Reproducible scRNA-seq Analysis Pipeline (PBMC)

(Add a cool image)

Simo Inkala

February 2026

`https://github.com/inkasimo/scrnaseq-pbmc-workflow`

`inkasimo@gmail.com`

Contents

1 Overview

This project implements a reproducible, containerized single-cell RNA-seq (scRNA-seq) analysis pipeline using Snakemake, Docker, and a Python-based CLI wrapper. The workflow is demonstrated on publicly available 10x Genomics PBMC datasets (donors 1–4).

The primary objective of this repository is to showcase the design, implementation, and execution of a production-style bioinformatics workflow. Emphasis is placed on reproducibility, dependency management, modular execution, and transparent end-to-end data processing rather than on biological discovery.

Accordingly, this project is intended as a technical portfolio artifact. While biologically reasonable defaults are used throughout, the analysis is not designed to generate novel biological claims, but to demonstrate a complete, auditable, and reproducible scRNA-seq pipeline.

1.1 Execution environment

Hardware:	
CPU	Intel Core Ultra 9 185H (16 cores, 22 threads)
RAM	32 GB
Storage	NVMe SSD: 1 TB (OS) + NVMe SSD: 2 TB (data)

All computations were performed on a local workstation running Windows 11 with WSL2 and Docker Desktop. The pipeline was executed entirely inside Docker containers to ensure environment isolation and reproducibility across hosts.

Due to the memory requirements of STAR genome indexing, the default WSL2 configuration was insufficient and resulted in out-of-memory (OOM) failures. The WSL2 memory limit was therefore explicitly increased to 32 GB. A reference configuration file is provided at `docs/wslconfig.example` to document the required settings and facilitate reproducible execution on similar systems.

2 Repository Organization

The repository is organized to enforce a clear separation between execution logic,configuration, containerized dependencies, and generated data. This structure follows best practices for reproducible computational workflows and explicitly distinguishes between upstream engineering steps and downstream analytical stages.

- `containers/`
  Docker build context defining a fully self-contained execution environment. This includes all system dependencies, bioinformatics tools, and a fully specified R environment restored via `renv`. The archived image is intended to be the primary execution environment for reproducible runs.

- `workflow/`
  Snakemake workflow definition encoding the full directed acyclic graph (DAG).This includes rules for FASTQ acquisition or validation, quality control, optional read trimming, reference preparation, STARsolo alignment, and downstream analysis stages.

- `config/`
  User-editable configuration files (notably `config.yaml`) controlling dataset definitions, resource usage, algorithmic parameters, and execution toggles (e.g. downloading FASTQs vs. validating existing data, enabling trimming, network analysis settings).

- `resources/`
  Static, versioned resources bundled with the workflow to ensure reproducibility. This includes the 10x Genomics barcode whitelist and MSigDB gene set collections (Hallmark and C7), avoiding reliance on unstable external URLs.

- `data/` Input data and reference assets generated or downloaded at runtime. This directory contains raw FASTQs, reference genomes and annotations, STAR indices, and (if enabled) trimmed FASTQs.

- `results/` All pipeline outputs and logs. This includes QC reports, STARsolo outputs, downstream Seurat objects, differential expression results, network analyses, and detailed execution logs. Outputs are intentionally excluded from version control due to size.

- `docs/`
  Project documentation, including the user manual, this technical report, developer notes, and a small set of representative execution artifacts under `docs/example_outputs/` to demonstrate successful pipeline execution.

- `scripts/`
  R scripts implementing downstream analyses. These include Seurat-based QC and annotation, pseudobulk differential expression and equivalence testing, enrichment analysis, and co-expression network construction.

- `run_analysis.py`
  A Python-based CLI wrapper that orchestrates Snakemake execution inside Docker. The wrapper enforces section-based execution, provides explicit control over which parts of the pipeline are run, and prevents accidental full executions.

3 R Environment and Reproducibility Strategy

3.1 Overview

All R-based analyses in this workflow are executed inside a Docker container with a fully specified and reproducible R environment.

Dependency management is handled using `renv`, while containerization ensures isolation from host-specific configurations. This design minimizes environment drift and allows consistent re-execution across machines and over time.

