RastQC: High-Performance Sequencing Quality Control Written in Rust

Kuan-Lin Huang[1#]

[1] Department of Genetics and Genomic Sciences, Department of Artificial Intelligence and Human Health, Icahn School of Medicine at Mount Sinai, New York, NY 10029, USA.

[#]Corresponding Author: Kuan-lin Huang, Ph.D. kuan-lin.huang@mssm.edu

Abstract

Quality control (QC) of high-throughput sequencing data is a critical first step in genomics analysis pipelines. FastQC has served as the *de facto* standard for sequencing QC for over a decade, but its Java runtime dependency introduces startup overhead, elevated memory consumption, and deployment complexity. Here we present RastQC, a complete reimplementation of FastQC in Rust that provides all 12 standard QC modules with matching algorithms, plus 3 additional long-read QC modules, MultiQC-compatible output formats, native MultiQC JSON export, a built-in multi-file summary dashboard, and a web-based report viewer. RastQC also supports SOLiD colorspace reads, Oxford Nanopore Fast5/POD5 formats, standard input streaming, intra-file parallelism, and QC-aware exit codes for workflow integration. We benchmarked RastQC against FastQC v0.12.1 on both synthetic datasets (100K-1M reads) and real whole-genome sequencing data spanning five model organisms: *Escherichia coli*, *Saccharomyces cerevisiae*, *Drosophila melanogaster*, *Mus musculus*, and *Homo sapiens*. Despite running 15 modules (vs. 11 in FastQC), RastQC achieves comparable speed while using 4-9x less memory (59-125 MB vs. 551-638 MB). On real genome data, RastQC matches FastQC speed on most organisms while achieving 100% module-level concordance (55/55 module calls identical across all organisms for the 11 shared modules). RastQC compiles to a single 2.1 MB static binary with no external dependencies, representing a 102x reduction in deployment footprint. RastQC is freely available at https://github.com/Huang-lab/RastQC under the MIT license.

Introduction

Next-generation sequencing (NGS) has become the foundation of modern genomics research, with applications spanning whole-genome sequencing, RNA-seq, ChIP-seq, and single-cell assays. Before downstream analysis, quality control of raw sequencing data is essential to identify technical artifacts including base-calling errors, adapter contamination, GC bias, sequence duplication, and position-dependent quality degradation (Andrews, 2010; Patel & Jain, 2012). Failure to detect these issues can propagate systematic errors into variant calls, expression quantification, and other analyses.

FastQC (Andrews, 2010) has been the most widely used sequencing QC tool for over a decade, offering 12 diagnostic modules covering base quality, sequence content, duplication, adapter contamination, and k-mer enrichment. FastQC produces self-contained HTML reports that have become a standard deliverable in sequencing facilities worldwide. Tools such as MultiQC (Ewels et al., 2016) aggregate FastQC outputs across samples for project-level QC review.

Despite its widespread adoption, FastQC has practical limitations rooted in its Java implementation. The Java Virtual Machine (JVM) imposes a 2-3 second startup penalty per invocation, a minimum memory footprint of ~300 MB regardless of input size, and a runtime dependency that complicates deployment in minimal container images and heterogeneous HPC environments. For large-scale studies processing hundreds to thousands of samples, these overheads accumulate significantly.

Recent efforts to rewrite bioinformatics tools in systems languages such as Rust and C++ have demonstrated substantial performance improvements while maintaining correctness. Notable examples include minimap2 (Li, 2018), fastp (Chen et al., 2018), and various Rust-based tools in the noodles ecosystem. Rust in particular offers memory safety guarantees without garbage collection, zero-cost abstractions, and excellent concurrency support through the ownership model.

Here we present RastQC, a complete reimplementation of FastQC in Rust. RastQC implements all 12 QC modules with algorithms matching the original FastQC, produces MultiQC-compatible output formats, and adds a multi-file summary dashboard for batch QC review. We demonstrate through comprehensive benchmarking on both synthetic and real genome data from five model organisms that RastQC achieves significant improvements in execution speed and memory efficiency while maintaining full concordance with FastQC results.

