

Politecnico di Torino

Politecnico di Torino

Master in Computer Engineering - Software

A.a. 2025/2026

Graduation Session September 2026

pyGAGAM: A faithful, scalable, and scverse-compatible reimplementation of GAGAM for scATAC-seq

Relatori:

Roberta Bardini
Stefano Di Carlo
Alessandro Savino
Lorenzo Martini

Candidati:

Mehmed Yusa Erguven

Abstract

Single-cell ATAC sequencing (scATAC-seq) measures chromatin accessibility, which regions of the genome are open and available for regulation, one cell at a time. It reports this signal over open regions of the genome called peaks, not over genes. To connect it to the rest of the single-cell ecosystem, the peak signal is commonly summarised into a gene activity matrix, a per-gene estimate of regulatory activity in each cell. GAGAM (Genomic Annotated Gene Activity Matrix) builds such a matrix by linking peaks to genes through functional annotations, namely promoters, exons, and co-accessible distal enhancers. The single-cell field has largely converged on scverse, a community of interoperable Python tools built on the AnnData data model. However, GAGAM's original implementation is an initial research contribution: a collection of R and Python scripts that enables reproducibility but does not yet meet the usability and interoperability requirements of a FAIR digital asset. It is unpackaged, hard to reuse across datasets, and sits outside the scverse ecosystem.

This thesis presents pyGAGAM, a pure-Python, scverse-compatible library that reimplements the latest GAGAM methodology (v1.2). It operates natively on AnnData, follows the scanpy-style `pp`/`tl`/`pl` API, and ships as an installable, tested, and documented package published to a public index. Its monolithic construction routine is made modular, split into small, replaceable units for the promoter, exon, and enhancer contributions, the pillars of the GAGAM methodology, which recombine into the complete matrix and make it straightforward to add new annotation tracks, such as those in the latest ENCODE release. pyGAGAM also reimplements GRAIGH, a complementary method that projects accessibility onto GeneHancer regulatory elements, adding a gene-regulation view alongside GAGAM's gene-activity matrix.

A direct comparison with the original GAGAM results on the study's five public datasets shows agreement to within about 0.03 to 0.04 on the Residual Average Gini Index (RAGI), an established measure of how well a gene-activity matrix captures cell-type-specific signal, on every dataset and under both of the study's evaluation protocols. It reproduces the reference gene-activity matrix on identical inputs with a correlation of about 0.99. On the V2.0.0 PBMC dataset (4,623 cells, 135,317 peaks) a complete run from raw data takes about 3.8 minutes, against about 128 minutes for the reference R/Python pipeline, more than 30 times faster; with optional multicore and GPU acceleration the pipeline processes a 717,000-cell atlas (CATlas) dataset at a comparable 150,000-peak density in about 47 minutes, and a 250,000-cell subset against the full 926,000-cCRE ENCODE registry in about 83 minutes. The reimplemented GRAIGH companion reproduces the GeneHancer

registry exactly, and clustering its element matrix reaches an adjusted mutual information of 0.80 on PBMC.

By reproducing GAGAM faithfully and faster, adding the GRAIGH gene-regulation view, and packaging both as an interoperable, well-tested library, py-GAGAM makes these methods readily usable in modern single-cell analysis.

Summary

The single-cell assay for Transposase-Accessible Chromatin sequencing (scATAC-seq) measures the accessibility of chromatin one cell at a time, showing which regulatory regions of the genome are open and thus which genes a cell is able to express. It can be combined with single-cell RNA sequencing (scRNA-seq); however, scATAC-seq reports its signal over genomic *peaks* rather than over genes, as scRNA-seq does. It is therefore a common and crucial need to summarise that signal as a *gene activity matrix* (GAM): a cell-by-gene estimate of regulatory activity that the rest of the single-cell software ecosystem can use directly. GAGAM (Genomic Annotated Gene Activity Matrix) builds such a matrix by linking peaks to genes using functional annotations, namely promoters, exons, and co-accessible distal enhancers, which ties gene accessibility to gene expression. However, the original implementation is a set of R and Python research scripts that are not packaged, not trivial to reuse for different datasets, tied to fixed file paths, dependent on deprecated interfaces, and unable to interoperate with the modern Python single-cell ecosystem, scverse. It was therefore necessary to create a new implementation in Python that is packaged, reusable, and interoperable with the scverse ecosystem.

This thesis presents **pyGAGAM**, a pure-Python, scverse-compatible library that reimplements and consolidates the latest version of the GAGAM methodology, GAGAM v1.2. The work has three objectives: to port the GAGAM methodology to Python faithfully; to integrate it into the scverse ecosystem so that it works alongside established tools such as scanpy, muon, and snapatac2; and to improve its performance in terms of speed and scalability. pyGAGAM follows the scverse `pp/tl/pl` API convention and operates throughout on the AnnData data model. Its main design decision is to break the original monolithic construction routine into small replaceable contribution units, one each for the promoter, exon, and enhancer signals, which can be used and replaced independently and which recombine to reproduce the monolithic routine's result exactly. The paper's modelling choices are exposed as documented parameters, and the package is unit- and integration-tested, documented, and distributed as an scverse-compatible library. The same foundation also reimplements GRAIGH, a complementary method that projects the accessibility onto GeneHancer regulatory elements, adding a gene-regulation

view alongside GAGAM's gene-activity one.

pyGAGAM is validated against GAGAM v1.2 on the five public datasets used in the original study. When the two are scored like for like, comparing the paper's own released results against pyGAGAM's under one consistent quality metric (RAGI) and a single clustering procedure, they agree to within about 0.03 to 0.04 RAGI on every dataset and on both evaluation axes reported in the original work: the matrix judged under its own clustering, and under a shared external one. A complete run from raw data is roughly **30 times faster** than the reference R/Python pipeline. On the third objective, the library adds optional multicore and GPU acceleration, which lets it scale to datasets larger than those the original GAGAM was tested on: the pipeline has been run on a 717,000-cell atlas (CATlas) at several peak counts, from realistic sizes up to the full peak set (926,000 peaks). The main contribution is the library itself: a reusable, scverse-compatible tool that is also substantially faster and more scalable than the original.

Acknowledgements

I would like to thank my supervisor, Professor Roberta Bardini, and my co-supervisors, Professors Stefano Di Carlo and Alessandro Savino, and Dr. Lorenzo Martini. I am particularly grateful to Lorenzo for his guidance and helpful discussions throughout this work.

Pursuing this master's degree also gave me the opportunity to meet and work with wonderful people at Tropico Security. I am especially grateful to Carlo De Micheli, Alessandro Raponi, and Antonio Costagliola for their support, encouragement, and understanding as I balanced my studies and work.

I am deeply grateful to my mother, Hatice Ergüven, for her love, her belief in me, and her support throughout my education. Her encouragement has meant more to me than I can express in these few lines. I also thank my sister, Zeynep Ergüven Peker, and my brother-in-law, Hasan Peker, for their love and support. Knowing that I could count on my family, even from a distance, made these years easier.

Finally, I would like to thank my wife, Gülşah Ay Ergüven, for her love, patience, and encouragement throughout this journey. Thank you for being beside me through the difficult moments and for sharing in the small successes along the way. Completing my master's as we begin a new chapter of our lives together makes this milestone especially meaningful, and I am grateful to share it with you.

Table of Contents

List of Figures VIII

1 Introduction 1
- 1.1 Objectives . . . 2
- 1.2 Contributions . . . 2
- 1.3 Structure of the thesis . . . 3

2 Background 4
- 2.1 From chromatin accessibility to single-cell ATAC-seq . . . 4
- 2.2 Gene activity matrices . . . 5
  - 2.2.1 Existing gene-activity methods . . . 7
  - 2.2.2 Regulatory elements and their annotation . . . 7
  - 2.2.3 Co-accessibility: linking distal peaks to promoters . . . 9
  - 2.2.4 The GAGAM method . . . 10
  - 2.2.5 Evaluating a gene activity matrix: RAGI . . . 11
- 2.3 GeneHancer and the regulatory-element view . . . 12
- 2.4 The scverse single-cell ecosystem . . . 13
- 2.5 The reference implementations . . . 14

3 Methodology 15
- 3.1 Requirements and design goals . . . 15
- 3.2 Architecture and scverse integration . . . 17
- 3.3 Reimplementing the GAGAM pipeline in Python . . . 19
  - 3.3.1 Preprocessing and peak annotation . . . 19
  - 3.3.2 Co-accessibility . . . 22
  - 3.3.3 Modular gene-activity construction . . . 23
  - 3.3.4 Faithfulness and the parameter surface . . . 23
- 3.4 GRAIGH: a GeneHancer-element matrix . . . 26
- 3.5 Evaluation tooling . . . 26
- 3.6 Engineering for scale . . . 29

4 Results and Discussion 31
4.1 Evaluation setup . . . 31
4.2 Faithful reproduction of GAGAM . . . 33
4.3 Engineering for scale . . . 37
4.4 Packaging, testing, and availability . . . 41
4.5 GRAIGH results . . . 42
4.6 Extensibility case studies . . . 43
4.7 Discussion and limitations . . . 48

5 Conclusion and Future Works 50
5.1 Conclusion . . . 50
5.2 Future work . . . 51

A Reproducibility 53
A.1 Datasets . . . 53
A.2 Reference annotations . . . 53
A.3 Software environment . . . 54
A.4 Test coverage . . . 55

Bibliography 57

List of Figures

2.1 From chromatin accessibility to a peak-by-cell matrix 6
2.2 Three ways to link peaks to a gene 8
2.3 Building the gene activity score 10
2.4 How RAGI reads a matrix 11
2.5 Building the GeneHancer-element matrix 13

3.1 From reference scripts to a single library 17
3.2 The pyGAGAM pipeline 20
3.3 The evaluation harness 27
3.4 The clustering recipe 28

4.1 RAGI on the self-clustering axis 35
4.2 RAGI on the common-clustering axis 36
4.3 Reference pipeline versus pyGAGAM (V2.0.0) 38
4.4 From-raw runtime across the five datasets 39
4.5 Scaling on CATlas 40
4.6 Gene-activity connections for MS4A1 44
4.7 Adding v4 distal classes: controlled RAGI 47

Chapter 1

Introduction

Single-cell sequencing has made it possible to study a tissue one cell at a time. Two assays are especially common: single-cell RNA sequencing (scRNA-seq), which measures gene expression, and single-cell ATAC sequencing (scATAC-seq), which measures chromatin accessibility, that is, how open each region of the genome is and therefore how available it is for regulation. The two give complementary views of a cell: what it is expressing, and how that expression is regulated. A practical obstacle is that scATAC-seq reports its signal over genomic peaks rather than over genes, whereas interpreting that signal biologically, and integrating it with gene-expression data, is done in terms of genes. A gene activity matrix (GAM), a cell-by-gene estimate of regulatory activity built from the accessibility of nearby peaks, is the usual way to close that gap. The next chapter covers the biological background in full.