3.2 R Version and Package Management

The container runtime is pinned to a fixed R base image (`rocker/r-ver:4.3.3`). R package dependencies are declared in a version-locked `renv.lock` file, and package resolution is pinned to a CRAN snapshot via the environment variable:
`RENV_CONFIG_REPOS_OVERRIDE=https://packagemanager.posit.co/cran/2024-01-01`.

The lockfile captures:

- The R version used (R 4.3.3)
- The CRAN snapshot date (`https://packagemanager.posit.co/cran/2024-01-01`)
- The exact versions and sources of all R packages required for the analysis

During the Docker image build, the full R environment is restored non-interactively using `renv::restore()`. The restored library is installed into a fixed path inside the container (`/opt/renv/library`) via `RENV_PATHS_LIBRARY`, and no R package installation or snapshotting occurs at runtime.

Bioconductor resolution is explicitly pinned during the image build (`Bioconductor 3.18`) to ensure consistent installation of Bioconductor packages.

3.3 Vendoring of Seurat

The core analysis depends on Seurat [1] and SeuratObject [2], which are fast-evolving packages and common sources of reproducibility issues due to upstream repository changes or availability.

To mitigate this risk, specific versions are vendored directly into the repository:

- `renv/vendor/seurat-v4.4.0.tar.gz`
- `renv/vendor/seurat-object-v4.1.4.tar.gz`

During Docker image build, the vendored sources are copied into the container (`COPY renv/vendor/ /work/renv/vendor/`) and installed by `renv::restore()` as local package sources recorded in `renv.lock`.
As a result:

- Docker builds do not rely on GitHub availability or branch state for these packages
- Seurat versions are decoupled from upstream CRAN/Bioconductor changes
- The exact Seurat source code used in the analysis is preserved with the project

4 Docker Integration

The Docker image is built from `rocker/r-ver:4.3.3`, providing a fixed R runtime on a Debian-compatible base. System-level dependencies required for R packages and external tools (e.g. STAR, FastQC) are installed explicitly during the image build.

The image also configures explicit `renv` paths to avoid host-dependent behavior:

- `RENV_PATHS_LIBRARY=/opt/renv/library`
- `RENV_PATHS_CACHE=/opt/renv/cache`
- `RENV_CONFIG_CACHE_SYMLINKS=FALSE`

This ensures that both the installed packages and the renv cache are fully contained within the image filesystem and are not affected by host-side caches or symlinks.

All R packages are restored during image build using `renv::restore()`. The Docker build does not modify the lockfile; dependency changes must be made explicitly by updating `renv.lock` and committing it to version control.

4.1 Docker Image Distribution

A pre-built Docker image corresponding to the archived workflow environment is published to GitHub Container Registry (GHCR). The versioned release can be pulled using:

```
docker pull ghcr.io/inkasimo/scrnaseq-pbmc-workflow:v1.0.1
```

For strict archival reproducibility, the digest-pinned image corresponding exactly to the environment used to generate the archived results can be pulled using:

```
docker pull ghcr.io/inkasimo/scrnaseq-pbmc-workflow@sha256:
80354b76e76405636c43e73902236e0399d26978a214227afbafa46fc0555bb8
```

Local image builds (`docker build`) are supported for development purposes but are not guaranteed to be bit-identical to the archived image due to potential upstream system library changes.

4.2 Separation of Concerns

- Host system: used primarily to edit code and commit changes. Dependency updates are captured by regenerating and committing `renv.lock`. In practice, lockfile updates are performed by launching an interactive R session inside the Docker container used for development, then committing the resulting lockfile on the host.
- Docker image (execution): consumes the lockfile in a read-only fashion during image build and runtime, providing a stable and reproducible execution environment.
- IDE tooling: automatic IDE-driven package installation is disabled to prevent contamination of the project dependency set.

Result

This setup ensures:

- A pinned and auditable R environment (R version, CRAN snapshot, and package sources)
- Long-term stability of critical dependencies (notably Seurat via vendoring)
- Clear provenance of all software components
- Safe integration with Snakemake and container-based execution

In summary, the combination of `renv`, vendored critical packages, and Docker provides a robust and auditable foundation for reproducible single-cell RNA-seq analysis.