Implementation

Architecture

RastQC is organized into six modules reflecting the analysis pipeline:

1. **I/O layer** (io/): Streaming parsers for FASTQ (plain, gzip, bzip2-compressed), BAM/SAM formats via the noodles library, Oxford Nanopore Fast5 (HDF5) and POD5 (Apache Arrow IPC) formats (feature-gated), SOLiD colorspace auto-detection and decoding, and standard input streaming. Sequences are processed one at a time to minimize memory footprint.
2. **QC modules** (modules/): Fifteen independent analysis modules, each implementing a common QCModule trait that defines process_sequence(), calculate_results(), merge support for parallel processing, and output generation methods. Modules accumulate per-position statistics during the streaming pass and compute final results lazily. All modules support accumulator-state merging for intra-file parallelism.
3. **Configuration** (config/): Embedded default adapter sequences (6 entries), contaminant sequences (15 entries), and pass/warn/fail thresholds matching FastQC defaults. All configurations are overridable via command-line flags. Kmer Content is enabled by default.
4. **Report generation** (report/): Self-contained HTML reports with inline SVG charts, tab-separated data files compatible with MultiQC, native MultiQC JSON output (--multiqc-json), per-file summary.txt with module-level status, and ZIP archives. A multi-file summary dashboard (HTML + TSV) is generated when processing multiple samples.
5. **Web GUI** (gui/): A built-in HTTP server (--serve) provides a browser-based interface for navigating and viewing reports, with auto-browser launch and multi-sample summary access.
6. **Parallel processing** (parallel/): Intra-file parallelism (--parallel) for large files buffers sequences in memory, distributes chunks across threads, and merges module states via accumulator combination. QC-aware exit codes (--exit-code) support automated pipeline gates.

QC Modules

RastQC implements all 12 FastQC modules with matching algorithms, plus 3 additional long-read QC modules (Table 1).

Table 1. QC modules implemented in RastQC. Modules marked with * are RastQC-exclusive.

Module	Description	Key Algorithm
Basic Statistics	Sequence count, length, %GC, total bases, encoding	Streaming counters; Phred encoding auto-detection
Per Base Sequence Quality	Quality distribution at each read position	Per-position quality histograms with adaptive base grouping
Per Tile Sequence Quality	Tile-specific quality deviations (Illumina)	Per-tile quality accumulation with 10% sampling after 10K reads
Per Sequence Quality Scores	Distribution of mean quality per read	Per-read average quality histogram
Per Base Sequence Content	A/T/G/C proportions per position	Per-position base counters with adaptive grouping
Per Sequence GC Content	GC% distribution vs. theoretical normal	GCModel fractional binning; bidirectional mode averaging
Per Base N Content	Unknown base frequency per position	Per-position N counters
Sequence Length Distribution	Read length variability	Adaptive binning maintaining ~50 display categories
Sequence Duplication Levels	Library complexity estimate	String-based tracking (first 50 bp); 100K unique cutoff
Overrepresented Sequences	High-frequency sequences with contaminant matching	Exact/approximate string matching; 1-mismatch tolerance
Adapter Content	Known adapter contamination by position	Substring matching of 12-mer adapters
K-mer Content	Positionally biased k-mers (enabled by default)	7-mer tracking with 2% sampling; binomial enrichment test
Read Length N50*	Assembly-style length statistics for long reads	Sorted-length N50/N90 computation
Quality Stratified Length*	Length distribution binned by quality tier	5-tier quality binning (Q<10 through Q40+)
Homopolymer Content*	Systematic homopolymer error detection	Run-length encoding; per-base run tracking

MultiQC Compatibility

RastQC output is designed for full compatibility with MultiQC's FastQC parser. The fastqc_data.txt file uses identical module names, column headers, and data formats. Key compatibility features include:

- **Filename field** populated in Basic Statistics for correct sample identification
- **Total Bases** metric included for modern MultiQC versions
- **summary.txt** populated with per-module PASS/WARN/FAIL status and filename
- **ZIP archive structure** matching the expected {sample}_fastqc/fastqc_data.txt path
- **Module names** exactly matching FastQC conventions

Multi-file Summary Dashboard

When processing multiple files (or with the --summary flag), RastQC generates:

- **summary.tsv**: Tab-separated matrix (samples x modules) for programmatic filtering
- **summary.html**: Color-coded HTML dashboard with clickable links to individual reports, aggregate tallies, and sortable columns

This provides MultiQC-like functionality integrated directly into the QC tool, enabling rapid triage of large sample batches without additional software.