GAGAM (Genomic Annotated Gene Activity Matrix) [1] is a method for building such a matrix from explicit biological annotations rather than from a single fixed window around each gene, combining promoter, exon, and co-accessible distal-enhancer signal. It was developed in the research group in which this thesis was carried out, and the original study reported it to be competitive with established gene-activity methods. Its reference implementation, however, is a set of R and Python research scripts that depend on a heavy toolchain (Signac, Monocle3, and Cicero), and reusing that methodology in a new analysis takes considerable extra work. The code is not installable as a library, it is tied to fixed file paths, it is run interactively, and it sits outside the modern Python single-cell ecosystem.

That ecosystem has largely converged on scverse [2], a set of community libraries built around the AnnData data structure, with scanpy as its central analysis toolkit. A method that is not available there is, in practice, hard for the community to adopt, to combine with other tools, or to maintain. The work in this thesis is therefore primarily an engineering effort: to turn the existing GAGAM method into a coherent, maintainable, and interoperable Python library that stays faithful

to the original results, rather than to design a new algorithm. The same library additionally reimplements GRAIGH, a companion method from the group that represents accessibility through GeneHancer regulatory elements, giving a gene-regulation view alongside GAGAM's gene-activity one.

1.1 Objectives

The thesis has three objectives.

- **O1: implement the GAGAM methodology in Python.** Reproduce the full GAGAM pipeline as Python code, from peak annotation through co-accessibility to the final gene activity matrix, and confirm that it stays faithful to the original implementation on the datasets used in the source study.
- **O2: integrate pyGAGAM into the scverse ecosystem.** Build the library so that it follows scverse conventions: operate on the AnnData data model, expose a scanpy-style `pp/tl/pl` API, interoperate with neighbouring tools, and meet the community's expectations for packaging, documentation, and testing.
- **O3: improve performance and scalability.** Make the pipeline substantially faster than the reference R implementation and able to run on atlas-scale datasets, through an efficient implementation and optional multicore and GPU acceleration.

The emphasis throughout is operational. The contribution is the software, meaning its design, its faithfulness to the original, and its fit with the ecosystem, not a new gene-activity model.

1.2 Contributions

Concretely, the thesis delivers the following.

- **pyGAGAM**, a pure-Python, scverse-compatible library that implements GAGAM v1.2 on AnnData and follows the `pp/tl/pl` API convention.
- A **modular decomposition** of the original monolithic construction routine into small, independently usable contribution units (promoter, exon, and enhancer), which can be inspected, replaced, or recombined, and which reproduce the one-shot result exactly.

- A **faithful validation** against the original GAGAM on its five published datasets and both reported evaluation axes: scored like for like, the two agree to within about 0.03 to 0.04 RAGI, while a complete run from raw data is roughly 30 times faster than the reference R/Python pipeline.
- **Packaging, documentation, and tests** to scverse community standards: an installable, version-controlled package published to a public package index (TestPyPI), a documentation site with a worked tutorial, and unit and integration tests run under continuous integration.
- optional **multicore and GPU acceleration** of the heavier steps, which leaves the result effectively unchanged and lets the pipeline scale to datasets of hundreds of thousands of cells, far beyond the sizes used in the original study.

Two further contributions extend the same foundation beyond these objectives:

- **GRAIGH**, a complementary gene-regulation representation: a cells-by-GeneHancer-element matrix built on the same AnnData / `pp-tl-pl` foundation, with a shared feature-specificity metric, and validated on real data (the loader reproduces the GeneHancer v5.15 registry, and clustering the element matrix reaches AMI 0.80 on PBMC).
- three short **case studies** built on the library's composable API (a gene-centric co-accessibility visualization and two investigations of regulatory-element biology), evidence that the design is usable for real downstream analysis.

1.3 Structure of the thesis

The remainder of the thesis is organised as follows.

- The **Background** chapter gives the biological and software context: scATAC-seq and gene activity matrices, the GAGAM method and the RAGI metric, the GeneHancer view behind GRAIGH, and the scverse ecosystem.
- The **Methodology** chapter sets out how the library is built: its architecture and scverse integration, the translation of GAGAM from R to Python, and the construction of the GRAIGH GeneHancer-element matrix.
- The **Results and Discussion** chapter reports the faithful reproduction of GAGAM, the behaviour at scale, the GRAIGH results, the extensibility case studies, and the limitations of the work.
- The **Conclusion and Future Works** chapter summarises the outcome and outlines directions for future work.

Chapter 2

Background

This chapter lays out the background that the rest of the thesis builds on. It starts from the biology of chromatin accessibility and single-cell ATAC-seq, works up to the GAGAM method and how a gene activity matrix is evaluated, and then covers the GeneHancer view used by GRAIGH, the scverse software ecosystem, and the original implementations that this work replaces.

2.1 From chromatin accessibility to single-cell ATAC-seq

Most of a cell's DNA is wound around histone proteins and packed into *chromatin*. In that packed state it is largely out of reach. A protein that needs to read or regulate a gene can only get to the stretches of DNA that are left exposed, and those exposed stretches are the ones we call *accessible*. They tend to be the regions a cell is actively using to control its genes, such as promoters and enhancers. A map of where the genome is open, then, doubles as a rough map of which genes the cell is set up to express.

ATAC-seq (Assay for Transposase-Accessible Chromatin using sequencing) turns that openness into a measurable signal [3]. The work is done by an enzyme, Tn5, preloaded with short known DNA tags called *adapters*. Wherever Tn5 can reach the DNA it both cuts it and inserts an adapter at the cut, a one-step reaction called tagmentation. The enzyme is bulky, so it only reaches DNA that is exposed. Almost all of its cuts therefore fall in the open regions.

Only fragments that picked up adapters can be sequenced, so every read traces back to a spot that was open. Map the reads onto the genome and the open regions stand out: they gather many reads, while closed regions gather few or none. The read count along the genome is what we call the *accessibility signal*.

Early ATAC-seq was done in *bulk*. The enzyme is applied to a whole sample

of cells at once, all the reads are pooled, and what comes out is one accessibility signal averaged across the population. The individual cell types can no longer be told apart. Single-cell ATAC-seq (scATAC-seq) keeps each read tied to the cell it came from, giving one accessibility profile per cell rather than one for the whole population.

Two designs made this work, both introduced in 2015. One isolates the cells physically, putting each in its own reaction so its reads never mix with the rest [4]. The other, combinatorial indexing, skips isolation entirely: cells pass through several rounds of barcoding until each carries a unique combination of labels, and those labels later sort the pooled reads back to their cell of origin [5]. The assay is now in wide use.

Resolving cells one at a time is what makes scATAC-seq powerful. It can separate the cell types in a mixed tissue, and line accessibility up against single-cell RNA sequencing (scRNA-seq), which reads out gene expression in the same kind of sample. Together the two assays describe a cell from complementary angles: which regulatory regions are open, and which genes it is actively transcribing.

Single cells are sparse. Each one yields few fragments, so at most positions it shows nothing, and where it does the count is usually one or two; the matrix is close to binary. To give every cell the same features, the reads from all cells are aggregated and the reproducibly accessible intervals are called as *peaks*. The experiment then reduces to a sparse *peak-by-cell matrix*, the accessibility analogue of the gene-by-cell matrix from scRNA-seq, only far larger: hundreds of thousands of peaks in place of tens of thousands of genes. Figure 2.1 traces these steps, from open chromatin through to the peak-by-cell matrix.

The peak feature space brings its own problem. Peaks are genomic coordinates, not genes. Some steps cope with this directly; clustering, for instance, is usually run straight on the peak matrix. The trouble comes when the clusters have to be interpreted, because interpretation happens in gene terms. A cluster is labelled as a cell type from its marker genes, accessibility is compared and integrated with scRNA-seq gene by gene, and most existing single-cell tools and reference annotations expect genes rather than peaks. Bridging to all of that means turning the per-peak signal into a per-gene one, which is where the next section begins.

2.2 Gene activity matrices

A gene activity matrix (GAM) is a gene-by-cell matrix whose entries estimate how active each gene is in each cell, built from the accessibility of the peaks linked to that gene. It is the bridge out of the peak feature space: it turns the peak-by-cell matrix, which gene-level tools cannot use directly and which cannot be compared with scRNA-seq, into a gene-by-cell matrix that both can.

