TIREx Tracker: The Information Retrieval Experiment Tracker

Tim Hagen
University of Kassel and hessian.AI
Germany, Kassel

Maik Fröbe
Friedrich-Schiller-Universität Jena
Germany, Jena

Jan Heinrich Merker
Friedrich-Schiller-Universität Jena
Germany, Jena

Harrisen Scells
University of Kassel and hessian.AI
Germany, Kassel

Matthias Hagen
Friedrich-Schiller-Universität Jena
Germany, Jena

Martin Potthast
University of Kassel,
hessian.AI, and ScaDS.AI
Germany, Kassel

Abstract

The reproducibility and transparency of retrieval experiments depends on the availability of information about the experimental setup. However, the manual collection of experiment metadata can be tedious, error-prone, and inconsistent, which calls for an automated systematic collection. Expanding `ir_metadata`, we present the TIREx tracker, a tool that records hardware configurations, power/CPU/RAM/GPU usage, and experiment/system versions. Implemented as a lightweight platform-independent C binary, the TIREx tracker integrates seamlessly into Python, Java, or C/C++ workflows and can be easily integrated into shard task submissions, as we demonstrate for the TIRA/TIREx platform. Code, binaries, and documentation of the TIREx tracker are publicly available at https://github.com/tira-io/tirex-tracker.

CCS Concepts

• **General and reference → Experimentation**; • **Information systems → Information retrieval**; *Retrieval efficiency*.

Keywords

IR metadata; Reproducibility; Information retrieval evaluation

ACM Reference Format:
Tim Hagen, Maik Fröbe, Jan Heinrich Merker, Harrisen Scells, Matthias Hagen, and Martin Potthast. 2025. TIREx Tracker: The Information Retrieval Experiment Tracker. In *Proceedings of the 48th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '25), July 13–18, 2025, Padua, Italy.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3726302.3730297

1 Introduction

In information retrieval (IR), comparative experimental evaluation has a long tradition fostered by dedicated conferences like TREC, CLEF, NTCIR, or FIRE. In the last years, with an increasing focus on reproducibility and transparency of experiments, data and code sharing are more and more encouraged (also through peer review), and many IR conferences now have resource and reproducibility tracks. Initiatives such as artifact review and badging [7], model cards [14], or `ir_metadata` [3] further promote openness, and the ACL has recently included resource and environmental impact reporting in their checklist for responsible NLP,[1] as larger and larger transformer-based models and computationally demanding approaches are employed. In this respect, tracking and comparing resource consumption alongside effectiveness has also become an important aspect for IR experiments.

However, tracking the use of computational resources is still tedious (e.g., tools such as `ir_metadata` [3] require manual work) and approaches to automate the collection of metadata for IR experiments (setup, conditions, etc.) are so far limited to Python [2]. To address these issues and to further improve reproducibility and transparency of IR experiments, we develop the TIREx tracker,[2] a lightweight and easy-to-integrate tool for automatically capturing experiment metadata and efficiency aspects. Our approach extends `ir_metadata` to include various reproducibility and efficiency aspects such as hardware specifications, energy consumption, and hardware utilization. By design, the TIREx tracker's API is kept as simple as possible to enable a wide usage without placing additional demands on researchers. In a case study, we demonstrate the tracker's versatility through seamless integration with TIREx [8].

2 Related Work

Reproducibility in IR. Reproducibility in information retrieval (IR) is an ongoing challenge [19]. Recent efforts have focused on metadata collection and dockerization since metadata plays an important role to improve replicability [11]. Breuer et al. [3] propose the `ir_metadata` specification for IR experiments, which is based on the PRIMAD model [6], which identifies six major components of an experiment: **P**latform, **R**esearch goal, **I**mplementation, **M**ethod, **A**ctor, and **D**ata. As such, `ir_metadata` includes, among others, metadata about the operating system, hardware specifications, dependencies, the researcher themselves, and their research goal. It also specifies limited Git repository metadata (remote URL and commit hash). However, richer Git metadata, such as whether the commit is up-to-date, whether untracked files exist, or whether tracked files contain uncommitted changes, would further improve reproducibility.

To our knowledge, `repro_eval` [2], is the only tool that automatically generates `ir_metadata`, but its scope is limited to Python-based experiments. Beyond metadata collection, containerization has been explored as a means to ensure reproducibility. The Open-Source IR Replicability Challenge [5] proposed a Docker specification for

SIGIR '25, Padua, Italy

ACM ISBN 979-8-4007-1592-1/2025/07
https://doi.org/10.1145/3726302.3730297

---

[1] https://aclrollingreview.org/responsibleNLPresearch/
[2] https://github.com/tira-io/tirex-tracker

IR experiments, while TIREx [8] extends the Shared Task Platform TIRA [9] with IR-specific features. TIREx enforces replicability by executing Docker-submissions in a sandboxed environment, preventing external dependencies (e.g., internet access).

Despite the advantages of containerization for reproducibility, adoption remains limited—likely due to the overhead of defining and managing Docker containers. Our approach with the TIREx tracker seeks to address this by automatically collecting metadata needed to reconstruct a Docker image after an experiment is run. This reduces the manual effort required from researchers but gives detailed insight into the environment the experiments were run in, improving reactive reproducibility actions [16].

When used with TIREx, the TIREx tracker enables seamless evaluation on TIRA by automatically collecting metadata for the Platform, Implementation, Method, and Data. Researchers are then provided with a link to claim ownership of their submission, thereby incorporating the Actor component of PRIMAD. In cases where the experimental setup does not require network access or extensive software dependencies, this approach allows for easy reconstruction of the runtime environment using Docker.

Energy Tracking. The introduction of transformer-based models has significantly impacted information retrieval, primarily due to their superior language modeling capabilities. However, this improvement typically comes at the cost of efficiency (in terms of training or inference costs), as transformer models require substantially more computational resources. These additional resources lead to higher energy consumption per query. With the widespread adoption of such models, the environmental impact of information retrieval research has become a growing concern [18, 21]. This has led to initiatives such as the ReNeuIR workshop series—now in its fourth edition at SIGIR [4]—and, more broadly, the inclusion of environmental impact reporting in ACL's Responsible NLP Checklist.

Efficiency tracking in IR is not new, but modern models' increasing computational demands made energy tracking more critical than ever. Several frameworks were developed to address this. Research-focused solutions include Carbontracker (420 GitHub stars, actively maintained) [1] and experiment-impact-tracker (281 stars, last commit four years ago) [10]. More general-purpose tools, such as pyJoules[3] (79 stars) and CodeCarbon[4] (1.3k GitHub stars, actively maintained), aim to track energy consumption and emissions across diverse applications.

