 17 is better with respect to memory optimization
- Public cloud, choice of provider, and region
- Platform as a Service (PaaS) and Serverless (e.g., Function as a Service (FaaS))
- Containers and Kubernetes - Adopting energy efficient architectures such as containers or serverless
- Scheduling and batch vs. real-time - Demand Shaping
- Carbon awareness into the Kubernetes Scheduler
- Cost and Sustainability: Reducing operational costs and sustainability are aligned
- Enforce Quotas and Rate Limiting
- Reduce the Network Footprint
- Identify energy utilization of applications across different design choices
- Optimize data generation of applications
- Create reusable modules to minimize development and maintenance efforts

- Enable parallel processing using distributed computing and minimize energy consumption
- Measure KPIs on the features used by customers and remove unused features
- Energy saving mode for applications, system processes during periods of low load
- Energy budgets for applications
- Create Energy profile for application (Core Vs Non-core tasks - Logging, Monitoring, Audit, Redundancy etc...)
- Improve resilience of equipment to minimize redundancy (Ensure legacy hardware can support at least core features of software)
- Shared libraries for applications
- Server-side rendering (SSR): With SSR, a web page's HTML is generated on the server and sent to the client, resulting in quicker initial load time and improved search engine optimization
- React frameworks: Next.js as a framework has many built-in features for energy optimization
- Progressive Web Applications (Hybrid of native and web apps) are more energy efficient
- Optimize availability and scalability needs for application

5. **Storage (0/5)**

Areas Implemented:

- No Areas Implemented

Areas to be Implemented:

- Dynamic provisioning of volumes for EC2 instance / Pod to automatically scale the storage size
- Identify and remove unused data and resources in cloud
- Implementing data retention, archival and deletion policies
- Utilize caching to store frequently used data
- Optimize log and metrics collection

6. **Quality (0/8)**

Areas Implemented:

- No Areas Implemented

Areas to be Implemented:

- Code and design review for sustainability
- Document decisions using Key Design Decisions (KDDs) or Architectural Decision Records (ADRs)
- Profiling tools and static analysis - Code profiling and optimization tools
- Upgrade Runtimes and Modules
- Change aware testing
- Test Case Optimization
- Risk Based Testing
- On-demand and scalable Test Environments

7. Operations (0/12)

Areas Implemented:

- No Areas Implemented

Areas to be Implemented:

- Automated sustainability maturity assessment for applications
- Observability and CarbonOps
- Automation
- Switching instances to chipsets that offer the same processing power at lower levels of energy consumption
- Sustainable DevOps practices integrate sustainability into the software delivery process
- Optimize resource allocation using ML
- Maximize utilization of resources
- Spot Instances
- Setup sustainability goals for infrastructure, development and applications and monitor them using dashboards and reports
- Reduce the cooling requirements of servers by ensuring that applications (servers) run in optimal temperature range
- Identify zombie workloads that consume resources (e.g., Application not running within container / VM)
- Dashboard for resource utilization at cluster, node, and pod level and grouped by applications in Kubernetes