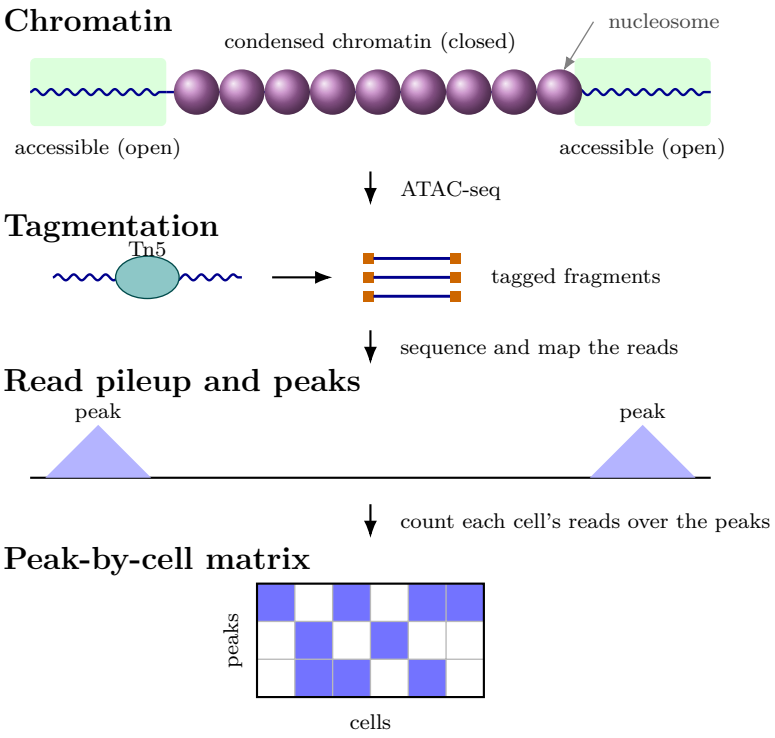


Figure 2.1: From chromatin accessibility to a peak-by-cell matrix. In condensed (closed) chromatin the DNA is wound around nucleosomes and packed tightly; in accessible (open) regions it is exposed. The Tn5 transposase of ATAC-seq reaches only the exposed DNA, where it cuts and inserts sequencing adapters in one step (tagmentation), leaving short adapter-flanked fragments. Sequencing those fragments and mapping them back to the genome gives a read pileup that is high over the accessible regions, and the enriched intervals are called *peaks*. In the single-cell assay every read is traced to its cell of origin, so the experiment becomes a sparse peak-by-cell matrix (columns are cells, rows are peaks), with dark entries marking the accessible peaks in each cell.

Building a GAM requires two decisions: which peaks belong to a gene, and how their accessibility is combined into a single value. Existing methods settle the first with genomic distance, or with data-driven co-accessibility between peaks. GAGAM, the method this thesis reimplements, brings in a further source of information that the others leave out, a biological annotation of what each peak actually is. It still uses co-accessibility for one step, to link enhancers to their promoters, but the annotation is what sets it apart. The next subsection reviews the existing methods, and the rest of the section builds up the approach GAGAM takes.

2.2.1 Existing gene-activity methods

Most established methods score a gene from the peaks in a window around it. Signac sums the accessibility over the gene body together with a short region upstream of the transcription start site (TSS), the promoter [6]. ArchR instead weights each peak by its distance to the gene, so nearby peaks count for more and the contribution fades with distance, cut off at neighbouring genes [7]. snapatac2 offers a comparable proximity-based gene matrix inside the Python ecosystem [8]. These models are fast and simple, and for many purposes good enough, so they are the default in most toolkits.

A second family goes further and links distant peaks from the data itself. Cicero scans across cells for pairs of peaks whose accessibility rises and falls together; a distal peak that tracks a gene's promoter this way is taken as a likely regulator, and its accessibility is folded into the gene's score [9]. This picks up enhancers that a fixed window would never reach.

What neither family uses is what a peak actually is. A distance rule cannot tell a promoter from an intron (a non-coding stretch inside a gene), and a co-accessibility link joins peaks by correlation alone, without asking whether the distal partner looks like an enhancer. GAGAM adds that missing ingredient. It labels each peak by its regulatory role, builds the gene score from those roles, and calls on co-accessibility only where it genuinely needs it, to link enhancers to promoters. Figure 2.2 places the three approaches side by side on a single gene locus, one stretch of the genome.

2.2.2 Regulatory elements and their annotation

GAGAM's annotations rest on three kinds of genomic element. The first is the *promoter*, the region around a gene's *transcription start site* where transcription begins; when a promoter is accessible, the gene is either being transcribed or ready to be. Enhancers work at a distance. An *enhancer* boosts the transcription of one or more target genes and can sit anywhere from a few to hundreds of kilobases away, looping round to touch the promoter of the gene it controls. *Exons*, finally,

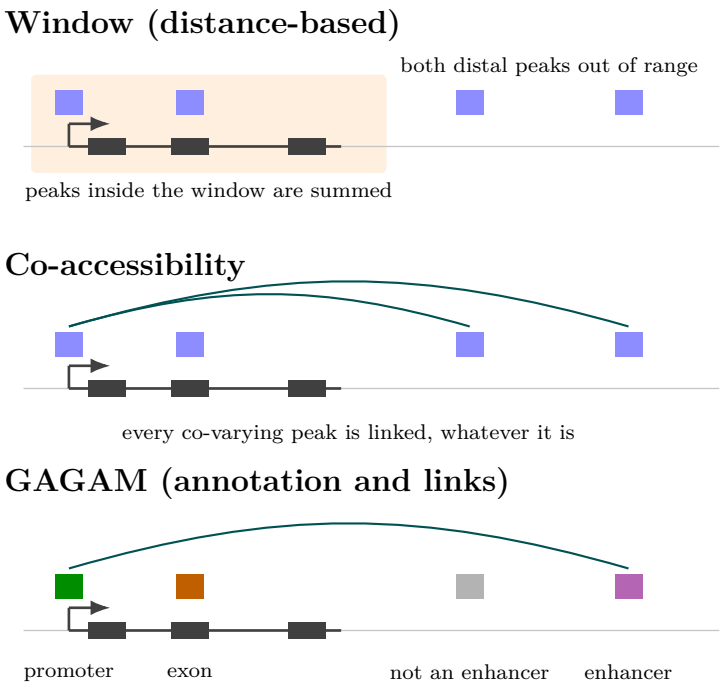


Figure 2.2: Three ways of deciding which peaks contribute to a gene's activity, shown on the same locus (gene body with exons in black, the transcription start site arrowed, accessibility peaks as coloured blocks). A window method sums the peaks within a fixed distance of the gene and misses both distal peaks. A co-accessibility method links every distal peak whose accessibility co-varies with the promoter, which reaches the true enhancer but also pulls in an accessible site that is not one. GAGAM first labels each peak by its regulatory role, so it keeps the promoter and exon signal, links only the peak annotated as an enhancer, and leaves out the co-varying non-enhancer.

are the parts of a gene retained in the mature transcript; a peak that falls over an exon adds evidence of activity from within the gene body itself.

Rather than work these roles out from scratch for every dataset, current analyses label peaks against a reference catalogue of *candidate cis-regulatory elements* (cCREs). The one GAGAM uses comes from the ENCODE project's SCREEN registry, which sorts close to a million human elements (and several hundred thousand mouse ones) by their chromatin signatures into promoter-like (`PLS`), proximal and distal enhancer-like (`pELS` and `dELS`), and classes bound by CTCF, a protein that helps organise chromatin into regulatory domains [10]. Labelling a peak is then a matter of overlapping it with the registry and taking the class it lands on.

The class matters, because GAGAM does not treat all peaks alike. Among the distal elements, it is `dELS` that carries the active-enhancer histone mark H3K27ac, so those are the peaks GAGAM counts as enhancers. Newer releases of the registry add further accessible-but-non-enhancer classes, `CA-CTCF` and `CA-TF`, bound by CTCF or by other transcription factors; whether these carry any extra gene-activity signal is one of the questions taken up in a later case study.

2.2.3 Co-accessibility: linking distal peaks to promoters

Annotation can tell us that a peak is a distal enhancer, but not which gene that enhancer acts on. The missing link comes from *co-accessibility*. Across many cells, pairs of peaks whose accessibility rises and falls together tend to be co-regulated and in physical contact, so a distal enhancer that is consistently co-accessible with a gene's promoter is a good candidate for regulating that gene.

Cicero is the tool GAGAM uses to measure this [9]. Single cells are too sparse to correlate directly, so Cicero first pools similar cells into small groups, called metacells, and works with their aggregated profiles. Within a genomic window of about 500 kilobases it then estimates the co-accessibility of every pair of peaks with a regularised model, a graphical lasso, that shrinks weak and spurious correlations towards zero and penalises pairs that lie far apart. The output is a peak-by-peak co-accessibility graph, in which a strong edge between a gene's promoter and a distal peak is read as that peak regulating the gene.

There is a recently published Python package that is CIRCE [11], reimplements Cicero's co-accessibility method and is built on the same data structures as the rest of the Python single-cell ecosystem (the scverse project, covered later in this chapter), so the whole step stays in Python. With peaks labelled by role and enhancers linked to their promoters, GAGAM has everything it needs to assemble the matrix, which the next subsection describes.

2.2.4 The GAGAM method

GAGAM [1, 12] turns the annotated peaks into a gene score by combining three contributions, one for each kind of regulatory evidence introduced above. The promoter is the anchor. GAGAM starts from the principle that a gene is active in a cell only where its promoter is open, so a promoter peak at the gene's TSS is what first marks the gene as expressed.

Beyond the promoter, two further contributions refine the score. The *enhancer* contribution brings in distal enhancer peaks, the `dELS` class, that are co-accessible with the gene's promoter, kept only when the link is strong enough and within about 300 kilobases (a tighter cap than Cicero's 500-kilobase scan window), and weighted by how strongly the two co-vary. The *exon* contribution adds peaks that fall over the gene's exons, the parts retained in the mature transcript, weighted so that peaks closer to the TSS count for more and the contribution fades with distance.

The three contributions are summed into a single value for each gene in each cell, as Figure 2.3 shows. Using all three gives **GAGAM1**; dropping the exon contribution, so that only the promoter and enhancer remain, gives **GAGAM2**. The original study finds GAGAM1 generally the stronger of the two, which suggests the exon signal carries real information. A final normalisation, borrowed from Cicero [9], brings the three contributions and the per-cell totals onto a comparable scale. The result is the gene-by-cell gene activity matrix that the rest of the analysis uses, and the final subsection turns to how a matrix like it is judged.