A key limitation of these tools is their language dependence. Most of them, including Carbontracker and CodeCarbon, are Python-specific. To address the lack of energy tracking solutions for C++, CPPJoules [17] was recently introduced as a C++ alternative inspired by pyJoules. However, this results in separate language-specific libraries, whereas a unified, language-agnostic solution would be preferable. A cleaner approach would be a central native C interface that enables seamless integration across different programming languages and adding language integration through thin wrappers.

Beyond language constraints, certain design choices in existing tools affect their accuracy and usability. Notably, CodeCarbon resorts to approximations when direct energy data is unavailable. Its methodology page states: "We could not find any good resource showing statistical relationships between TDP and average power, so we empirically tested that 50% is a decent approximation [of the power consumption]."[5] Such assumptions can lead to inaccurate energy estimates. Additionally, CodeCarbon's installation instructions[6] omit crucial dependencies, such as Intel Power Gadget on Windows or the necessary permissions for `/sys/class/powercap/intel-rapl` on Linux. Since it silently falls back to estimation rather than alerting users to missing configurations, users may unknowingly rely on incorrect measurements.[7]

Most existing tools rely on Intel Power Gadget to track energy consumption on Intel CPUs in Windows environments. However, Intel Power Gadget presents two key issues: (1) it requires a separate installation, complicating deployment, and (2) it was deprecated at the end of 2023, replaced by Intel PCM, which current tools do not yet support. CodeCarbon is actively working on integrating Intel PCM, but broader adoption remains an open challenge.

3 Implementation

To provide a flexible and lightweight mechanism for collecting runtime environment data and resource consumption metrics, we separate the TIREx tracker into a minimal native tracking layer and language-specific bindings (see Figure 1). The native library provides a minimal interface to track most metrics and is independent of both the operating system and the hardware architecture. The TIREx tracker's native library tracks system metrics (e.g., CPU and GPU utilization, energy consumption, and memory usage) and system metadata (e.g., the operating system's name and version). Additional language-specific bindings allow capturing language-specific metadata (e.g., installed packages) and facilitate easy integration with different programming environments of information retrieval experiments. Currently, bindings are provided for Python and JVM-based languages (e.g., Java, Kotlin), but since the TIREx tracker's C library is a self-contained shared library, bindings for other languages can easily be added.

In the following sections, we first describe which measures are tracked and how (Section 3.1), then we detail how the native, low-level, cross-platform C library is designed (Section 3.2), how the Python/Java language-specific, high-level wrappers are used (Section 3.3), and, lastly, how the results can be exported to `ir_metadata`-compatible files (Section 3.4). Figure 1 provides an overview of these components and interactions with third party APIs.

3.1 Measuring System Metrics and Metadata

Reliably measuring system metrics and collecting metadata across diverse computing environments presents several challenges (see also Section 2). To unify and simplify this tedious task, we have designed the TIREx tracker to support the most popular operating systems (Linux/Windows/macOS), CPU architectures (x86/ARM), and hardware vendors (Intel/AMD/Apple CPUs, and Nvidia/Apple GPUs). We expose a common interface that is agnostic to the underlying system, so we do not impose restrictions on the researcher's choice of hardware or software environment. Internally, the metrics and metadata are tracked by data providers that each specialize

---

[3] https://github.com/powerapi-ng/pyJoules

[4] https://github.com/mlco2/codecarbon

[5] https://mlco2.github.io/codecarbon/methodology.html

[6] https://mlco2.github.io/codecarbon/installation.html

[7] See for example issues #515 and #677 in CodeCarbon's GitHub repository.

Table 1: All measures of the TIREx tracker and their supported platforms (✓ supported, (✓) partial, ✗ unsupported, 🐧 Linux, ⊞ Windows, macOS). Partial support indicates support for only some vendors (e.g., only Nvidia GPUs). Python (🐍) and Java (☕) metrics are only available in their respective wrappers. Measures are constant (⊙, i.e., never change during tracking), cumulative (→, e.g., time or energy) or periodically polled time series values (•••).

