Mastercard Catalogue of Patterns and Practices

Structure Overview

The catalogue is currently structured in 5 categories, covering a broad spectrum of roles in the Software Engineering community:

Architectural Patterns
Patterns that can be applied when designing and architecting your application to ensure that it is the most efficient, and reduces waste that result in CO_2 emissions.

Technology Stack & Tooling Patterns
Patterns related to technology stacks, what choices to make when you are starting off a project, and what trade-offs you need to consider.

Code and Algorithm Patterns
Patterns that can be applied when writing code for your application to ensure it is as efficient as possible. Remember that the code that is not written is the greenest of all!

Methodology Patterns
Patterns that are related to Software Development methodology, which have a direct impact on Green Software principles. For example Agile Practices encourage an iterative and incremental approach to building software, ensuring that the features that are built first are the ones most valuable to customers, and not more, therefore reducing wastes.

Operational Patterns
Patterns that are related to the running software in Production, including management of the hardware and equipment required to execute systems.

We are currently working with the different internal groups to expand the Catalogue to cover areas such as:

- Persistence and Storage
- Big Data
- AI

Examples of Patterns

Make Docker images as lean as possible

Description

Docker images can grow very large in size (up to gigabytes), consuming a lot of resources when stored in repositories for distribution, and copied across the network.

Solution

- Make sure to only include the required dependencies, and delete dependencies that are not required, or used only to build the image itself. This can be done in particular using multi-stage builds.
- Use base images that have been stripped to the bare minimum, such as slim images or distroless images.
- Compress (flatten) layers.

SCI Impact

SCI = (E * I) + M per R Software Carbon Intensity Spec

- E: We reduce the total electricity required to transfer large images on the network, and writing and reading to/from disk.
- M: We reduce embodied emissions by reducing the need for extra storage devices to store images.

Assumptions

N/A

Considerations

This has a positive impact on security, as this reduces the surface attack of a given container.

References

- Best practices for writing Dockerfiles
- Image Building Best Practices
- Multi-Stage Builds

Correctly apply throttling/rate limiting

Description

A system under intense pressure, which is not able to respond to an increased number of incoming requests, is uselessly executing with overloaded memory and CPU

Solution

Apply throttling/rate limiting to ensure your application does not get overloaded, and executes with optimal CPU and memory usage. Requests can be queued for deferred processing where possible, also allowing energy awareness optimizations.

SCI Impact

`SCI = (E * I) + M per R` Software Carbon Intensity Spec

Using throttling/rate limiting will:

- E: We reduce the total electricity required by ensuring that only the resources needed to serve the current load are used, and not more.

Assumptions

N/A

Considerations

- It may reduce cloud costs by ensuring that only the resources needed are used
- It helps protecting against malicious attacks, for example denial of service (DoS)

References

N/A

Leverage parallel processing to maximize a server's processing capacity

Description

Servers may be under-utilized for the need of the application, only using a small number of the CPUs at its disposal by processing requests sequentially.

Solution

Use parallel processing to optimize processing of incoming requests by ensuring optimal utilization of CPU and memory.

SCI Impact

`SCI = (E * I) + M per R` Software Carbon Intensity Spec

Using parallel processing will:

- E: We reduce the total electricity required by ensuring that only the resources needed to serve the current load are used, and not more.

Assumptions

Workload can be parallelized. This may be more difficult in the case of stateful applications.

Considerations

It may reduce cloud costs by ensuring that only the resources needed are used.

References

N/A

Tune your Java Garbage collection for optimal memory usage

Description

When the memory of a long-running JVM program gets filled, without old objects being re-claimed, a large amount of resources can be spent by the JVM trying to free memory.

Solution

Learn and understand Garbage collection strategies, and ensure Garbage collection is appropriately fine-tuned for your application to ensure optimal memory usage, and stop-the-world GC. Ensure objects are created with minimal scope (for example local method variables rather than instance variables).

SCI Impact

`SCI = (E * I) + M per R` Software Carbon Intensity Spec

By tuning Garbage collection:
- E: We reduce the energy spent by the process needlessly thrashing by ensuring the garbage collector performs its task most efficiently.

Assumptions

Metrics are collected as part of the operations of your applications, and indicate issues in memory utilization where, for example, there are recurring stop-the-world GCs, affecting performance.

Considerations

Garbage collection in Java 11 uses G1GC, which behaves differently to the older Concurrent Mark Sweep garbage collector

References

Available Collectors

Strive for Simplicity

Description

Complexity in a system makes a few things more difficult:

- Understanding of the system and its dependencies
- Maintenance of that system, and extending it to implement new features,
- Troubleshooting and resolving availability issues,
- Making the system highly available and resilient.

All this as a consequence causes additional energy consumption and CO_2 which could be avoided.

Solution

Strive for simplicity, and always consider whether there are simpler approaches to the ones you are considering. Consider multiple alternatives for your architecture. Practice moderation and avoid over-engineering. Measure code complexity using appropriate tools.

"Simplicity is prerequisite for reliability"
—Edsger W. Dijkstra (1975)

SCI Impact

`SCI = (E * I) + M per R` Software Carbon Intensity Spec

In striving for simplicity:

- E: We reduce the total energy consumed by the application by reducing caused by maintenance, lack of efficiency, unnecessary complex computations

Assumptions

N/A

Considerations

Simplicity improves the maintainability and understandability of a system, therefore reducing costs of maintenance and development of new features.

References

N/A

You're Not Gonna Need it! (YAGNI)

Description

When developing features on an application, it can be tempting to develop additional elements or features, in the perspective of extending the system in the future, or supporting additional integration points. This practice not only introduces the risk of having to maintain a part of the system that will not be used, but that may also need to be re-written in the future because assumptions and requirements have changed.

Solution

Only implement things when you actually need them, not in prevision of future requirements or applications. Refactor early and frequently.

SCI Impact

`SCI = (E * I) + M per R` Software Carbon Intensity Spec

By adopting YAGNI:

- E: We reduce the total electricity consumed by needlessly writing and maintaining the code

Assumptions

N/A

Considerations

N/A

References

- YAGNI by Martin Fowler