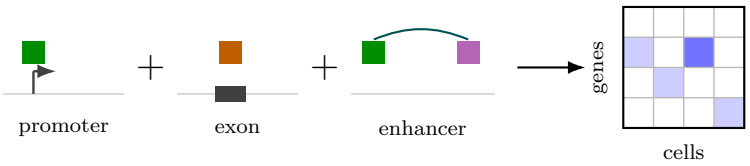


Figure 2.3: How GAGAM builds a gene's activity from its annotated peaks. Three contributions are combined: the **promoter** peak at the transcription start site, which anchors the score; the **exon** peaks over the gene's exonic regions, weighted so that peaks nearer the TSS count for more; and the co-accessible distal **enhancer** peaks, weighted by co-accessibility strength. Their sum is the gene's activity in one cell, a single entry of the gene-by-cell matrix (dark cell); repeating over every gene and cell fills the matrix. GAGAM1 uses all three contributions, while GAGAM2 leaves out the exon term. A final Cicero-style normalisation places the entries on a comparable scale.

2.2.5 Evaluating a gene activity matrix: RAGI

A gene activity matrix is only an estimate, so its quality has to be judged by how well it recovers biology that is already known. The GAGAM study uses the *Residual Average Gini Index* (RAGI) for this [13]. It works from two fixed lists of genes. *Marker* genes are specific to particular cell types, so a good matrix should make them look concentrated in a few clusters. *Housekeeping* genes are expressed in more or less every cell, so they should look much the same across clusters, whatever the matrix.

For each gene, its activity is averaged within every cluster, and the Gini index measures how uneven those per-cluster averages are, high when the activity piles up in a few clusters and low when it is shared evenly. RAGI is then the mean Gini of the marker genes minus the mean Gini of the housekeeping genes. The subtraction is the point of the metric. Even a gene that is not cell-type-specific picks up some Gini just from uneven cluster sizes and noise, so the housekeeping genes set that baseline, and taking it away leaves the genuine cell-type-specific signal, the *residual* that gives the metric its name. A higher RAGI therefore means the marker genes stand out well above the housekeeping baseline, which is what a useful matrix should achieve (Figure 2.4).

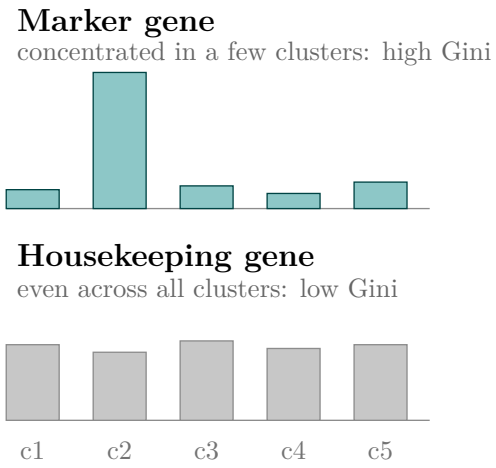


Figure 2.4: How RAGI reads a gene activity matrix. For a marker gene (top), a good matrix concentrates the activity in the few clusters of the cell type it marks, an uneven profile with a high Gini index; for a housekeeping gene (bottom), the activity is spread evenly across clusters, giving a low Gini. RAGI is the mean Gini of the marker genes minus the mean Gini of the housekeeping genes, so the greater the difference in unevenness between the two, the higher the score. Bar height is the average activity in each cluster; the values shown are illustrative.

RAGI still depends on which clustering of the cells it is measured against, and

the GAGAM study reports it two ways that differ exactly there. The first uses each matrix's own clustering. The cells are grouped using the matrix under test, and RAGI is measured on that grouping, which judges the matrix end to end but blurs two things, because a change in RAGI can come from a better clustering rather than from better values.

The second uses one grouping that is the same for every matrix and comes from outside them, either ground-truth cell-type labels, where a dataset has them, or a clustering of the raw accessibility data. Because the grouping is external and shared, a difference in RAGI can only come from the gene-activity values, which makes it the cleaner test of whether pyGAGAM reproduces GAGAM. The study reports these two views as its Figure 7 and Figure 8. Where ground-truth labels exist, the clustering each matrix produces is also compared against them directly with standard agreement metrics, the adjusted Rand index (ARI) and adjusted mutual information (AMI), defined where they are used later.

The gene-activity account is now complete; the next section turns to a different view of the same accessibility data, the regulatory-element view behind the companion method GRAIGH.

2.3 GeneHancer and the regulatory-element view

Both methods in this thesis start from the same peak accessibility; they differ in what they summarise it onto. GAGAM collapses the accessibility onto genes, giving one activity score per gene. GRAIGH, the companion method, keeps the regulatory elements themselves as the unit instead [14]. It builds on *GeneHancer*, a genome-wide catalogue of human regulatory elements, chiefly enhancers and promoters, each linked to the genes it is predicted to regulate, assembled as part of the GeneCards resource [15]. Where the candidate cis-regulatory registry from earlier in this chapter labels a peak only by its regulatory type, GeneHancer goes further and ties each element to specific target genes.

GRAIGH uses that catalogue to do the summarising. Each accessibility peak is matched to the GeneHancer element whose genomic region it falls inside, and each element then takes on the accessibility of the peaks it contains. The result is a matrix of GeneHancer elements by cells, where each entry is how accessible a given element is in a given cell. The features are now the regulatory elements, each carrying its own list of target genes, rather than peaks or genes. Figure 2.5 shows the construction.

Keeping the elements as the features gives a gene-regulation view alongside GAGAM's gene-activity one. A per-gene activity score reports the net result, how active a gene is, but hides which regulatory element produced it; GRAIGH instead shows the regulatory elements directly, so cells can be compared and clustered by

which elements are open, and each element's target genes provide the gene-level interpretation. GRAIGH is the additional, companion method that this thesis reimplements as part of pyGAGAM, so it shares the library's scverse-compatible form; later chapters cover how that is done and how it performs.

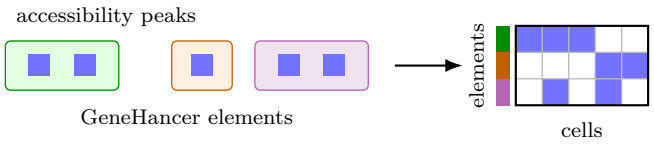


Figure 2.5: How GRAIGH re-expresses the accessibility over regulatory elements. Each accessibility peak (blue) is matched to the GeneHancer element (coloured box) whose region it falls inside. Every element then takes on the accessibility of the peaks it contains, giving a matrix of GeneHancer elements by cells (rows coloured to match the elements) in place of GAGAM's gene-by-cell matrix. Each element is also linked to a target gene, which makes the result interpretable.

2.4 The scverse single-cell ecosystem

Both methods are only useful if analysts can fit them into the tools they already use, so the second objective of this thesis is interoperability, which in practice means following the conventions of *scverse*, the community ecosystem that Python single-cell analysis has largely converged on [2]. At its centre is the `AnnData` data model [16], a container that holds a cell-by-feature matrix together with aligned tables of per-cell and per-feature annotations, low-dimensional representations, graphs, and unstructured results. Because every tool reads and writes the same structure, results accumulate on one object as it passes through a pipeline. `MuData` extends this to multi-modal data, such as a 10x Multiome experiment that measures ATAC and RNA in the same cells [17]. Because `AnnData` is cell-by-feature, pyGAGAM's contract is a cell-by-peak input and a cell-by-gene output, the transposes of the peak-by-cell and gene-by-cell matrices used above.

scanpy is the analysis toolkit at the centre of that ecosystem [18], and it sets an API convention that scverse packages are expected to follow. Functions are grouped into `pp` for preprocessing, `tl` for the tools that compute results, and `pl` for plotting; they act in place on an `AnnData` object or return a copy, and write their outputs into its standard slots. Following this convention is what makes a new package feel native to users of the ecosystem, and pyGAGAM adopts it throughout.

scverse also sets expectations for what a maintainable ecosystem package looks like, beyond producing correct numbers: a documented and tested public API, continuous integration, and standard packaging, codified in the shared

`cookiecutter-scverse` project template. Meeting these standards is part of what the second objective requires, and later chapters describe how pyGAGAM does so.

Computing gene activity from scATAC-seq is not new, and the tools met earlier do a form of it: Signac and ArchR in R [6, 7], and snapatac2 in Python [8]. pyGAGAM is meant to work alongside these rather than replace them, since any of them can supply the cell-by-peak matrix it reads, directly as an `AnnData` for snapatac2 or after a routine conversion for the R tools. What none of them provided was a pure-Python, scverse-compatible implementation of the annotation-based GAGAM approach, and that is the gap this thesis fills. The reference implementations it starts from are the subject of the final section.

2.5 The reference implementations

GAGAM was first released as a set of research scripts [19] rather than as a software package [1, 12]. The pipeline spans two languages and several heavyweight dependencies: peak preprocessing and clustering through Signac [6] and Monocle3 [20], co-accessibility through Cicero [9], the gene-activity construction itself in R, and supporting Python scripts for peak classification and for computing RAGI.

GRAIGH is much the same: a set of Python scripts [21] from the same group, wrapped in a Snakemake and Singularity workflow, that build the GeneHancer-element matrix and evaluate it [14]. It already runs in Python, but as a study pipeline rather than as a reusable library.

None of this is unusual for the analyses behind a research paper, and it served that purpose well, but it makes the methods hard to reuse. The code is not installable as a library; it is tied to fixed file paths and directory layouts; it is run interactively rather than through a stable programmatic interface; it mixes R and Python tooling, some of it pinned to older, lightly maintained versions; and it sits outside the Python single-cell ecosystem that most current users work in. Turning this body of method into a coherent, maintainable, scverse-compatible Python library, one that stays faithful to the original results while being modular, configurable, tested, and interoperable, is the engineering problem the rest of the thesis addresses. The next chapter sets out the requirements and the architecture that shape the solution.