	Identifier	Description	Platform			Type
			🐧	⊞		
OS	OS_NAME	Name and version of the operating system under which is currently running.	✓	✓	✓	⊙
	OS_KERNEL	The version of the kernel that the operating system is running on.	✓	✓	✓	⊙
Time	TIME_START	Timestamp when the tracking was started.	✓	✓	✓	⊙
	TIME_STOP	Timestamp when the tracking was stopped.	✓	✓	✓	⊙
	TIME_ELAPSED_WALL_CLOCK_MS	The ("real") wall clock time elapsed during tracking.	✓	✓	✓	→
	TIME_ELAPSED_USER_MS	Time spent in the platform's user mode.	✓	✓	✓	→
	TIME_ELAPSED_SYSTEM_MS	Time spent in the platform's system mode.	✓	✓	✓	→
CPU	CPU_USED_PROCESS_PERCENT	CPU usage of the tracked process in percent per logical CPU cores.	✓	✓	✓	•••
	CPU_USED_SYSTEM_PERCENT	CPU usage of the entire system in percent per logical CPU cores.	✓	✓	✓	•••
	CPU_AVAILABLE_SYSTEM_CORES	Number of CPU cores available in the system.	✓	✓	✓	⊙
	CPU_ENERGY_SYSTEM_JOULES	The energy consumed by the CPU by the entire system over the tracked period in joules.	✓	✓	✓	→
	CPU_FEATURES	List of hardware features the CPU supports (e.g., the instruction set, encryption capabilities).	✓	✓	✓	⊙
	CPU_FREQUENCY_MHZ	Current CPU speed in megahertz.	✓	✓	✗	•••
	CPU_FREQUENCY_MIN_MHZ	Minimum possible CPU speed in megahertz.	✓	✓	✗	⊙
	CPU_FREQUENCY_MAX_MHZ	Maximum possible CPU speed in megahertz.	✓	✓	✗	⊙
	CPU_VENDOR_ID	A textual name for the vendor of the CPU.	✓	✓	✓	⊙
	CPU_BYTE_ORDER	The endianness (big-, little-, or mixed-endian) used by the CPU.	✓	✓	✓	⊙
	CPU_ARCHITECTURE	The architecture (x86, x86_64, ARM, ...) of the CPU.	✓	✓	✓	⊙
	CPU_MODEL_NAME	The name of the concrete CPU model.	✓	✓	✓	⊙
	CPU_CORES_PER_SOCKET	Number of CPU cores located on a single physical socket.	✓	✓	✓	⊙
	CPU_THREADS_PER_CORE	Number of logical CPU cores (threads) per core.	✓	✓	✓	⊙
	CPU_CACHES	The sizes of each CPU cache (e.g., L1, L2, L3) in kilobytes.	✓	✓	✓	⊙
	CPU_VIRTUALIZATION	The virtualization technology supported by the CPU (VT-x or AMD-V), if any.	✓	✓	✓	⊙
RAM	RAM_USED_PROCESS_KB	RAM usage of the tracked process in kilobytes.	✓	✓	✓	•••
	RAM_USED_SYSTEM_MB	RAM usage of the entire system in megabytes.	✓	✓	✓	•••
	RAM_AVAILABLE_SYSTEM_MB	Amount of RAM available in the system in megabytes.	✓	✓	✓	⊙
	RAM_ENERGY_SYSTEM_JOULES	The energy consumed by the DRAM by the entire system over the tracked period in joules.	✓	✓	✗	→
GPU	GPU_SUPPORTED	True if a GPU is detected in the system, and we support tracking it.	✓	✓	✓	⊙
	GPU_MODEL_NAME	The name of the GPU model detected in the system.	(✓)	(✓)	✗	⊙
	GPU_DRIVER_VERSION	The version of the installed GPU drivers.	(✓)	(✓)	✗	⊙
	GPU_NUM_CORES	Number of GPU cores available in the system.	(✓)	(✓)	✗	⊙
	GPU_USED_PROCESS_PERCENT	GPU usage of the tracked process in percent.	(✓)	(✓)	✗	•••
	GPU_USED_SYSTEM_PERCENT	GPU utilization of the entire system in percent.	(✓)	(✓)	✗	•••
	GPU_VRAM_USED_PROCESS_MB	GPU VRAM usage of the tracked process in megabytes.	(✓)	(✓)	✗	•••
	GPU_VRAM_USED_SYSTEM_MB	GPU VRAM usage of the entire system in megabytes.	(✓)	(✓)	✗	•••
	GPU_VRAM_AVAILABLE_SYSTEM_MB	Amount of GPU VRAM available in the system in megabytes.	(✓)	(✓)	✗	⊙
	GPU_ENERGY_SYSTEM_JOULES	The energy consumed by the GPU for the entire system in joules.	(✓)	(✓)	✗	→
Git	GIT_IS_REPO	True if the current working directory is (part of) a Git repository.	✓	✓	✓	⊙
	GIT_ROOT	Holds the "working directory" of the repository (the path to root of the repository's file tree).	✓	✓	✓	⊙
	GIT_HASH	SHA1 hash of all files checked into the repository.	✓	✓	✓	⊙
	GIT_LAST_COMMIT_HASH	Latest Git commit SHA1 hash.	✓	✓	✓	⊙
	GIT_BRANCH	Checked-out Git branch name.	✓	✓	✓	⊙
	GIT_BRANCH_UPSTREAM	Upstream branch of the checked-out Git branch name.	✓	✓	✓	⊙
	GIT_TAGS	List of Git tag(s) at the current commit, if any.	✓	✓	✓	⊙
	GIT_REMOTE ORIGIN	URL of the `origin` remote if exists.	✓	✓	✓	⊙
	GIT_UNCOMMITTED_CHANGES	True if some changes are not yet committed.	✓	✓	✓	⊙
	GIT_UNPUSHED_CHANGES	True if some changes are not yet pushed.	✓	✓	✓	⊙
	GIT_UNCHECKED_FILES	True if there are files that are not ignored (by a `.gitignore` file) and also not checked into the repository.	✓	✓	✓	⊙
	GIT_ARCHIVE_DIR	Creates a zip archive containing all files inside the repository that are not ignored by the `gitignore`.	✓	✓	✓	⊙
Python	PYTHON_VERSION	Python version (e.g. 3.12.0) used to run the tracked program.	✓	✓	✓	⊙
	PYTHON_EXECUTABLE	Python executable used to run the program.	✓	✓	✓	⊙
	PYTHON_MODULES	Python modules visible in the current environment.	✓	✓	✓	⊙
	PYTHON_INSTALLED_PACKAGES	Python packages installed in the current environment.	✓	✓	✓	⊙
	7 more	Python executable, arguments, script path, script contents, interactive, notebook path, notebook contents.	✓	✓	✓	⊙
Java	JAVA_VERSION	Java Runtime Environment version (e.g., 21.0.6) used to run the tracked program.	✓	✓	✓	⊙
	JAVA_HOME	Java installation directory.	✓	✓	✓	⊙
	JAVA_CLASS_PATH	Java class path, i.e., where to search for classes and packages.	✓	✓	✓	⊙
	17 more	Java system properties available from `System.getProperties()`.	✓	✓	✓	⊙

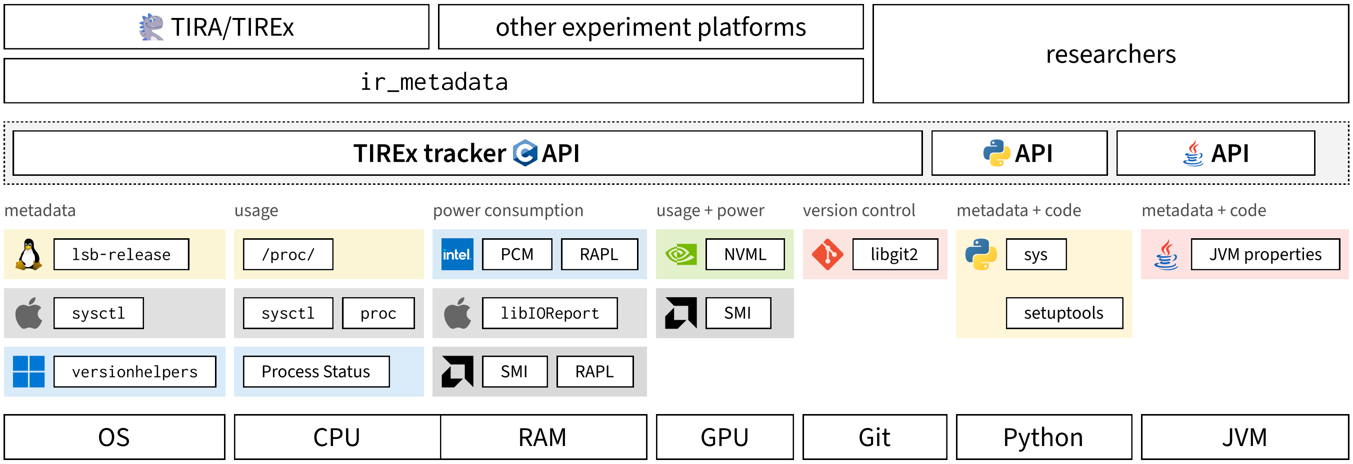


Figure 1: Overview of the interfaces and components of the TIREx tracker APIs (from top to bottom): integrations with experiment platforms (e.g., TIRA/TIREx), format (i.e., `ir_metadata`), tracking logic (C/Python/JVM), sources, and tracked entities.