4.3 Windows / WSL / Docker Desktop

This workflow runs inside Docker and bind-mounts the repository into the container. On Windows, Docker Desktop must have access to the repository path; otherwise the container will not be able to read files under `/work`.

If the following type of error is encountered:

```
Snakefile "workflow/Snakefile" not found
```

Docker Desktop file-sharing settings should be checked for the drive containing the repository. Bind-mount restrictions are common on managed or corporate machines; in such cases, running the workflow on Linux or macOS avoids these limitations.

5 Execution Model

The pipeline is orchestrated using Snakemake and executed inside Docker containers to ensure reproducibility and avoid host-specific environment drift.

The workflow is implemented as a `workflow/Snakefile` (the authoritative DAG and rule logic) and an optional Python wrapper (`run_analysis.py`) that provides a safer, section-based CLI for running the workflow inside the Docker image.

Conceptually, users select a "section" (e.g. QC, alignment, downstream), the wrapper maps that intent to explicit Snakemake targets (mostly `*.done` sentinel files), and Snakemake computes and executes the minimal set of rules required to produce those targets.

5.1 Snakemake workflow (`workflow/Snakefile`)

The Snakefile encodes the pipeline as a directed acyclic graph (DAG) of rules, where each rule declares explicit `input`, `output`, and execution logic (shell or Python `run` blocks).

A core design choice is the use of lightweight sentinel "done" files (e.g. `fastqc.done`, `starsolo.done`, `seurat_qc.done`) as rule outputs. These serve two purposes:

- They provide a stable, human-auditable completion marker for each stage without requiring Snakemake to hash or reason over large binary outputs.
- They make the workflow resilient to partial outputs: a rule is only considered complete when its sentinel file is created after all expected artifacts have been written successfully.

Sentinels are created atomically (write `.tmp` then move/replace), which reduces the chance of leaving a misleading "complete" marker after an interrupted job.

The workflow also supports two execution branches: `raw` (default) and `trimmed` (optional). Trimming is applied only to Read 2 and, if enabled, all downstream steps automatically route to the trimmed branch while preserving the on-disk STARsolo layout (legacy mapping of `untrimmed` → `raw`).

Configuration is driven by `config/config.yaml`, with CLI overrides supported for key toggles (e.g. download FASTQs, build STAR index, enable trimming). Rules are written to be explicit about required resources (threads, reference paths, gene sets, marker sets), and logs are captured under `results/logs/` for auditability.

5.2 Python wrapper (`run_analysis.py`)

While the Snakefile can be executed directly inside Docker, most users run the pipeline via `run_analysis.py`, which wraps Snakemake in a section-based CLI.

The wrapper's main job is to translate high-level user intent into explicit Snakemake targets. Each CLI "section" corresponds to a curated list of target files (primarily `*.done` sentinels and a small number of concrete outputs such as `multiqc_report.html`). Snakemake then builds exactly what is required to satisfy those targets, and nothing else.

This approach improves usability and safety:

- **Section-based execution:** users run a named stage (`qc`, `align`, `downstream`, …) instead of managing raw Snakemake targets.
- **Prevents accidental full runs:** the wrapper requires selecting a section, rather than implicitly executing the full `rule all`.
- **Mode toggles are explicit:** flags like `--trimmed` and `all_no_download` translate into controlled Snakemake config overrides (e.g. `trim_enabled=true`, `download_fastqs=false`).
- **Docker execution is standardized:** the wrapper always executes Snakemake inside the pinned container image, bind-mounting the repository and passing consistent CPU/thread settings.

- **Early failure checks:** the wrapper validates that Docker bind mounts can see `workflow/Snakefile`, avoiding confusing "Snakefile not found" failures caused by file-sharing restrictions on some Windows/WSL setups.

The wrapper also exposes convenience commands for discovery (`--list-sections`, `--list-donors`) and supports passing through advanced Snakemake options via `--extra` for power users.