Chapter 3

Methodology

The previous chapter set out the method this thesis reimplements and the software ecosystem it targets. This chapter turns to the work itself. It begins with the requirements that shaped the library and the design they led to, then follows the GAGAM pipeline from raw peaks to the finished gene activity matrix as it was translated from R to Python, describes the companion GRAIGH construction, sets out the tooling used to judge the result against the original, and closes with the engineering that lets the pipeline run at scale.

3.1 Requirements and design goals

GAGAM and GRAIGH already worked when this project began, but as research scripts rather than as software others could readily pick up. The goal here is not a new gene-activity model; it is a faithful and reusable one. The method had to be turned into a library that produces the same results as the original, fits the tools people already use, and can be maintained and built on. That goal resolves into a handful of requirements, and together they shaped every decision in the rest of this chapter.

- **Faithfulness.** The library must reproduce the original GAGAM results on the datasets used in the source study. Correctness here is measured against the reference implementation and the published figures, not simply asserted.
- **Interoperability.** It must operate on the `AnnData` data model and follow scverse conventions [2], to compose with neighbouring tools in the ecosystem such as scanpy, muon, and snapatac2.
- **Modularity.** The single construction routine of the original had to be broken into small, independent steps that can be inspected, replaced, or recombined, and that reproduce the one-shot result exactly.

- **Configurability.** Every modelling choice the paper makes should be exposed as a documented parameter, so the same code can run either as a faithful reproduction or as a present-day analysis.

- **Software quality.** The library must be packaged, documented, and tested to the standards the ecosystem expects of a maintained package.

- **Performance and scalability.** The pipeline should be fast enough for interactive use and able to run on datasets substantially larger than those the original was tested on.

These requirements line up with the three objectives set out in the Introduction: faithfulness and modularity serve the first, interoperability and software quality the second, and performance and scalability the third. Configurability cuts across all three. It earns its own treatment because two of the requirements pull against each other in practice: faithfulness asks the library to behave exactly like a method built several years ago, on the reference data and annotations of that time, while interoperability and everyday use call for current data and defaults. pyGAGAM resolves this by separating the two. It ships modern defaults, and it exposes every paper-era choice as a parameter, so a faithful reproduction and a present-day analysis are the same code run with different arguments.

Modularity is the central design decision, and it is worth saying why. The reference GAGAM pipeline is spread over a number of R and Python scripts, but the gene-activity construction itself sits in a single R script that runs from top to bottom. That script reads the preprocessed data and labelled peaks and computes the promoter, exon, and enhancer contributions one after another before combining and normalising them. It is not a function that can be called with arguments; it is a flat sequence that also folds clustering, evaluation, and plotting into the same file. The three contributions are written as consecutive blocks, but they are tied together through shared intermediate variables, so none of them can be run, inspected, or replaced on its own. pyGAGAM takes this apart into separate, self-contained steps, each an independently usable function, so a user can stop at any intermediate result or swap in a different rule for one step without disturbing the others. For the common case it also provides a single function, `build_gam`, that runs the whole construction in one call; the step-by-step and one-shot routes are guaranteed to produce an identical matrix, checked by an automated test, so the decomposition changes nothing about the result. The following sections describe the architecture and the pipeline built around this decision.

3.2 Architecture and scverse integration

pyGAGAM follows scverse conventions throughout, so the library is organised around the `AnnData` data model [16]. It takes a cell-by-peak `AnnData` and returns a cell-by-gene one, and in between each step reads from and writes to the standard slots of that object instead of passing bespoke structures from one function to the next. Peak coordinates and cCRE labels sit in the per-feature table (`var`), the co-accessibility graph in the per-feature pairwise slot (`varp`), per-cell information in `obs`, and the settings of each run in the unstructured slot (`uns`). For a multi-modal experiment such as a 10x Multiome, which measures ATAC and RNA in the same cells, the accessibility half is a cell-by-peak `AnnData` of exactly this kind; in the ecosystem it sits alongside the expression half inside a `MuData` [17] container, and that accessibility `AnnData` is what pyGAGAM takes as its input. Because this is the object the rest of the ecosystem already works with, results accumulate on it as it passes through the pipeline, and any scverse tool can read what pyGAGAM leaves behind. Figure 3.1 sets the resulting library against the reference implementation it replaces.

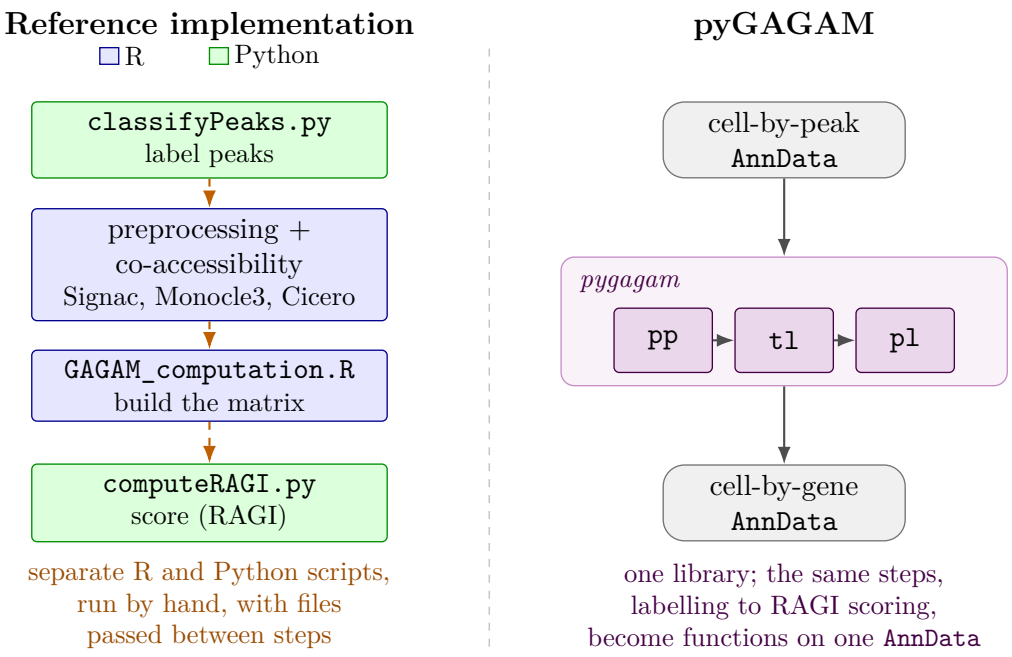


Figure 3.1: The engineering transformation behind pyGAGAM: the same pipeline as a set of separate R and Python scripts that pass files between steps by hand (left), and as a single installable library in which one `AnnData` object flows through the `pp`, `tl`, and `pl` namespaces (right).

The functions follow the scanpy convention [18] of three namespaces: `pp` for preprocessing, `tl` for the tools that compute results, and `pl` for plotting. The GAGAM pipeline maps onto them directly, as Table 3.1 shows, so someone who already uses scanpy finds pyGAGAM laid out where they expect it to be.

Table 3.1: The pyGAGAM pipeline mapped onto the scverse `pp`/`tl`/`pl` API. Each stage is a small set of functions that read from and write to the shared `AnnData` object.

Stage	API	Main functions
Load genome and label peaks	`pp`	`load_genome`, `classify_peaks`, `liftover_peaks`
Co-accessibility	`tl`	`coaccessibility`
Gene activity matrix	`tl`	`identify_promoter_peaks`, `promoter_contribution`, `exon_contribution`, `enhancer_contribution`, `combine_contributions`, `build_gam`
GeneHancer matrix (GRAIGH)	`pp`, `tl`	`load_genehancer`, `build_gh_matrix`
Clustering and scoring	`tl`	`cluster_cells`, `ragi`
Visualisation	`pl`	`plot_gene_connections`, `peaks_per_gh`, `specificity_heatmap`

The steps communicate only through the `AnnData` object, which does more than keep them separable: it makes them interoperable with the wider ecosystem. Any cell-by-peak `AnnData` serves as input, whatever produced it, so the upstream preprocessing can be snapatac2, muon, or a raw 10x matrix rather than a loader the library forces on the user. The enhancer step reads its co-accessibility graph from the standard `varp` slot and does not care which tool wrote it, so a graph from another method can stand in for CIRCE's. And where a step has to borrow the shared working slots that tools like scanpy use for embeddings and cell-neighbour graphs, as the co-accessibility computation does, it saves whatever those slots already hold and puts it back when it finishes, so a call into pyGAGAM never quietly overwrites a result the user had computed earlier. Behaving this way is what lets pyGAGAM sit inside a larger analysis rather than take it over.

Configurability lets the library ship modern defaults while still reproducing the original study exactly. Every modelling choice the paper makes is a documented parameter with a default, and the defaults are the current ones: the latest release of the ENCODE cCRE registry, for example, in place of the one available when GAGAM was published. Passing the paper-era values as arguments restores the

exact configuration behind the original study, down to the registry version and the distance cut-offs, without touching the code.

Finally, the library is built to the standards the ecosystem expects of a maintained package:

- it follows the scverse community's `cookiecutterscverse` project template and is built with a standard Python packaging backend;
- it is published to a public package index (TestPyPI);
- its public API is documented on a site with a tutorial and worked guides;
- its individual steps are covered by unit and integration tests; and
- those tests run under continuous integration on every change.

With the shape of the library established, the next section follows the GAGAM pipeline itself as it was translated from R and Python scripts into this form.

3.3 Reimplementing the GAGAM pipeline in Python

This section follows the GAGAM pipeline from raw peaks to the finished gene activity matrix and describes how each stage was translated from the reference scripts into pyGAGAM. The translation is a port, not a redesign: at every step the goal is the output the original produces, reached through the composable, `AnnData`-based functions of the previous section. The stages are taken in pipeline order: preprocessing and peak annotation first, then co-accessibility, then the modular construction of the matrix, and finally the reproduction decisions that keep the result faithful to the published method. Figure 3.2 maps the whole pipeline and the data that passes between its stages; the clustering and scoring at the end are covered later, in Section 3.5.