in a set of measures (e.g., OS information, CPU, GPU, RAM, or energy). Providers share a common interface to easily support additional metrics or platforms in the future. Table 1 lists the metrics and metadata available in the TIREx tracker, grouped by the their "target" (OS, CPU, Git, . . .). Overall, the TIREx tracker collects up to 69 measures (depending on the language binding), which we categorize into (1) *constants* that remain unchanged during execution (e.g., hardware specifications, OS details, and repository metadata), (2) *cumulative measures* that aggregate or cumulate a value over the tracked time span (e.g., time elapsed or energy consumed), and (3) *time series measures* that continuously track a metric (e.g., CPU and RAM usage) over time.

Operating System Metadata. To capture general system information, we fetch the operating system (OS) name, version, and kernel details (the latter is mostly relevant on Linux). On Linux, we get the information from `/etc/lsb-release` for Linux Standard Base (LSB)-compliant distributions and fall back to `/etc/os-release` for broader compatibility with non-compliant systems like Fedora. On macOS, we use `sysctl` to obtain the OS version and kernel details. And, for Windows, we query the system provided `versionhelpers.h`.[8]

Git Versioning Metadata. Git[9] is a popular version control system commonly used in (research) software development. By providing metadata about the Git repository an experiment is run from, the TIREx tracker supports the reproducibility and transparency of the research, e.g., it may be useful to know the last commit hash or whether the repository contains untracked, uncommitted, or unpushed files. Such metadata allows for verification of the source code used at execution time to ensure experiment integrity. We track all Git-related metadata using the statically linked libgit2[10] library. This keeps TIREx tracker in a single self-contained binary, so we do not rely on any external Git installation and guarantee consistent access to version control metadata across environments.

CPU Usage and Energy Consumption. To track CPU-related metrics, we leverage a combination of hardware-specific tools and system-dependent APIs to ensure broad compatibility and to avoid relying on manual user configurations. To determine the CPU's capabilities, PyTorch's cross-platform cpuinfo[11] library is used if available. For unsupported measures, we fall back to platform-specific calls, e.g., `sysctl` on macOS. Runtime efficiency metrics on Linux are extracted from the `/proc/` directory,[12] that, for instance, contains the CPU utilization over time. For energy measurements, tools on macOS typically rely on the `powermetrics` command line tool. But since that tool requires root privileges, we instead directly use the `libIOReport` library that is internally used by `powermetrics`, enabling energy data collection without elevated permissions. On Windows and Linux, energy measurement tools usually use the Running Average Power Limit (RAPL) interface and Intel Power-Gadget [1, 10, 17]. Both require manual setup, and Power-Gadget is no longer officially supported. We additionally and alternatively use Intel PCM, the successor to Power-Gadget, which is compiled directly into our binary, eliminating the need for external dependencies or reliance on deprecated software on Intel-based systems. For AMD, we are working on integrating the AMD SMI library[13], which provides, among others, energy information for CPU, GPU, and RAM on supported AMD setups.

RAM Usage and Energy Consumption. Our RAM usage metrics are tracked using platform-specific APIs, similar to the CPU monitoring. On Linux, memory usage is again retrieved from the `/proc/` file system, that provides system-wide and process-specific memory statistics. On Windows, we utilize the Process Status API[14] and the System Info API[15] to query process-specific and global memory usage respectively. Finally, for macOS, we use the builtin `libproc`[16] for process information and `sysconf` for global RAM usage. For RAM

---

[8] Note that Windows does not distinguish versions 10 and up: learn.microsoft.com/windows-hardware/drivers/ddi/wdm/ns-wdm-_osversioninfoexw

[9] https://git-scm.com/

[10] https://libgit2.org/

[11] https://github.com/pytorch/cpuinfo

[12] https://linux.die.net/man/5/proc

[13] https://rocm.docs.amd.com/projects/amdsmi/en/latest/

[14] https://learn.microsoft.com/en-us/windows/win32/api/_psapi/

[15] https://learn.microsoft.com/en-us/windows/win32/api/sysinfoapi/

[16] https://developer.apple.com/documentation/kernel/sys#3571036

energy consumption metrics, we rely on RAPL and Intel PCM on both Linux and Windows, leveraging the same infrastructure used for CPU energy tracking. RAM energy measurement on macOS is unsupported due to the lack of appropriate system interfaces.

GPU Usage and Energy Consumption. For GPU usage, we currently support NVIDIA GPUs on Linux and Windows through the NVIDIA Management Library (NVML),[17] which is part of the GPU driver. NVML provides insights into GPU utilization, VRAM usage, and energy consumption, and therefore, constitutes the primary data source for our measure implementation. On macOS, dedicated VRAM monitoring is not applicable due to the shared memory architecture of Apple GPUs. While GPU energy consumption can be queried on macOS via `libIOReport`, we do not currently plan to support GPU metrics on macOS. Monitoring AMD GPUs remains a challenge, and resource tracking tools similar to the TIREx tracker usually lack support for AMD GPUs [17]. To bridge this gap, we are currently integrating the AMD SMI library to track the energy consumption of AMD GPUs.

Process-Specific Energy Consumption Tracking. Typically, hardware APIs merely report the overall energy consumption of whole system components (e.g., entire CPUs, GPUs, or RAM) but do not narrow down energy usage per process. As a means to estimate process-specific energy consumption, one can assume a strong correlation of resource utilization with the resource's power draw. For example, if a process utilizes half of the CPU cores of a system, it seems fair to also attribute half of the CPU's energy consumption to that individual process. While this estimation relies on a typically not entirely true assumption (e.g., using a larger proportion of the CPU might be more efficient than using two smaller slices), it allows for meaningful per-process energy insights without requiring the process to be run in isolation and allows the TIREx tracker to track both overall CPU/GPU/RAM energy consumption and the consumption of the tracked process itself.