5.3 Available wrapper sections

The full user manual documents all sections and examples; for quick reference, the wrapper exposes the following top-level sections:

- `download_data`
- `download_data_and_qc`
- `qc`
- `ref`
- `trim`
- `trim_and_qc`
- `align`
- `upstream`
- `upstream_no_download`
- `build_seurat_object_qc`
- `filter_and_normalize_seurat`
- `cluster_annotate_seurat`
- `deg_and_tost`
- `networks`
- `downstream`
- `all`
- `all_no_download`
- `unlock`

Internally, the wrapper maps these user-facing sections onto specific target sets (for example, `qc` expands to per-donor FastQC sentinels plus the aggregated MultiQC report). This design keeps the workflow modular and transparent while making execution approachable for users who do not want to interact directly with Snakemake DAG mechanics.

6 Data

Raw FASTQ files are obtained from publicly available 10x Genomics Chromium X Series datasets corresponding to the 5k Human PBMC 3′ Gene Expression (GEM-X v4) libraries for Donors 1–4. Each donor is provided as a separate FASTQ archive and downloaded directly from the 10x Genomics data portal.

The samples consist of peripheral blood mononuclear cells (PBMCs) isolated from healthy human donors (male, age 18–35), prepared by 10x Genomics and sourced from Cellular Technologies Limited. Cells were processed using the Chromium GEM-X Single Cell 3′ Reagent Kits v4 chemistry and sequenced on an Illumina NovaSeq 6000 platform at approximately 39,000 read pairs per cell.

Libraries were generated as paired-end, dual-indexed runs with the following configuration: 28 cycles for Read 1 (cell barcode + UMI), 10 cycles for i7 index, 10 cycles for i5 index, and 90 cycles for Read 2 (cDNA insert). Read 1 therefore encodes the 16 bp cell barcode and 12 bp UMI, while Read 2 contains the transcript-derived sequence used for alignment and quantification.

The datasets were originally processed by 10x Genomics using the `cellranger multi` pipeline, and automated cell type annotations were generated using the `human_pca_v1_beta` reference model. **However, in this project, only the raw FASTQ files are used; all alignment and downstream analyses are performed independently within the present workflow.**

Each donor dataset contains approximately 5,000–6,000 recovered cells and represents cryopreserved human PBMC suspensions. The data are licensed under the Creative Commons Attribution 4.0 International (CC BY 4.0) license and are publicly accessible through the 10x Genomics website.

Companion datasets include the individual 5k PBMC datasets for Donors 1–4 and a 20k multiplexed PBMC dataset combining all donors; however, this workflow analyzes the donors independently.

7 Quality Control

Raw FASTQ quality was assessed using FastQC v0.11.9 and summarized with MultiQC v1.33. QC results were used to evaluate the necessity of adapter trimming prior to alignment.

Across donors and lanes, base quality profiles are high and stable over the read length, and the aggregated MultiQC status table reports `PASS` for R2 *Adapter Content*, indicating no evidence of pervasive adapter contamination in the cDNA read.

The primary flagged item is *Overrepresented sequences*, most notably in Donor 4 R2 files. The top sequences occur at approximately FastQC's reporting threshold and are consistent with known short-oligo / library-structure artifacts (e.g., template-switch / SMARTer-like carryover) rather than systematic adapter read-through affecting a substantial fraction of reads.

For 10x Genomics 3′ Gene Expression data, explicit trimming is typically unnecessary and can be counterproductive: standard single-cell pipelines (e.g., Cell Ranger and STARsolo) rely on preserving the expected read structure, and modest 3′ artifacts are generally handled via soft-clipping during alignment without compromising UMI counting. Therefore, trimming was omitted for the main analysis and is only enabled if strong adapter signal is observed in QC.

The workflow supports an optional trimming branch via a dedicated wrapper section (and corresponding Snakemake rules), but this branch was not activated for the reported results given the QC outcome (see Section **Read Trimming (cutadapt)**).

All QC reports are written to `results/qc/` (FastQC per-donor outputs and aggregated MultiQC reports), with execution logs stored under `results/logs/qc/`.

8 Read trimming (cutadapt)

The workflow supports an optional trimming branch implemented with `cutadapt` and exposed through dedicated wrapper sections (e.g. `trim`, `trim_and_qc`) and the `-trimmed` flag. When activated, all downstream steps (QC, alignment, and analysis) resolve against trimmed FASTQs and write into a separate `trimmed/` results branch.