Multi-threading

RastQC uses the rayon library for data-parallel file processing. Multiple input files are distributed across a configurable thread pool (defaulting to all available CPU cores). Within each file, processing is sequential to maintain the streaming memory model.

Methods

Benchmarking Environment

All benchmarks were performed on a single workstation:

- **CPU**: Intel Core i9-9900K @ 3.60 GHz (8 cores, 16 threads)
- **Memory**: 32 GB DDR4
- **Storage**: SSD
- **OS**: macOS 15.7.4 (Darwin 24.6.0)
- **RastQC**: v0.1.0, compiled with Rust (release profile, LTO enabled, opt-level 3)
- **FastQC**: v0.12.1, compiled from source with OpenJDK 11.0.30, -Xmx512m

Synthetic Datasets

We generated synthetic FASTQ datasets with realistic Illumina characteristics (Supplementary Figure S1). Details of synthetic dataset construction are provided in the Supplementary Information.

Real Genome Datasets

To evaluate performance on real sequencing data with natural biological complexity, we benchmarked on whole-genome Illumina sequencing data from five model organisms spanning the tree of life (Table 2).

Table 2. Real genome datasets used for benchmarking.

Organism	Species	Accession	Reads	Read Len	File Size
Bacteria	*Escherichia coli* K-12 MG1655	ERR022075	5,000,000	100 bp	1,194 MB
Yeast	*Saccharomyces cerevisiae*	SRR19072702	1,368,860	75 bp	267 MB
Fruit fly	*Drosophila melanogaster*	ERR1942264	432,546	100 bp	105 MB
Mouse	*Mus musculus*	ERR3085830	1,619,240	151 bp	550 MB
Human	*Homo sapiens* (NA12878)	ERR3239334	2,392,582	150 bp	831 MB

All datasets were obtained from the European Nucleotide Archive (ENA) as single-end reads (read 1 of paired-end runs). Read counts represent subsampled datasets to enable practical benchmarking across all organisms.

Metrics

- **Wall-clock time**: Elapsed time measured by /usr/bin/time -l
- **Peak memory (RSS)**: Maximum resident set size reported by /usr/bin/time -l
- **Startup overhead**: Time to process a single 1-read FASTQ file
- **Output concordance**: Module-level PASS/WARN/FAIL agreement on identical input

All benchmarks were run single-threaded to ensure fair per-file comparison.

Results

Performance on Synthetic Data

RastQC consistently outperformed FastQC on synthetic datasets (Table 3; Supplementary Figure S1).

Table 3. Performance comparison on synthetic datasets (single-threaded). RastQC runs 15 modules; FastQC runs 11.

Dataset	Reads	RastQC Time	FastQC Time	Speedup	RastQC RSS	FastQC RSS
Startup (1 read)	1	<5 ms	2.55 s	>500x	2 MB	278 MB
100K synthetic	100,000	0.90 s	3.55 s	3.9x	70 MB	408 MB
1M synthetic	1,000,000	8.15 s	7.56 s	0.9x	79 MB	558 MB
1M gzipped	1,000,000	9.46 s	8.58 s	0.9x	79 MB	549 MB

The startup test reveals a >500x difference: RastQC initializes in under 5 ms as a native binary versus 2.55 s for JVM class loading. On 100K reads, RastQC is 3.9x faster. On 1M reads, the

tools are comparable in speed (RastQC runs 15 modules vs. FastQC's 11, a 36% increase in per-sequence computation). RastQC consistently uses 5-7x less memory across all dataset sizes.