Python Environment Metadata. When used to track a Python program, we determine the Python version, executable, command line arguments, and visible modules using Python's built-in `sys` module.[18] A list of installed dependencies and their exact versions are queried via the `setuptools` package,[19] allowing for a comprehensive, pinned dependency list agnostic to the way dependencies are defined.[20] Additionally, we check if the Python program is run as an interactive (Jupyter) notebook. The entry point Python script and (if available) the Jupyter notebook are also recorded.

Java Environment Metadata. For programs running on the Java virtual machine (JVM), we collect additional Java environment metadata using Java's built-in system properties,[21] providing information about the Java runtime environment (JRE), vendor (e.g., Oracle), virtual machine (JVM), and class path (i.e., the paths to search when importing classes or packages), properties essential to define the behavior in which compiled Java byte code is executed.

(a) In C (using the `tirex_tracker.h` header file):

```c
#include <tirex_tracker.h>
int main() {
  // Configure measures to track.
  tirexMeasureConf conf[] = {
    {TIREX_TIME_ELAPSED_WALL_CLOCK_MS, TIREX_AGG_NO},
    tirexNullMeasureConf // sentinel value
  };
  tirexTrackingHandle* handle;
  tirexStartTracking(conf, 100, &handle);
  // Do something...
  tirexResult* result;
  tirexStopTracking(handle, &result);
  // Analyze the results.
  tirexResultFree(result);
}
```

(b) In Python (using the `tirex-tracker` PyPI package):

```python
from tira.tracker import tracking
with tracking() as results:
  # Do something...
print(results)
```

(c) In Java (using the `io.tira:tirex-tracker` Maven package):

```java
import io.tira.tracker.*;
void main() {
  var result = Tracker.track(() -> {
    // Do something...
  });
  System.out.println(result);
}
```

(d) In Kotlin (using the `io.tira:tirex-tracker` Maven package):

```kotlin
import io.tira.tracker.*;
fun main() {
  val result = track {
    // Do something...
  }
  println(result)
}
```

Listing 1: Using TIREx tracker in C, Python, and Java/Kotlin.

3.2 A Native, Low-level, and Cross-Platform Experiment Tracking Library

Most of the measures outlined above (Section 3.1; see Table 1) are implemented in the TIREx tracker's native library, a lightweight, low-level and cross-platform C API consisting of a single header file with fewer than 150 lines of code[22] and compact binaries (smaller than 4 MB). The library's C API is designed to be both simple and flexible. Users can request any of the supported measures using the identifiers from Table 1 and, with that list of requested measures, call the `tirexStartTracking` function to start a tracking thread (see Listing 1). For polled time series measures, users can specify a polling interval to balance overhead and accuracy. Moreover, batched aggregations (i.e., min, max, mean, or no aggregation) can be configured to reduce the amount of data stored in the time series (e.g., for polling RAM usage every 100 ms to not miss spikes but considering only the 1 s-maximum overall). To stop the tracking, users

---

[17] https://developer.nvidia.com/management-library-nvml

[18] https://docs.python.org/3/library/sys.html

[19] https://github.com/pypa/setuptools

[20] Even though some researchers choose to pin versions in a `requirements.txt` file, Python imposes no restrictions on the dependency management tool, so we cannot rely just on parsing the `requirements.txt` file.

[21] https://docs.oracle.com/en/java/javase/23/docs/api/system-properties.html

[22] Comments were not counted.

call the `tirexStopTracking` function, which returns a `tirexResult` data structure containing the requested metrics and metadata (see Listing 1). After analyzing the results, users can free the memory allocated by the TIREx tracker's library by calling `tirexFreeResult`. Optionally, users can also enable fine-grained logging by setting a log callback function using `tirexSetLogCallback`.

The TIREx tracker's C library is designed to fail fast and safely when a certain measure is not available on a system in such situations. We expose the API as a pure C header file to avoid name mangling issues common in C++ headers, facilitating an easy integration with various higher-level programming languages, e.g., the Python and Java-specific wrappers in Sections 3.3. To reduce potential incompatibilities due to missing shared libraries, we compile most dependencies statically into the binary and only rely on system libraries that are part of drivers (e.g., NVML). Additional user-side configuration is only required for energy tracking in rare situations, e.g., for tracking AMD CPUs with RAPL. Our library intentionally only exposes the most basic functionality (as opposed to, e.g., some profiling libraries like omniperf[23]) to avoid dependencies, make the library lightweight, and keep the API simple and easy to use.

3.3 Simplified Retrieval Experiment Tracking in Python, Java and Kotlin

The most popular information retrieval frameworks are written in Java (e.g., Lucene, Anserini, and Terrier [15, 20]) or Python (e.g., Pyserini and PyTerrier [12, 13]). To integrate the TIREx tracker into these frameworks and to open up hardware metrics and metadata tracking for Python and Java users, we implement additional language-specific, high-level wrappers on top of the C library: (1) the `tirex-tracker` package for Python and (2) for JVM-based languages like Java or Kotlin the `io.tira:tirex-tracker` package.

Python Wrapper. The TIREx tracker's Python wrapper uses the foreign function calling module `ctypes`[24], which is built into Python, to expose the native C library through a type-safe, tested, and lightweight API. The Python library provides the `start_tracking` and `stop_tracking` Python functions that work analogous to the native `tirexStartTracking` and `tirexStopTracking` functions which internally parse the results into a Python dictionary and free the native memory. For easier use, we also provide the same functionality as a context manager (shown in Listing 1), and as a function decorator. By using either the context manager or the function decorator, users can easily avoid memory leaks due to dangling references to unstopped tracking threads or results. The bundled Python wheels are just 3 MB in size and can be installed from PyPI.[25]

Java/Kotlin Wrapper. Similarly, the TIREx tracker's Java package wraps the native C library for JVM-based applications, using Java Native Access (JNA)[26]. Again, the static `startTracking` and `stopTracking` Java methods analogous to the native library's `tirexStartTracking`/`tirexStopTracking` functions handle the result parsing into a Java hash map and are complemented with convenience methods for tracking lambdas (for Java) or inline blocks (for Kotlin) of code (see the `track` method shown in Listing 1). The compiled Java JAR is also just 3 MB big and can be installed with Maven or Gradle from the GitHub Package Registry.[27]

(a) Upload via the bash command line:

```
tira-cli upload \
    --directory '<directory-with-run>' \
    --dataset '<ir-datasets-id>'
```

(b) Upload in Python experiments:

```
from tira.third_party_integrations import \
  persist_and_normalize_run
from tira.tirex.tracker import tracking
from pyterrier import get_dataset

dataset = get_dataset("irds:<ir-datasets-id>")
with tracking() as tracking_results:
  run = ...# Retrieve results for the dataset's topics

persist_and_normalize_run(
  run,
  system_name="<system-name>",
  upload_to_tira=dataset,
  tracking_results=tracking_results,
)
```

Listing 2: Uploading a run and its `ir_metadata`-compatible metrics/metadata to TIRA/TIREx via bash and Python.