Trimming is applied to Read 2 (cDNA) only; Read 1 (cell barcode + UMI) is never sequence-modified. Read pairs are filtered jointly using `-pair-filter=any`, so pairs are discarded if R2 fails the minimum-length threshold.

8.1 Configuration

Trimming parameters are defined in `config/config.yaml` and passed into the Snakemake rule via `TRIM_R2_*` variables.

```
trim:
  enabled: false
  adapter_r2: AGATCGGAAGAG
  q_r2: 20
  minlen_r2: 20
```

In the reported trimming run, `cutadapt v4.9` was executed with `-A AGATCGGAAGAG`, `-Q 0,20`, and `-minimum-length 0:60`, i.e. an effective `minlen_r2` of 60.

8.2 Implementation details

Outputs are written lane-by-lane to temporary files and atomically published on success; the `trim.done` sentinel is created only after all lanes complete. This prevents partially written FASTQs from being consumed downstream.

8.3 Observed impact

Across all donors, the proportion of read pairs discarded as too short after trimming was low and consistent (approximately 0.3–0.4%). This indicates minimal short-insert filtering and no evidence of pervasive adapter read-through.

Given the strong baseline QC metrics and the negligible trimming impact, downstream analyses in this report use the default untrimmed branch. The trimming pathway remains available but was not required for these datasets.

9 Alignment and Quantification

9.1 Choice of aligner: STARsolo vs Cell Ranger

Alignment and UMI-aware quantification were performed using `STARsolo` (STAR v2.7.11b) rather than Cell Ranger. The primary motivation is transparency and reproducibility: STARsolo exposes all alignment and barcode-handling parameters explicitly, integrates directly into Snakemake, and runs inside the pinned Docker environment.

This avoids reliance on opaque defaults or vendor-controlled pipelines while preserving compatibility with standard 10x data structures (gene–cell matrices equivalent to Cell Ranger output).

9.2 Reference genome and annotation

The reference was built locally from the following components:

- GRCh38 primary assembly
- GENCODE v45 gene annotation
- STAR v2.7.11b
- 10x GEM-X 3′ v4 barcode whitelist (bundled in `resources/barcodes/`)

The STAR index was generated inside the container using `-sjdbOverhang 89` (R2 length 90 bp), and is not distributed with the repository due to size. Index generation requires approximately 25–30 GB RAM.

9.3 Read structure and chemistry assumptions

Libraries correspond to 10x Chromium GEM-X 3′ v4 chemistry. Read structure assumptions were made explicit in the STARsolo invocation:

- R1: 16 bp cell barcode + 12 bp UMI (28 bp total)
- R2: cDNA (90 bp)
- `-soloType CB_UMI_Simple`
- `-soloCBlen 16`
- `-soloUMIlen 12`
- `-soloBarcodeReadLength 28`
- `-soloBarcodeMate 2`

Whitelist filtering was enforced via `-soloCBwhitelist`.

9.4 Counting and outputs

Gene-level UMI counting was performed internally by STARsolo using `-soloFeatures Gene`. Both raw and filtered gene–cell matrices were generated. No external counting tools (e.g. featureCounts) were used.

9.5 BAM output

BAM output was disabled via `-outSAMtype None` to reduce disk usage and memory overhead. STARsolo gene–cell matrices are complete without BAM files. BAM generation can be enabled if required for downstream inspection.

9.6 Execution environment

Alignment was executed inside Docker under WSL2. Memory was configured to allow STAR index generation and alignment:

```
[wsl2]
memory=32GB
processors=8
swap=16GB
```

No additional host-specific tuning was required.

9.7 Alignment summary statistics

Alignment metrics were consistent across donors. Unique mapping rates ranged from 68.8% to 72.4%, multi-mapping from 19.3% to 21.7%, and reads unmapped due to short length from 7.6% to 9.4%. Mismatch rates were stable at approximately 0.21–0.23%. No chimeric reads were reported.