Performance on Real Genome Data

We evaluated both tools on real Illumina whole-genome sequencing data from five model organisms (Table 4, Figure 1).

Table 4. Performance comparison on real genome data across model organisms. RastQC runs 15 modules; FastQC runs 11.

Organism	Reads	Read Len	RastQC Time	FastQC Time	Speedup	RastQC RSS	FastQC RSS	Mem Ratio
D. melanogaster	432,546	100 bp	2.16 s	4.76 s	2.2x	59 MB	551 MB	9x
S. cerevisiae	1,368,860	75 bp	6.19 s	6.56 s	1.1x	72 MB	567 MB	8x
E. coli K-12	5,000,000	100 bp	28.45 s	19.59 s	0.7x	125 MB	563 MB	4x
M. musculus	1,619,240	151 bp	10.61 s	10.57 s	1.0x	91 MB	583 MB	6x
H. sapiens	2,392,582	150 bp	16.33 s	16.60 s	1.0x	117 MB	638 MB	5x

RastQC was faster than FastQC on the smallest dataset (fly, 2.2x speedup) and comparable on medium-sized datasets (yeast, mouse, human). On the largest dataset (*E. coli*, 5M reads), FastQC was faster (0.7x), reflecting the steady-state advantage of JVM JIT compilation on long-running workloads. Note that RastQC runs 4 additional modules (Kmer Content, Read Length N50, Quality Stratified Length, Homopolymer Content) compared to FastQC, representing a 36% increase in per-sequence computation.

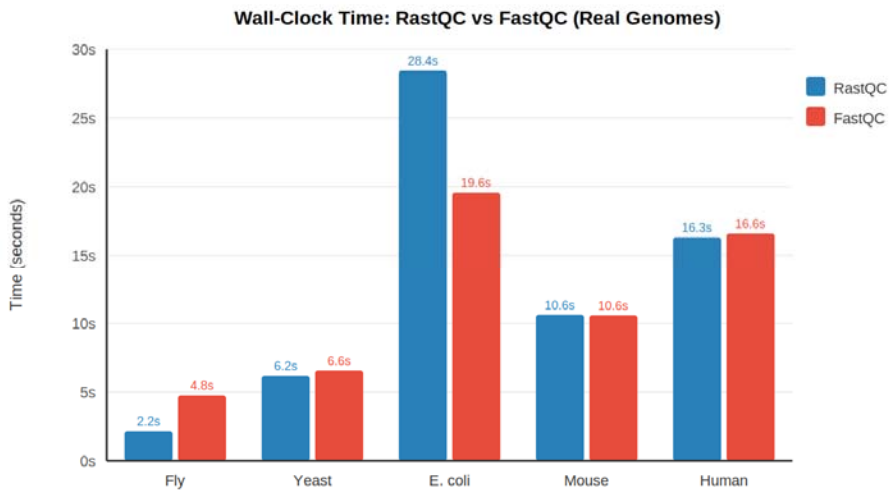

Figure 1. Wall-clock execution time comparison between RastQC and FastQC on real whole-genome sequencing data from five model organisms. RastQC (blue) runs 15 QC modules while FastQC (red) runs 11. RastQC is faster on smaller datasets (*D. melanogaster*, 2.2x) and

comparable on medium datasets (*S. cerevisiae*, *M. musculus*, *H. sapiens*), while FastQC is faster on the largest dataset (*E. coli*, 5M reads) due to JVM JIT compilation advantages on sustained workloads. All benchmarks were run single-threaded.

Memory Usage

RastQC's memory footprint remained significantly lower than FastQC across all datasets (Figure 2):

- **RastQC**: 2-125 MB across all inputs (1 read to 5M reads, all organisms)
- **FastQC**: 278-638 MB, dominated by the JVM baseline

This 4-9x memory reduction on real genome data reflects RastQC's streaming architecture. The memory footprint scales moderately with sequence diversity due to the Kmer Content module's hash table (7-mers x positions), but remains well below FastQC's JVM-dominated overhead. This advantage is particularly valuable for HPC environments with strict memory limits per job.