3.4 Standardized Metadata Export by Extending the `ir_metadata` Specification

The TIREx tracker API tracks metrics and metadata that are even relevant to many fields in computer science beyond reproducibility in information retrieval. Its generic API allows for flexibly reading this data in a structured way. Since, for information retrieval experiments, Breuer et al. [3] have already standardized the `ir_metadata` format to capture the most relevant IR-specific metadata, we extend the TIREx tracker to automate the export of the collected metrics by adding a `tirexResultExportIrMetadata` function which exports the tracking results to an `ir_metadata`-compatible file.

With the TIREx tracker's plethora of measures (see Table 1), a lot of metadata does not directly "fit" into the current `ir_metadata` specification (version 0.1), which, for example, does not have a field for storing most of the non-constant hardware resource usage metrics (e.g., the time series of RAM used by the tracked process). Hence, we extend the existing `ir_metadata` schema and propose version 0.2-beta to accommodate all measures from Table 1.[28]

Easy Retrieval Experiment Tracking with the TIREx Tracker. Our language wrappers (Section 3.3) extend the natively exported metadata by additional language-specific metadata (e.g., the Python version or the Java class path; see Table 1) by adding these fields to the `ir_metadata` file. Thus, the TIREx tracker's language bindings make it easy to track hardware metrics and metadata in retrieval experiments, regardless of the retrieval framework. Users would,

---

[23] https://rocm.github.io/rocprofiler-compute/introduction.html
[24] https://docs.python.org/3/library/ctypes.html
[25] https://pypi.org/project/tirex-tracker
[26] https://java-native-access.github.io/jna/5.16.0/javadoc
[27] https://github.com/tira-io/tirex-tracker/packages/
[28] Our proposed specification of the `ir_metadata` version 0.2-beta is available online: https://github.com/tira-io/tirex-tracker#ir_metadata-extension

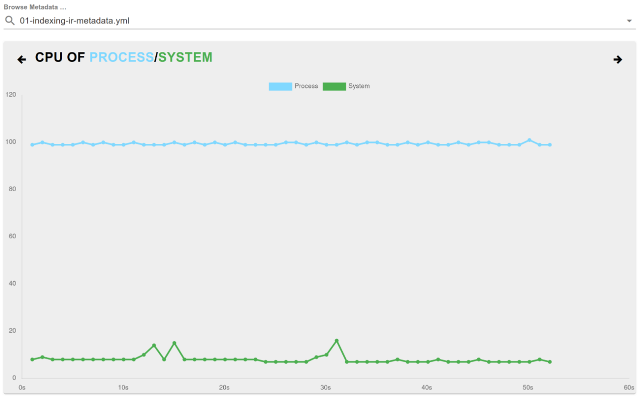

Browse Metadata ...
01-indexing-ir-metadata.yml

Figure 2: Rendered CPU consumption with `ir_metadata` in TIRA (analogous for GPU/RAM usage).

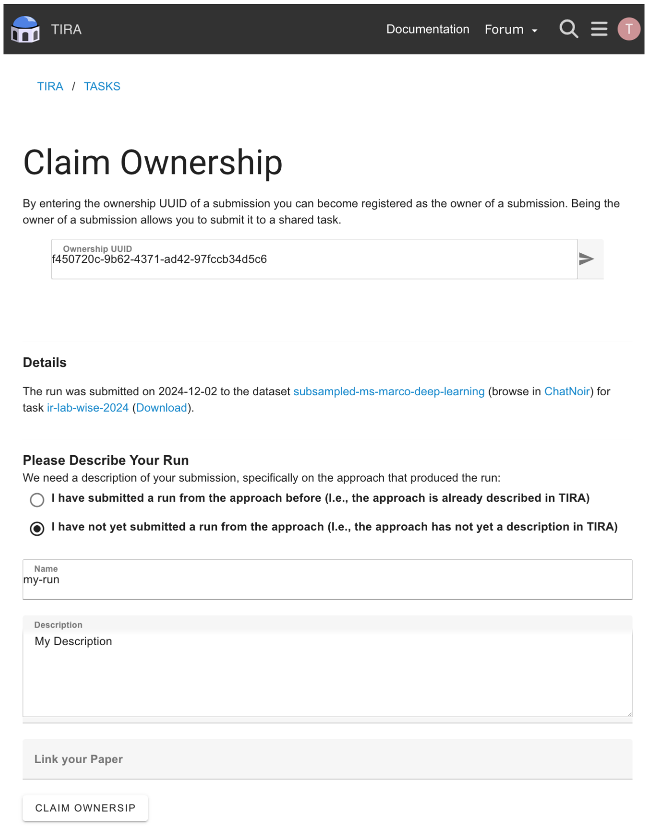
for example, in Python use the `tracking` context manager to wrap the retriever (Listing 2). After running and evaluating their retrieval system, the user can export the `ir_metadata`-compatible run metadata, hardware metrics, and optional system metadata (e.g., the run name or description) using the `export_ir_metadata` function, and place it next to the run file (usually a TREC run file).

We further envision retrieval-framework-specific integrations of the TIREx tracker with popular retrieval experimentation frameworks like Pyserini and PyTerrier [12, 13], to also include the computation graph of the retrieval pipeline (e.g., the PyTerrier transformer modules the pipeline consists of) in the metadata.

4 Case Study: Run Submissions with `ir_metadata` to TIRA/TIREx

As the TIREx tracker allows to automatically track and export the metadata and resource metrics of information retrieval experiments into standardized files, a potential use case is to incorporate automated metadata collection into run submission platforms used for shared tasks such as TREC, CLEF, or NTCIR with only very minor modifications of the submission platform itself. For instance, the `ir_metadata` specification can be incorporated into the uploaded run files themselves (which does not require modifications of the upload form) or can be uploaded as additional small `ir_metadata` file (usually only a few kilobytes; does require a modification of the upload form). Because the workload (i.e., metadata collection and standardized export) is handled by the TIREx tracker APIs which are operated by the participant, the experiment platforms themselves do not require significant additional maintenance effort and instead can contribute improvements and/or modifications to fit specialized needs to the TIREx tracker, to benefit all other experiment platforms as well. To exemplify one such platform integration, we integrate the TIREx tracker into (anonymous) run file submission on the TIRA/TIREx platform [8, 9].