Donor	Input Reads (M)	Unique (%)	Multi (%)	Unmapped Short (%)	Mismatch (%)
Donor1	227.1	71.89	20.18	7.83	0.23
Donor2	207.0	72.41	19.33	8.16	0.22
Donor3	185.7	72.42	19.84	7.64	0.21
Donor4	207.8	68.77	21.74	9.37	0.22

Table 1: STARsolo alignment summary statistics (raw branch).

Mapping performance is consistent with high-quality 10x 3′ scRNA-seq libraries; donor4 shows a modestly lower unique mapping rate and higher short-read fraction but remains within expected bounds for PBMC datasets.

10 Downstream Analysis

10.1 Overview

Downstream analyses were performed on donor-specific expression datasets derived from STARsolo filtered gene–cell matrices. Seurat objects were used for quality control, normalization, clustering, and marker-based annotation. Pseudobulk differential expression and co-expression networks were derived from the resulting QC-filtered and annotated data representations.

The workflow proceeds in five stages: (i) Seurat object construction from STARsolo outputs, (ii) donor-specific quality control and normalization, (iii) clustering and marker-based cell-type annotation, (iv) donor-aware pseudobulk differential expression with equivalence testing, and (v) cell-type–specific co-expression network construction with cross-donor consensus integration. All analyses were executed inside the containerized R environment to ensure full reproducibility.

All Seurat-based analyses were performed using `Seurat v4.4.0` and `SeuratObject v4.1.4` as specified in the `renv.lock` file embedded in the Docker image.

11 Methods

11.1 Seurat object construction

Filtered STARsolo gene–barcode matrices were imported into Seurat. Feature names were set to gene symbols; symbol-less Ensembl placeholder rows were removed. Duplicate symbols were collapsed by summing counts to ensure a unique feature space per object.

11.2 Quality control filtering

Quality control was performed independently per donor using data-adaptive thresholds derived from empirical distributions of QC metrics.

Filtering rules:

- `percent.mt` $\leq \min(q_{90}, 25)$
- `nFeature_RNA` $\geq q_{05}$ and $\leq \min(q_{99}, 6000)$
- `nCount_RNA` $\geq q_{05}$ and $\leq q_{99}$
- `percent.hb` ≤ 1

For each donor, thresholds and filtering summaries were written to disk. Filtering decisions were therefore fully auditable.

11.3 Normalization and feature selection

QC-filtered objects were normalized using Seurat's `LogNormalize` method (library-size scaling followed by log-transformation; scale factor = 10,000). Highly variable genes (HVGs; $n = 2000$) were identified using the VST method.

Scaling and regression of `percent.mt` were applied to HVGs for dimensionality reduction only. Scaled values were not used for differential expression or network inference.

11.4 Clustering and marker-based annotation

Dimensionality reduction was performed using PCA (first 30 components), followed by nearest-neighbor graph construction and Louvain clustering (resolution = 0.3). UMAP embeddings were generated using a fixed random seed.

Cell type annotation was performed using predefined marker gene sets. For each marker set, module scores were computed using `AddModuleScore`. Each cell was assigned the label corresponding to its highest module score. Cluster-level majority labels were also computed for reporting and downstream grouping. PBMC marker genes represented in table 2.

11.5 Pseudobulk differential expression and equivalence testing

Differential expression was performed at the donor level using a pseudobulk strategy.

Pseudobulk construction: For each donor and coarse immune group (table 3.), raw UMI counts were summed across cells (minimum 50 cells per donor $\times$ group). Donor was treated as the experimental unit.

Cell type	**Marker genes**
T-cells	TRAC, CD3D, CD3E, LTB
CD4 T-cells	IL7R, CCR7, LTB, MALAT1
CD8 T-cells	CD8A, CD8B, NKG7, GZMK
NK-cells	NKG7, GNLY, PRF1, FCGR3A, XCL1
B-cells	MS4A1, CD79A, CD74, HLA-DRA
Plasma cells	MZB1, XBP1, JCHAIN
CD14 Monocytes	LYZ, S100A8, S100A9, LGALS3, CTSS
FCGR3A Monocytes	LYZ, FCGR3A, MS4A7, LGALS3
Dendritic cells	FCER1A, CST3, CLEC10A
Platelets	PPBP, PF4, NRGN

Table 2: Canonical PBMC marker genes used for manual cell-type annotation. Marker sets were curated from established PBMC single-cell RNA-seq references and workflows, including the 10x Genomics PBMC 3k/10k datasets [5], dendritic cell characterization by [6], and Seurat-based PBMC analyses [7, 8].