3.3.1 Preprocessing and peak annotation

Before a gene activity matrix can be built, three things have to be in place on the input `AnnData`: the peak coordinates, a gene annotation, and a regulatory label for each peak. Preprocessing assembles them, in pyGAGAM through a small set of `pp` functions that all annotate the same object. The original spread the work across a Python script and several R scripts that passed intermediate files between them.

scATAC data arrives with its peaks named in one of two ways. Cell Ranger writes the interval into the feature name as `chr1:100-200`, whereas the GAGAM

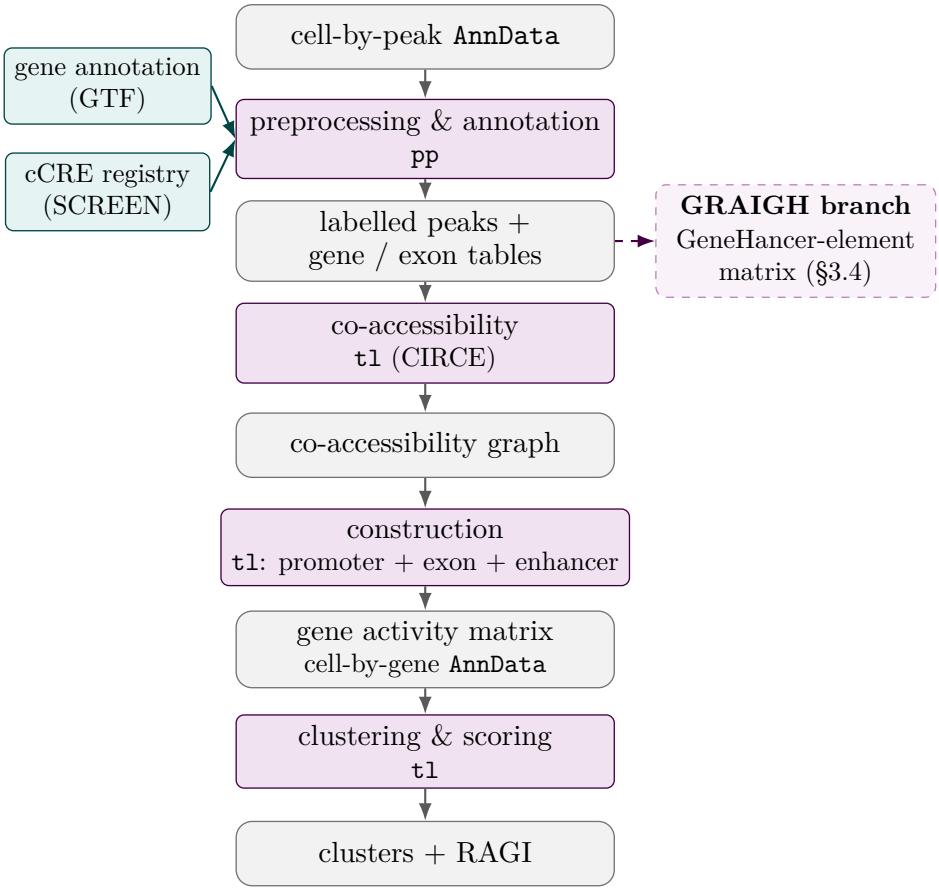


Figure 3.2: The pyGAGAM pipeline, from the cell-by-peak `AnnData` to the gene activity matrix and its evaluation. Grey nodes are data states carried on the `AnnData` object; violet nodes are the `pp` and `tl` stages that transform them. Preprocessing, co-accessibility, and construction are the subject of this section; clustering and scoring are covered in Section 3.5. Preprocessing draws in a gene annotation and the cCRE registry, and construction combines the promoter, exon, and enhancer contributions. The same labelled peaks also feed the companion GRAIGH matrix (Section 3.4).

scripts use `chr1_100_200`. pyGAGAM reads either form, parsing the chromosome, start, and end out of the name and storing them in the standard `var` columns that every later step expects. Coordinates are settled once, here. Nothing downstream has to reparse a name or guess a convention.

`pp.load_genome` reads the *gene annotation*: a catalogue, distributed as a standard `GTF` file, that records where every gene lies on the genome, on which chromosome and between which coordinates, on which strand, and where each of its exons falls. The construction step needs these positions to decide which peaks belong to which gene, so `load_genome` parses the file into two tables, one row per gene, carrying its position, strand, and transcription start site, and one row per exon. Two quirks of real annotation files had to be handled along the way.

The first is a naming mismatch. The peaks label their chromosomes `chr1`, `chr2`, and so on, whereas annotation files from the RefSeq database use catalogue codes such as `NC_000001.11`; the two do not line up until they are reconciled, so `load_genome` accepts an optional lookup table that maps one naming scheme onto the other. Annotations from the GENCODE and UCSC databases already use the `chr1` names and need no such step.

The second is a filtering trap. GAGAM scores protein-coding genes only, so genes are filtered by their *biotype*, the tag that records whether a gene codes for a protein. In a `GTF` this tag sits on the gene's own row but not on the rows that describe its exons, so filtering the exon rows by it would discard every exon and empty the exon contribution. pyGAGAM instead keeps every exon whose parent gene passes the filter, rather than testing for a biotype tag on the exon row itself, which keeps the contribution intact.

One dataset in the source study needs a further step. The peaks of 10X's original `atac_v1_pbmc_5k` sample are aligned to the older hg19 build, while the gene annotation and the cCRE registry are on hg38. Overlapping the two directly fails quietly: the shift between builds is large enough that few peaks fall inside the correct gene, and the matrix collapses to a few thousand genes rather than the thirteen to fifteen thousand of a complete matrix. `pp.liftover_peaks` converts the peak coordinates between builds with a standard UCSC chain file, and `pp.detect_genome_build` reads the build the data records so the mismatch is caught rather than left to corrupt the result. The lift uses the `pyliftover` package when it is installed and a small built-in chain reader otherwise. Either way it adds no hard dependency.

The final preprocessing step, `pp.classify_peaks`, labels each peak with its regulatory class: it overlaps the peaks with the ENCODE SCREEN registry of candidate cis-regulatory elements and writes the resulting class into `var`. It fetches the registry as a plain table, checks it against a known checksum, caches it, and intersects it with the peaks in memory, keeping only the standard chromosomes; this avoids the reference pipeline's conversion of a UCSC track with an external

command-line tool, which stumbled over the alternate contigs, the non-standard sequences that the track and the peak set did not share. Two SCREEN registry releases are built in, a current one, used by default, and an earlier one. For an exact reproduction, pyGAGAM can also load the very labels the original scripts read, the UCSC `encodeCcreCombined` track. The function records only the class the registry assigns to a peak; deciding which classes count as promoters and which as enhancers is left to the construction step, described shortly.

3.3.2 Co-accessibility

With the peaks labelled, the next stage measures co-accessibility between peaks: the signal the construction later uses to link distal enhancer peaks to the promoters they may regulate. As the Background chapter described, GAGAM obtains it from Cicero [9], an R tool that pools similar cells into metacells and fits a regularised model within a genomic window. pyGAGAM does not reimplement the statistics; it calls CIRCE [11], a Python reimplementation of Cicero's method, through a single function, `tl.coaccessibility`, so the whole step stays in Python rather than shelling out to R. The result is a peak-by-peak graph, written to the standard pairwise slot `varp`, that the enhancer contribution reads later.

By default the step follows GAGAM's. It first aggregates the cells into metacells, groups of about fifty neighbours, since single cells are too sparse to correlate directly, and then fits CIRCE's sliding graphical lasso across a window of roughly 500 kilobases to score how strongly each pair of nearby peaks co-varies. The window size and the metacell count are exposed as parameters, set by default to the values Cicero uses, and a single-cell mode that skips the metacell step is available for small inputs. Co-accessibility is the most expensive stage of the pipeline, and Section 3.6 describes how pyGAGAM speeds it up without changing its output.

`tl.coaccessibility` is deliberately a thin wrapper: CIRCE does the modelling, and the wrapper's job is to make it behave like a well-mannered scverse step. It reads the peak coordinates in whichever naming convention they arrive in, maps the genome to the organism CIRCE expects, and manages the temporary metacell object internally so the caller never has to. It also protects the caller's data. CIRCE's metacell construction computes an embedding and a neighbour graph and writes them into the same `AnnData` slots that scanpy uses for its own; the wrapper snapshots those slots and restores whatever the caller had in them, so running co-accessibility never overwrites an embedding or neighbour graph the user had already computed.

3.3.3 Modular gene-activity construction

The construction is the core of GAGAM and the core of the port. It takes the labelled peaks, the genome tables, and the co-accessibility graph and produces the gene activity matrix by combining three contributions, one for each kind of regulatory evidence the Background chapter introduced. Each contribution is a separate function that returns a cell-by-gene matrix; `tl.build_gam` runs all three and sums them, while a caller can equally compute them one at a time and combine them by hand, which is how a single contribution can be inspected or replaced. The promoter is the anchor, so it comes first.

A gene is treated as active in a cell only where its promoter is open, so the first step decides which peaks are promoters and which gene each belongs to. `tl.identify_promoter_peaks` marks a peak as a promoter if it carries the promoter cCRE label or overlaps a gene's transcription start site, keeps it only when the nearest gene lies within a short distance, and assigns it to that single gene. `tl.promoter_contribution` then sums the accessibility of each gene's promoter peaks across the cells, giving the first estimate of the gene's activity.

The exon contribution adds evidence from within the gene body. `tl.exon_contribution` finds peaks that fall over a gene's exons and adds their accessibility to that gene, weighted so that a peak counts for more the closer it sits to the transcription start site. This term is applied only where the gene's promoter is already accessible, so it refines an active gene's score rather than switching a gene on by itself.