In TIRA/TIREx, we use the TIREx tracker to monitor and render tracked system metrics and metadata for runs that are persisted and uploaded to the TIRA/TIREx. We envision that, ideally, every time when an experimenter writes a run file to disk, it is uploaded in the background to TIRA so that a rich data source of runs together with `ir_metadata` emerges that can be used by the whole IR community. Therefore, we modify the `persist_and_normalize_run` method available in TIRA (alternative methods to persist run files to disk could be modified accordingly). This method calls the TIREx tracker’s `export_ir_metadata` function to generate the `ir_metadata` file and then uploads the run file and the `ir_metadata` file to TIRA. We further add validators for `ir_metadata` to allow organizers to ensure that runs are only accepted when the `ir_metadata` is valid and includes all fields required by organizers. We also incorporate anonymous, unauthenticated run submissions into TIRA so that the collection of runs with `ir_metadata` does not require addtional effort from experimenters (though TIRA.io still only accepts valid uploads). For anonymous submissions, an identifier is displayed to users with which they can claim ownership of the submission on the TIRA website. Listing 2 shows an (anonymous) run submission with `ir_metadata`-compatible metadata using the command line and an experiment in Python that persists the run and metadata via the `persist_and_normalize_run` method, uploads it to TIRA, and finally displays the identifier. The returned identifier is used in Figure 3 to claim ownership for the submitted run. TIRA can also render the `ir_metadata` including the experiment’s resource consumption (see Figure 2). This aims to encourage the development of evaluation methodologies that combine efficiency with effectiveness.



Claim Ownership

By entering the ownership UUID of a submission you can become registered as the owner of a submission. Being the owner of a submission allows you to submit it to a shared task.

Ownership UUID: f450720c-9b62-4371-ad42-97fccb34d5c6

Details

The run was submitted on 2024-12-02 to the dataset subsampled-ms-marco-deep-learning (browse in ChatNoir) for task ir-lab-wise-2024 (Download).

Please Describe Your Run

We need a description of your submission, specifically on the approach that produced the run:

○ I have submitted a run from the approach before (I.e., the approach is already described in TIRA)

● I have not yet submitted a run from the approach (I.e., the approach has not yet a description in TIRA)

Name: my-run

Description: My Description

Figure 3: Claiming a submission with `ir_metadata` in TIRA.

This workflow adds only very little complexity for experimenters but still captures an extensive set of system metrics and run metadata (like hardware specifics and the code from the Git repository), that can later be used to reproduce "interesting" submissions to a shared task as Docker images for follow-up experiments.

5 Conclusion

We have introduced the TIREx tracker, a lightweight tool to automatically track metadata and hardware usage during retrieval experiments. By exporting the various information as `ir_metadata` files, the TIREx tracker aids reproducibility and facilitates efficiency reporting (e.g., run time and energy consumption) while imposing only minimal additional effort on researchers.

To maximize compatibility and allow usage across different systems, we expose the tracking API to C, Java, and Python and employ an extensive continuous integration and delivery suite that compiles, tests, and publishes all binaries and packages. In a case study on the TIRA/TIREx platform, we have already demonstrated the versatility of the TIREx tracker and its applicability to real-world experiment tracking and evaluation.

The TIREx tracker is extensible and we will continue to maintain and improve it as the default efficiency tracker of the TIRA project and the TIREx platform. We also plan further improvements to stability and reliability across diverse systems (e.g., support of AMD GPUs) and the integration of the tracker into widely used frameworks such as PyTerrier/Terrier, Pyserini/Anserini, and PISA, using the respective language wrappers.

References

[1] Lasse F. Wolff Anthony, Benjamin Kanding, and Raghavendra Selvan. 2020. Carbontracker: Tracking and Predicting the Carbon Footprint of Training Deep Learning Models. https://doi.org/10.48550/arXiv.2007.03051 arXiv:2007.03051

[2] Timo Breuer, Nicola Ferro, Maria Maistro, and Philipp Schaer. 2021. repro_eval: A Python Interface to Reproducibility Measures of System-Oriented IR Experiments. In *Proceedings of ECIR 2021 (LNCS, Vol. 12657)*, Djoerd Hiemstra, Marie-Francine Moens, Josiane Mothe, Raffaele Perego, Martin Potthast, and Fabrizio Sebastiani (Eds.). Springer, Berlin, 481–486. https://doi.org/10.1007/978-3-030-72240-1_51

[3] Timo Breuer, Jüri Keller, and Philipp Schaer. 2022. ir_metadata: An Extensible Metadata Schema for IR Experiments. In *Proceedings of SIGIR 2022*, Enrique Amigó, Pablo Castells, Julio Gonzalo, Ben Carterette, J. Shane Culpepper, and Gabriella Kazai (Eds.). ACM, New York, 3078–3089. https://doi.org/10.1145/3477495.3531738

[4] Sebastian Bruch, Maik Fröbe, Tim Hagen, Franco Maria Nardini, and Martin Potthast. 2025. ReNeuIR at SIGIR 2025: The Fourth Workshop on Reaching Efficiency in Neural Information Retrieval. To appear. In *Proceedings of SIGIR 2025*. ACM, New York, 4 pages.

[5] Ryan Clancy, Nicola Ferro, Claudia Hauff, Jimmy Lin, Tetsuya Sakai, and Ze Zhong Wu. 2019. Overview of the 2019 Open-Source IR Replicability Challenge (OSIRRC 2019). In *Proceedings of OSIRRC@SIGIR 2019 (CEUR Workshop Proceedings, Vol. 2409)*, Ryan Clancy, Nicola Ferro, Claudia Hauff, Jimmy Lin, Tetsuya Sakai, and Ze Zhong Wu (Eds.). CEUR-WS.org, Aachen, 1–7. https://ceur-ws.org/Vol-2409/invited01.pdf

[6] Nicola Ferro, Norbert Fuhr, Kalervo Järvelin, Noriko Kando, Matthias Lippold, and Justin Zobel. 2016. Increasing Reproducibility in IR: Findings from the Dagstuhl Seminar on "Reproducibility of Data-Oriented Experiments in e-Science". *SIGIR Forum* 50, 1 (2016), 68–82. https://doi.org/10.1145/2964797.2964808