DEG group	**Constituent cell types**
T-like	T-cells, CD4 T-cells, CD8 T-cells, NK-cells
B-like	B-cells, Plasma cells
Mono-like	CD14 Monocytes, FCGR3A Monocytes

Table 3: Cell-type groupings used for pseudobulk differential expression analysis.

Statistical model and differential expression: Pseudobulk differential expression was performed using `DESeq2` v1.42.1 (Bioconductor 3.18). Counts were modeled using the design:

$$\sim \texttt{donor} + \texttt{group},$$

thereby controlling for donor-specific baseline effects while testing cell-type group contrasts. Wald tests were used for coefficient inference, and $\log_2$ fold changes correspond to the estimated group effect. P-values were adjusted for multiple testing using the Benjamini–Hochberg procedure.

Contrasts:

Contrast	**Comparison**
C1	T-like vs B-like
C2	T-like vs Mono-like
C3	B-like vs Mono-like

Table 4: Pseudobulk differential expression contrasts evaluated using DESeq2.

Marker genes: Strict marker genes were defined as those satisfying

$$\texttt{padj} < 10^{-10} \quad \text{and} \quad |\log_2 \text{FC}| > 3.$$

Equivalence testing (TOST): Two one-sided tests were performed on DESeq2 $\log_2$ fold-change estimates using a predefined margin $\delta = 0.75$. Genes were considered conserved if:

$$\texttt{padj\_equiv} < 0.05 \quad \text{and} \quad |\log_2 FC| < \delta$$

This distinguishes true similarity from non-significance.

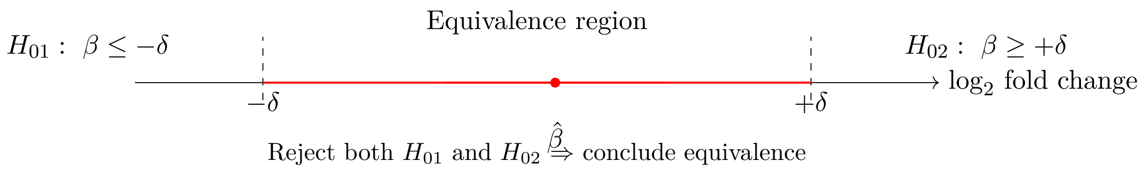
Equivalence region
$H_{01}: \beta \leq -\delta$
$H_{02}: \beta \geq +\delta$
$\log_2$ fold change
$-\delta$
$+\delta$
$\hat{\beta}$
Reject both H_{01} and H_{02} $\Rightarrow$ conclude equivalence

Figure 1: Two one-sided tests (TOST) for equivalence of $\log_2$ fold change within $[-\delta, +\delta]$.

11.6 Pathway enrichment analysis

Pathway analyses were performed using MSigDB Hallmark and C7 collections.

- GSEA (fgsea) on ranked gene lists (ranked by $\log_2$FC)
- ORA (clusterProfiler) on marker and conserved gene sets

Cross-contrast intersections and union gene sets were also tested for enrichment.

11.7 Cell-type–specific co-expression networks

Gene co-expression networks were constructed separately for each donor within predefined cell-type subsets `(CD4 T cells, B cells, and CD14 monocytes)`. For each donor and cell type, pairwise gene–gene associations were computed and converted into an undirected network.

Cross-donor integration was performed using a consensus approach: an edge was retained if observed in at least two independent donor-specific networks (consensus edge count ≥ 2). Given the limited number of biological replicates ($n = 4$), this requirement prioritizes reproducible co-expression relationships and reduces donor-specific noise.

Metacell aggregation: Cells were randomly pooled into metacells (size = 20) to stabilize correlation estimates.

Gene filtering: Genes detected (expression > 0) in at least 5% of metacells (not cells) within a donor $\times$ cell-type subset were retained.