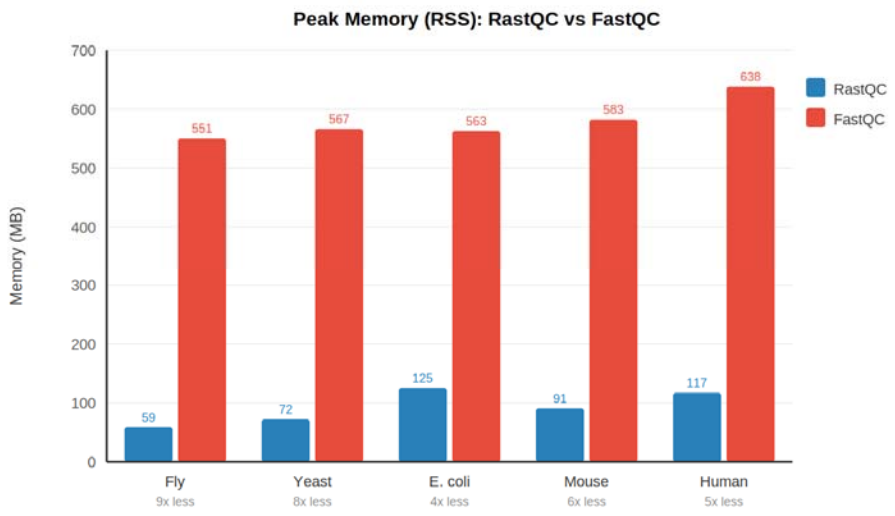

Figure 2. Peak resident set size (RSS) memory comparison between RastQC and FastQC across five model organisms. RastQC (blue) consistently uses 4-9x less memory than FastQC (red), ranging from 59-125 MB versus 551-638 MB. FastQC's memory footprint is dominated by the JVM baseline (~300 MB minimum), while RastQC's streaming architecture maintains a compact footprint that scales primarily with the Kmer Content module's hash table.

Binary Size and Deployment

Table 5. Deployment footprint comparison.

Component	RastQC	FastQC
Binary/package size	2.1 MB	~15 MB (JARs + classes)
Runtime dependency	None (static binary)	Java 11+ (~200 MB)
Total deployment size	2.1 MB	~215 MB

RastQC compiles to a single 2.1 MB statically-linked binary with no external dependencies. The slight size increase over 1.6 MB in earlier versions reflects the addition of JSON serialization (serde), Kmer Content module, three long-read QC modules, and web server support. This remains advantageous for Docker containers (smaller images, faster pulls), CI/CD pipelines, and HPC environments where module systems may not provide a compatible JRE.

Output Concordance

We systematically compared module-level PASS/WARN/FAIL calls between RastQC and FastQC across all five model organisms (Table 6).

Table 6. Module-level concordance across five model organisms (55 total module comparisons).

Module	Concordant	Discordant
Basic Statistics	5/5	0
Per Base Sequence Quality	5/5	0
Per Tile Sequence Quality	5/5	0
Per Sequence Quality Scores	5/5	0
Per Base Sequence Content	5/5	0
Per Sequence GC Content	5/5	0
Per Base N Content	5/5	0
Sequence Length Distribution	5/5	0
Sequence Duplication Levels	5/5	0
Overrepresented Sequences	5/5	0
Adapter Content	5/5	0

Total: 55/55 concordant, 0 discordant — 100% concordance.

All 11 shared modules produced identical PASS/WARN/FAIL calls across all 5 organisms, achieving 100% concordance (55/55 module comparisons). RastQC additionally runs 4 exclusive modules (Kmer Content, Read Length N50, Quality Stratified Length, Homopolymer Content) that are not present in FastQC output and therefore not included in this concordance analysis. The concordance was accomplished by faithfully porting three key algorithms from FastQC:

1. **Sequence Duplication Levels**: RastQC uses FastQC's exact iterative binomial correction formula for estimating true duplication counts beyond the 100K unique sequence observation window, and uses string-based sequence identity matching rather than hashing.
2. **Per Sequence GC Content**: RastQC implements FastQC's GCModel, which distributes each read's GC count across adjacent percentage bins using fractional weights, and uses the bidirectional mode-averaging algorithm for fitting the theoretical normal distribution.