The enhancer contribution brings in distal regulation. `tl.enhancer_contribution` takes the distal-enhancer peaks, the `dELS` class, that are co-accessible with a gene's promoter, keeps those whose link is strong enough and within a bounded distance, and weights each by how strongly it co-varies with the promoter. The linked accessibility is then attributed to the gene through the same promoter-to-gene map, so an enhancer contributes to a gene only by way of a promoter the construction has already tied to it.

`tl.combine_contributions` sums the three into one matrix. Using all three gives GAGAM1, the paper's headline version; dropping the exon term gives GAGAM2, and `tl.build_gam` selects between them through a single argument. A final normalisation, taken from Cicero, places the values on a comparable scale across cells, and the result is the cell-by-gene gene activity matrix that the rest of the pipeline uses. The exact form of that normalisation, and the other construction details that had to match the original, are the subject of the next subsection.

3.3.4 Faithfulness and the parameter surface

GAGAM is described in enough detail to reproduce, and pyGAGAM reproduces it: on identical inputs its matrix matches the paper's own released one to a per-gene correlation of about 0.99. Reaching that agreement took care with a few

details that move the result more than their prominence in the paper suggests, and with one point where the paper's text and its released code disagree outright. There, pyGAGAM follows the code, since the code is what produced the published numbers.

That one disagreement is how far a distal enhancer may sit from its promoter. The paper's text gives 30 kilobases, following Cicero's guideline, but the released pipeline uses 300 kilobases, and it is the larger value that reproduces the published scores. pyGAGAM defaults to 300 kilobases and exposes the distance as a parameter, so the text's 30 kilobases is a one-argument change away.

Two further details are set out in the paper but easy to implement loosely, and each moves the result. A promoter peak is tied to a gene through the *enlarged gene body*, the gene together with a short region before its start, rather than through the start point alone; implementing it that way keeps peaks that carry a promoter-like label but sit far from any gene out of distant genes' scores. And a peak that is both inside a gene body and labelled a distal enhancer is treated as an enhancer, not as an exon peak, so it is never counted in both terms; honouring that label priority keeps the exon term from reaching roughly twice as many genes as it should. Both are described in the paper; the work was in reproducing them precisely and confirming the outcome against its released matrix.

The subtlest step to match was the final normalisation. The tempting shortcut, rescaling each cell by its number of accessible peaks, is not what GAGAM does: it leaves the three contributions almost perfectly correlated with the original yet yields a visibly different finished matrix. Reproducing the original meant porting Cicero's own `normalize_gene_activities` exactly, which works in three moves. It first gives each cell a size factor that stands in for its sequencing depth, obtained by regressing every cell's total gene activity on its number of accessible peaks and reading off the fitted value, and divides the cell's row by that factor. It then applies an `expm1` transform, and finally rescales each cell so that its values sum to one. Ported exactly, it reconstructs the paper's normalised matrix to the last digit, and it was what finally brought pyGAGAM's matrix into line with the original.

Each of these choices, along with the cCRE label vocabulary and the co-accessibility threshold, is a documented parameter of `build_gam`, not a value fixed in the code (Table 3.2). A paper-faithful configuration and a modern-default one are then the same function called with different arguments, and it is the faithful configuration that drives the reproduction reported in the Results chapter, where the agreement is quantified in full.

Table 3.2: The main configurable choices of peak annotation and construction, with their defaults. Each is a parameter, so a paper-faithful run and a present-day one differ only in the arguments passed.

Choice	Default	Notes / paper-faithful setting
cCRE registry	SCREEN v4 (current)	UCSC `encodeCcreCombined` track for the paper's exact labels; `v3` also selectable
Contributions	promoter + exon + enhancer (GAGAM1)	promoter + enhancer for GAGAM2
Promoter / distal labels	`PLS` / `dELS`	`prom` / `enhD` with the UCSC labels
Distal distance cap	300 kb	the released-code value that reproduces the paper; the text's 30 kb via the same parameter
Promoter gene-body window	500 bp	promoter peak to gene gate
Exon TSS-decay length	5 kb	exon weighting
Co-accessibility threshold	mean of positive scores	which enhancer links are kept

3.4 GRAIGH: a GeneHancer-element matrix

GRAIGH [14] is the companion method introduced in the Background chapter, and pyGAGAM reimplements it on the same foundation as the gene activity matrix. It starts from the same cell-by-peak accessibility but summarises it onto regulatory elements rather than genes: the features become *GeneHancer elements*, entries from the GeneHancer database [15], each with genomic coordinates and a set of known target genes. The construction is a sibling to `build_gam`, sharing its `AnnData` contract and its place in the `pp`/`tl`/`pl` API, and it yields a cell-by-element matrix.

`pp.load_genehancer` reads the GeneHancer reference into two tables: the elements, each with an identifier, coordinates, a type (enhancer or promoter), and a confidence score; and the associations that link each element to its target genes. GeneHancer differs from the cCRE registry used earlier in one practical respect. It is licence-restricted and distributed only for human (hg38), so the loader never downloads it and instead takes file paths the user supplies. The association table supplies each element's target genes, and those genes let an element-level result be read back in gene terms.

`tl.build_gh_matrix` then applies the GRAIGH construction. It overlaps every peak with the element intervals to form a binary peak-to-element map, projects the accessibility through that map so that each element takes on the summed accessibility of the peaks overlapping it, drops elements that no peak overlaps, and normalises each cell. The result is a new `AnnData` of cells by GeneHancer elements, its `var` carrying each element's identifier, its overlapping-peak count, and its target genes. Being an `AnnData` like any other, it clusters with the same `tl.cluster_cells` the gene activity matrix uses.

One further tool ties the two representations together. `tl.feature_specificity` scores how specific each feature is to a cell type, in the paper's sense of a feature accessible in one cell type but not the others, and because it works on any cell-by-feature matrix it runs unchanged on a GeneHancer-element matrix and on a gene activity matrix. This common footing lets the two be compared directly, which is the point of the element view: GRAIGH's premise is that regulatory elements are more cell-type-specific than whole-gene activity, and the comparison sets up the GRAIGH results reported later. GRAIGH is an extension of the same foundation: it reuses the data model, clustering, and tooling, and adds a regulatory-element view alongside the gene-activity one that remains the core of this work.

3.5 Evaluation tooling

Building the matrix is only half the work; the Results chapter also has to judge how good it is and compare it with other methods. That calls for three tools, all

part of the same library: a way to group the cells into clusters, a score for the matrix, and baseline methods to compare it against (Figure 3.3).

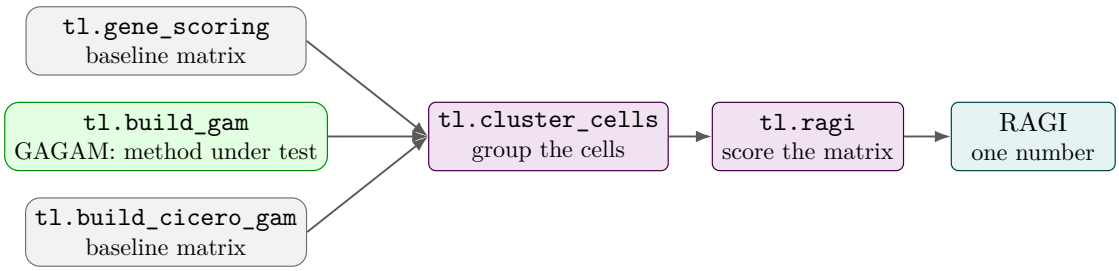


Figure 3.3: The evaluation harness. Each gene activity matrix, whether from GAGAM (the method under test) or from one of the two baselines, passes through the same two steps: `tl.cluster_cells` groups the cells and `tl.ragi` scores the matrix on that grouping. Because the same clustering recipe and the same scorer are applied to every matrix, a difference in the resulting RAGI reflects the matrix itself rather than the machinery around it.

Grouping the cells comes first, because the score is computed group by group, so a different grouping gives a different score. To compare pyGAGAM's score with the paper's fairly, then, the cells must be grouped the same way the paper grouped them. `tl.cluster_cells` reproduces the paper's recipe, which is Monocle3's [20]: it compresses the matrix to a few informative dimensions with LSI, lays the cells out with UMAP so that similar cells sit close together, links each cell to its nearest neighbours, and cuts the resulting graph into clusters with the Leiden algorithm (Figure 3.4). That last step hides a detail that is easy to get wrong and that decides the outcome. Leiden takes an *objective*, the rule for what counts as a good cluster, and a *resolution* that tunes how finely it cuts. The paper pairs one particular objective, CPM, with a very small resolution; handing that same small resolution to scanpy's default objective instead would lump every cell into a single cluster. Reproducing the paper's objective and resolution together is what keeps pyGAGAM's RAGI comparable to the paper's.

`tl.ragi` then computes RAGI, the score defined in the Background chapter: it is high when the marker genes pile up in a few clusters while the housekeeping genes stay spread out, and is computed as the mean Gini of the markers minus that of the housekeeping genes across the clusters. It is a port of the original `computeRAGI.py` and applies the same metric, so every matrix, whatever produced it, is judged on one consistent footing.