Edge definition: Spearman correlation was computed across metacells. Edges were retained using a sparsification strategy designed to reduce unstable or donor-specific correlations:

- Top-k neighbors per gene ($k = 50$)

- $|\rho| \geq 0.25$
- Positive correlations only

The top-k constraint enforces local sparsity and limits hub inflation. The correlation threshold suppresses weak, noise-driven associations, which is particularly important given the limited donor count. Restricting to positive correlations improves interpretability in small-sample settings.

Per-donor graphs were reduced to their largest connected component to avoid fragmentation artifacts.

Cross-donor consensus and module detection: Donor-specific edge lists were merged into a consensus graph. For each gene pair, we computed:

- `support` = number of donors in which the edge was observed
- `median_cor` = median correlation across supporting donors

Edges were retained if observed in at least two donors with consistent sign. The final consensus network was restricted to its largest connected component.

Leiden clustering was performed on the consensus graph using `median_cor` as the edge weight.

Consensus edge weighting: In addition to `median_cor`, a stability-adjusted weight was computed for ranking and visualization:

$$\texttt{weight\_consensus} = \texttt{median\_cor} \times \texttt{support\_frac}, \qquad \texttt{support\_frac} = \frac{\texttt{support}}{\#\text{ donors used}}$$

This support-scaled weight was not used for clustering, but is provided for edge prioritization and visualization.

Functional enrichment (ORA): Module gene sets were tested for over-representation using MSigDB Hallmark (H) and Immunologic Signatures (C7) collections. Enrichment was performed against the network gene universe and summarized in tabular and dot-plot form.

11.8 Network exports for Gephi

For each consensus network, Gephi-compatible tables were generated:

Edge list (`edges.csv`) including `median_cor`, `support`, `support_frac`, and `weight_consensus`. Node table (`nodes.csv`) including module assignment, degree, and betweenness centrality. Module membership table (`modules.csv`).

Nodes are additionally annotated with canonical marker membership flags and DEG-derived indicators (strict marker and conserved sets), enabling direct biological interpretation within the graph representation.

12 Results

12.1 Quality control and filtering outcomes

12.2 Clustering structure and annotation consistency

12.3 Differential expression between coarse immune groups

12.4 Pathway enrichment patterns

12.5 Consensus co-expression network properties

12.6 Module-level functional enrichment

A Figures and Tables Placeholders

B Example table (booktabs)

Table 5: Example table caption.

Column 1	Column 2	Column 3
A	B	C

References

[1] Hao Y, Stuart T, Kowalski MH, et al. *Dictionary learning for integrative, multimodal and scalable single-cell analysis.* Nature Biotechnology. 2023. doi:10.1038/s41587-023-01767-y.

[2] Satija R, Butler A, Hoffman P, Stuart T. *SeuratObject: Data Structures for Single Cell Data.* R package version 4.1.4. 2023.

[3] Liberzon A, Birger C, Thorvaldsdóttir H, Ghandi M, Mesirov JP, Tamayo P. *The Molecular Signatures Database (MSigDB) hallmark gene set collection.* Cell Systems. 2015.

[4] Godec J, Tan Y, Liberzon A, Tamayo P, Bhattacharya S, Butte AJ, Mesirov JP, Haining WN. *Compendium of Immune Signatures Identifies Conserved and Species-Specific Biology in Response to Inflammation.* Immunity. 2016.

[5] Zheng GX, Terry JM, Belgrader P, et al. *Massively parallel digital transcriptional profiling of single cells.* Cell. 2017;171(5):1202–1214.e15.

[6] Villani AC, Satija R, Reynolds G, et al. *Single-cell RNA-seq reveals new types of human blood dendritic cells, monocytes, and progenitors.* Science. 2017;356(6335).

[7] Stuart T, Butler A, Hoffman P, et al. *Comprehensive Integration of Single-Cell Data.* Cell. 2019;177(7):1888–1902.e21.

[8] Hao Y, Hao S, Andersen-Nissen E, et al. *Integrated analysis of multimodal single-cell data.* Nature Biotechnology. 2021;39: 856–865.

[9] Aran D, Looney AP, Liu L, et al. *Reference-based analysis of lung single-cell sequencing reveals a transitional profibrotic macrophage.* Nature Immunology. 2019;20(2):163–172.