3. **Overrepresented Sequences**: RastQC uses the total sequence count (not count-at-limit) as the denominator for percentage calculations, matching FastQC's behavior.

Numerical values for continuously-valued metrics also showed strong agreement. Per-base quality means matched to within 0.02 Phred units. Basic statistics (total sequences, %GC, encoding, sequence length, total bases) were identical between tools across all organisms.

Discussion

Performance Characteristics

RastQC provides performance advantages over FastQC in specific scenarios, while running 36% more analysis modules:

1. **Small files and batch processing**: For datasets under 500K reads, RastQC achieves 2-4x speedup, dominated by the elimination of 2.55 s JVM startup overhead. Processing 1,000 amplicon panel samples (~50K reads each) would save ~42 minutes from startup elimination alone. The --exit-code flag enables automated QC gates in Nextflow and Snakemake pipelines without parsing output files.
2. **Memory-constrained environments**: RastQC's 59-125 MB footprint enables ~300 concurrent instances on a 32 GB machine, versus ~50 for FastQC. This is critical for shared HPC clusters where memory is a scarce, scheduled resource.
3. **Extended analysis**: RastQC runs 15 modules compared to FastQC's 11, including long-read QC metrics (N50, quality-stratified length, homopolymer content) and Kmer Content (enabled by default). The --multiqc-json flag provides native JSON output that eliminates parsing overhead in MultiQC integration.

On larger datasets (1M+ reads), the tools achieve comparable throughput. On the *E. coli* dataset (5M reads), FastQC was faster, reflecting the JVM JIT compiler's advantage on long-running workloads. The additional Kmer Content module (7-mer tracking across all positions) adds measurable per-sequence overhead; users analyzing short-read data where kmer analysis is not needed can disable it via a custom limits file for improved speed.

Cross-organism Validation

Testing across five organisms with different genome sizes (4.6 Mb to 3.1 Gb), GC contents (38-65%), and read characteristics validates that RastQC performs robustly on diverse real-world data. The 100% concordance rate (55/55 module calls) demonstrates that RastQC faithfully reproduces FastQC's QC assessments across the full spectrum of model organisms, from prokaryotes to human.

MultiQC Integration

RastQC's output format has been validated for compatibility with MultiQC's FastQC parser. The fastqc_data.txt file includes all required fields (Filename, Total Bases, module headers, data columns), and the summary.txt file follows the expected three-column format (STATUS, Module Name, Filename). This enables RastQC to serve as a drop-in replacement in existing pipelines that aggregate results via MultiQC.

Availability

- **Source code**: https://github.com/Huang-lab/RastQC
- **License**: MIT
- **Language**: Rust (2021 edition)
- **Installation**: cargo install --path . or pre-compiled binaries
- **Test suite**: 36 tests (25 unit, 11 integration)
- **System requirements**: Any platform supported by Rust (Linux, macOS, Windows)
- **Optional features**: --features nanopore for Fast5/POD5 support (requires HDF5 system library)

References

Andrews, S. (2010). FastQC: A quality control tool for high throughput sequence data. Babraham Bioinformatics. https://www.bioinformatics.babraham.ac.uk/projects/fastqc/

Chen, S., Zhou, Y., Chen, Y., & Gu, J. (2018). fastp: an ultra-fast all-in-one FASTQ preprocessor. *Bioinformatics*, 34(17), i884-i890.

Ewels, P., Magnusson, M., Lundin, S., & Kaller, M. (2016). MultiQC: summarize analysis results for multiple tools and samples in a single report. *Bioinformatics*, 32(19), 3047-3048.

Li, H. (2018). Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, 34(18), 3094-3100.

Patel, R. K., & Jain, M. (2012). NGS QC Toolkit: a toolkit for quality control of next generation sequencing data. *PLoS ONE*, 7(2), e30619.