[7] Nicola Ferro and Diane Kelly. 2018. SIGIR Initiative to Implement ACM Artifact Review and Badging. *SIGIR Forum* 52, 1 (2018), 4–10. https://doi.org/10.1145/3274784.3274786

[8] Maik Fröbe, Jan Heinrich Reimer, Sean MacAvaney, Niklas Deckers, Simon Reich, Janek Bevendorff, Benno Stein, Matthias Hagen, and Martin Potthast. 2023. The Information Retrieval Experiment Platform. In *Proceedings of SIGIR 2023*, Hsin-Hsi Chen, Wei-Jou (Edward) Duh, Hen-Hsen Huang, Makoto P. Kato, Josiane Mothe, and Barbara Poblete (Eds.). ACM, New York, 2826–2836. https://doi.org/10.1145/3539618.3591888

[9] Maik Fröbe, Matti Wiegmann, Nikolay Kolyada, Bastian Grahm, Theresa Elstner, Frank Loebe, Matthias Hagen, Benno Stein, and Martin Potthast. 2023. Continuous Integration for Reproducible Shared Tasks with TIRA.io. In *Proceedings of ECIR 2023 (LNCS, Vol. 13982)*, Jaap Kamps, Lorraine Goeuriot, Fabio Crestani, Maria Maistro, Hideo Joho, Brian Davis, Cathal Gurrin, Udo Kruschwitz, and Annalina Caputo (Eds.). Springer, Berlin, 236–241. https://doi.org/10.1007/978-3-031-28241-6_20

[10] Peter Henderson, Jieru Hu, Joshua Romoff, Emma Brunskill, Dan Jurafsky, and Joelle Pineau. 2020. Towards the Systematic Reporting of the Energy and Carbon Footprints of Machine Learning. https://doi.org/10.48550/arXiv.2002.05651 arXiv:2002.05651

[11] Jeremy Leipzig, Daniel Nüst, Charles Tapley Hoyt, Karthik Ram, and Jane Greenberg. 2021. The Role of Metadata in Reproducible Computational Research. *Patterns* 2, 9 (2021), 100322. https://doi.org/10.1016/j.patter.2021.100322

[12] Jimmy Lin, Xueguang Ma, Sheng-Chieh Lin, Jheng-Hong Yang, Ronak Pradeep, and Rodrigo Frassetto Nogueira. 2021. Pyserini: A Python Toolkit for Reproducible Information Retrieval Research with Sparse and Dense Representations. In *Proceedings of SIGIR 2021*, Fernando Diaz, Chirag Shah, Torsten Suel, Pablo Castells, Rosie Jones, and Tetsuya Sakai (Eds.). ACM, New York, 2356–2362. https://doi.org/10.1145/3404835.3463238

[13] Craig Macdonald, Nicola Tonellotto, Sean MacAvaney, and Iadh Ounis. 2021. PyTerrier: Declarative Experimentation in Python from BM25 to Dense Retrieval. In *Proceedings of CIKM 2021*, Gianluca Demartini, Guido Zuccon, J. Shane Culpepper, Zi Huang, and Hanghang Tong (Eds.). ACM, New York, 4526–4533. https://doi.org/10.1145/3459637.3482013

[14] Margaret Mitchell, Simone Wu, Andrew Zaldivar, Parker Barnes, Lucy Vasserman, Ben Hutchinson, Elena Spitzer, Inioluwa Deborah Raji, and Timnit Gebru. 2019. Model Cards for Model Reporting. In *Proceedings of FAT* 2019*, Danah Boyd and Jamie H. Morgenstern (Eds.). ACM, New York, 220–229. https://doi.org/10.1145/3287560.3287596

[15] Iadh Ounis, Gianni Amati, Vassilis Plachouras, Ben He, Craig Macdonald, and Douglas Johnson. 2005. Terrier Information Retrieval Platform. In *Proceedings of ECIR 2005 (LNCS, Vol. 3408)*, David E. Losada and Juan M. Fernández-Luna (Eds.). Springer, Berlin, 517–519. https://doi.org/10.1007/978-3-540-31865-1_37

[16] Martin Potthast, Tim Gollub, Matti Wiegmann, and Benno Stein. 2019. TIRA Integrated Research Architecture. In *Information Retrieval Evaluation in a Changing World – Lessons Learned from 20 Years of CLEF*, Nicola Ferro and Carol Peters (Eds.). The IR Series, Vol. 41. Springer, Berlin, 123–160. https://doi.org/10.1007/978-3-030-22948-1_5

[17] Shivadharshan S, Akilesh P, Rajrupa Chattaraj, and Sridhar Chimalakonda. 2024. CPPJoules: An Energy Measurement Tool for C++. https://doi.org/10.48550/arXiv.2412.13555 arXiv:2412.13555

[18] Harrisen Scells, Shengyao Zhuang, and Guido Zuccon. 2022. Reduce, Reuse, Recycle: Green Information Retrieval Research. In *Proceedings of SIGIR 2022*, Enrique Amigó, Pablo Castells, Julio Gonzalo, Ben Carterette, J. Shane Culpepper, and Gabriella Kazai (Eds.). ACM, New York, 2825–2837. https://doi.org/10.1145/3477495.3531766

[19] Ellen M. Voorhees, Shahzad Rajput, and Ian Soboroff. 2016. Promoting Repeatability Through Open Runs. In *Proceedings of the Seventh International Workshop on Evaluating Information Access, EVIA 2016, a Satellite Workshop of the NTCIR-12 Conference, National Center of Sciences, Tokyo, Japan, June 7, 2016*, Emine Yilmaz and Charles L. A. Clarke (Eds.). National Institute of Informatics (NII).

[20] Peilin Yang, Hui Fang, and Jimmy Lin. 2017. Anserini: Enabling the Use of Lucene for Information Retrieval Research. In *Proceedings of SIGIR 2017*, Noriko Kando, Tetsuya Sakai, Hideo Joho, Hang Li, Arjen P. de Vries, and Ryen W. White (Eds.). ACM, New York, 1253–1256. https://doi.org/10.1145/3077136.3080721

[21] Guido Zuccon, Harrisen Scells, and Shengyao Zhuang. 2023. Beyond CO_2 Emissions: The Overlooked Impact of Water Consumption of Information Retrieval Models. In *Proceedings of ICTIR 2023*, Masaharu Yoshioka, Julia Kiseleva, and Mohammad Aliannejadi (Eds.). ACM, New York, 283–289. https://doi.org/10.1145/3578337.3605121