Two baseline matrices complete the comparison, the two the paper measures GAGAM against (Table 3.3). `tl.gene_scoring` is a window-based method, scoring

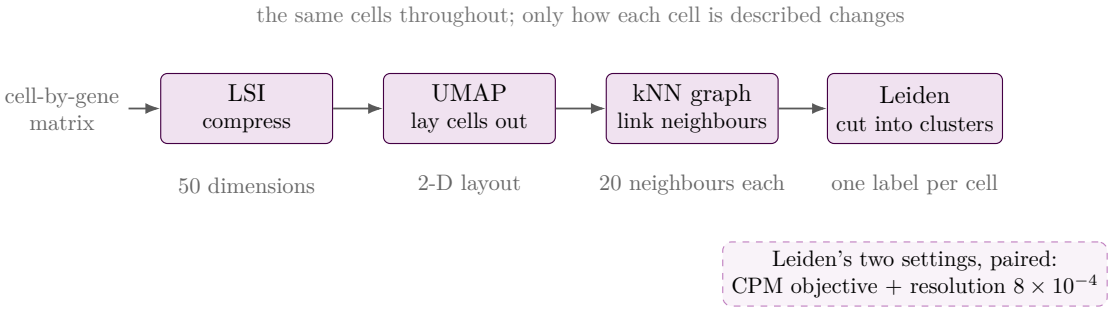


Figure 3.4: The clustering recipe reproduced by `tl.cluster_cells`, following Monocle3. LSI compresses the cell-by-gene matrix to 50 informative dimensions, UMAP lays the cells out in two dimensions, a k-nearest-neighbour graph ($k = 20$) links each cell to its closest neighbours in that layout, and Leiden cuts the graph into clusters. The set of cells never changes; each step only changes how a cell is described, from a long row of activities down to a single cluster label. Leiden's two settings must be reproduced together: the CPM objective paired with a small resolution (8×10^{-4}), since the same resolution under a different objective would merge every cell into one cluster.

a gene from the peaks in a fixed window around its start site, weighted by distance, in the manner of the gene-activity tools surveyed in the Background chapter; it stands for the simple distance-based approach. `tl.build_cicero_gam` reimplements Cicero's own gene-activity builder [9], the co-accessibility method GAGAM itself builds on, which CIRCE does not provide. Having both inside pyGAGAM lets the paper's comparisons be reproduced from one library instead of re-running the original R tools.

Table 3.3: The gene-activity methods compared in the Results chapter and what each requires. Gene scoring uses proximity alone; Cicero adds co-accessibility; and GAGAM adds the annotation-based promoter, exon, and enhancer terms.

Method	Basis	Terms combined	Requires
Gene scoring	distance to the TSS	one window term	peaks, genome
Cicero	co-accessibility	promoter + linked distal	peaks, genome, co-accessibility
GAGAM	annotation + co-accessibility	promoter + exon + enhancer	peaks, genome, cCRE labels, co-accessibility

3.6 Engineering for scale

The third objective is performance and scalability. Two steps dominate the runtime, co-accessibility and clustering, and pyGAGAM accelerates both under a single rule: speed up only what can be sped up without changing the result, and leave the faithfulness-critical steps alone. Two independent levers do the work, summarised in Table 3.4; the measured speed-ups are reported in the Results chapter.

The first lever needs no special hardware. Building the metacells takes much of the co-accessibility step's time, and the reference did it with a single-threaded neighbour search. pyGAGAM replaces that with a multithreaded search and computes the aggregation as a single sparse matrix product, using every core. Its output is bit-identical to the original, the same metacells and the same co-accessibility graph, so it is a free speed-up and is on by default.

Clustering can additionally use an optional GPU backend. LSI and UMAP are arithmetic that maps well onto a GPU, so pyGAGAM runs them there when a suitable GPU and its libraries are present, streaming the data in batches to keep memory bounded even at a million cells, and falling back to the CPU otherwise. The neighbour graph and the CPM Leiden step are deliberately kept on the CPU: their GPU equivalents do not reproduce them exactly and would break the faithful

clustering. Under both levers the gene activity matrix is unchanged and the quality scores match the unaccelerated run, so faithfulness is preserved; the size of the speed-up is left to the Results chapter.

Table 3.4: Where each step of the two cost centres runs, and why. The multithreaded metacell construction and the optional GPU backend for LSI and UMAP are the two speed-ups; the graphical lasso and the CPM Leiden step stay on the CPU so that the result stays faithful. Timings are reported in the Results and Discussion chapter.

Step	Runs on	Note
Metacell construction	CPU, multithreaded	bit-identical; the CPU speed-up
Graphical lasso	CPU	kept faithful; peak-driven cost
LSI and UMAP	GPU, optional	batched, with CPU fallback; the GPU speed-up
k-NN graph, Leiden (CPM)	CPU	kept on the CPU for faithful clustering

Chapter 4

Results and Discussion

This chapter reports what pyGAGAM delivers and what those results mean. It opens with the evaluation setup: the datasets every result is drawn from, the baselines each measurement is made against, and the metrics used to judge quality. The reproduction results follow, and they are the core of the chapter: scored like for like, the Python library closely matches the original GAGAM across all five datasets and both evaluation axes, to within about 0.035 RAGI. From there the chapter turns to behaviour at scale, presents the GRAIGH results, and walks through three short case studies built entirely on the public API. It closes with a discussion of what the results do and do not establish, and the limitations of the work.

4.1 Evaluation setup

Correctness is judged against the original GAGAM study on the same data it used. The evaluation reuses the five public single-cell ATAC datasets from Martini et al. [1]: two 10X scATAC PBMC samples (the older **V1.0.1** and the newer **V2.0.0** chemistry), the ATAC half of two 10X Multiome PBMC samples (**ChromiumX** and **Controller**), and a 10X **mouse brain**. They span about 4,600 to 11,900 cells and 97,000 to 162,000 peaks; full sizes and sample identifiers are in Appendix A. Four are human and one is mouse, which also exercises the pipeline on a second genome build.

Every end-to-end pyGAGAM result comes from a complete run from the raw data. Each dataset starts as the unprocessed 10X `filtered_peak_bc_matrix`, and pyGAGAM computes every step itself, from peak labelling and co-accessibility through gene activity matrix construction to clustering and scoring, with no cached intermediates carried over from the reference pipeline. Peaks are labelled with the UCSC `encodeCcreCombined` cCRE registry, the same source the paper used, so

that the comparison reflects the implementation and not a change of annotation. The V1.0.1 sample is distributed against the hg19 assembly and is lifted to hg38 with `pp.liftover_peaks` before labelling, so that all four human datasets share one coordinate system. Because clustering carries a random seed, every pyGAGAM figure is a mean and standard deviation over four seeds (42 to 45). Both variants of the method are evaluated throughout: **GAGAM1** (promoter, exon and enhancer contributions) and **GAGAM2** (promoter and enhancer only).

Quality is measured with **RAGI**, the Residual Average Gini Index introduced in the Background chapter. It is the mean Gini index of a panel of cell-type marker genes minus the mean Gini index of a panel of housekeeping genes: a good gene activity matrix concentrates each marker in the few cell groups that express it (a high marker Gini) while spreading housekeeping genes evenly across all of them (a low housekeeping Gini), so a higher RAGI means cleaner cell-type separation. The paper reports it on two axes, and both are kept here because they answer different questions. On the first, the self-clustering axis (the paper's Figure 7), each matrix is clustered on its own and then scored on that clustering; this rewards a matrix that both carries cell-type signal and clusters well, but it entangles the two. On the second, the common-clustering axis (Figure 8), every matrix is scored against a single shared grouping derived once from the raw accessibility data, so differences in clustering drop out and what remains reflects the matrix values. The second axis is therefore the cleaner test of whether pyGAGAM reproduces GAGAM's numbers, while the first shows the method in the setting the original study reported.

Each RAGI value is read against three reference points. **Paper** is the value published by Martini et al., computed with their original `computeRAGI.py` script and Monocle3 clustering. **Corrected GAGAM** takes the paper's *own* released gene activity matrix and re-scores it, unchanged, with pyGAGAM's RAGI and clustering; because the matrix is the paper's but everything downstream is pyGAGAM's, it is the like-for-like baseline that separates the gene activity matrix from how it happens to be scored and clustered. **pyGAGAM** is the library's own matrix, built from raw data and scored the same way. The Corrected GAGAM and pyGAGAM columns are directly comparable; the Paper column is not, for the two reasons explained in the next section. Every reproduction claim that follows is made against the Corrected GAGAM baseline.

One further pair of metrics appears later, for the GRAIGH results. Where a reference grouping of the cells is available, whether ground-truth labels or another clustering, the agreement between a computed clustering and that reference is measured with the Adjusted Rand Index (ARI) and Adjusted Mutual Information (AMI). Both are corrected for chance, so a random assignment scores near zero and a perfect match scores one; they are defined where they are first used, in the GRAIGH results (Section 4.5).

Experimental setup. All results were produced with pyGAGAM and its scverse-stack dependencies, CIRCE (`circe-py`) for co-accessibility and scanpy, `leidenalg`, and `umap-learn` for the clustering and scoring tools; the library targets Python 3.11 and above, and the accelerated results use pyGAGAM 0.2.0, which introduces the two levers discussed in Section 4.3. The reference implementation is the original GAGAM R pipeline (Signac, Monocle3, and Cicero) in a package-pinned environment matching the source study. The quality metrics (RAGI, and the ARI and AMI used later) are deterministic given the random seeds and do not depend on hardware; they reproduce identically across released versions of the library. Runtimes do depend on hardware and are reported in Section 4.3: the single-threaded from-raw runs were measured on a dedicated multi-core Linux node, with two datasets run on a shared node and reported as upper bounds, and the accelerated and atlas-scale runs on a single NVIDIA RTX A5000 (24 GB) or A4000 (16 GB) GPU host. Full dataset provenance, reference-annotation and software versions, and the hardware are collected in Appendix A.

4.2 Faithful reproduction of GAGAM

The most direct test of faithfulness is whether the reimplemented construction produces the same matrix as the original when it is given the same inputs. It does. The original study released its full set of intermediate matrices for one dataset, the ChromiumX Multiome; fed the reference's own labelled peaks, annotations, and co-accessibility for it, pyGAGAM's construction reproduces each contribution almost exactly. Across the roughly 13,000 genes, the promoter contribution correlates with the reference at 0.999 and the exon and enhancer contributions at about 0.98. The Cicero normalisation applied on top is reproduced bit for bit, so it does not degrade the agreement, and the full gene activity matrix correlates at about 0.99 per gene. Reaching that agreement meant recovering several construction details exactly, the promoter gene-assignment rule, the exon-selection rule, the tie-breaks, and the normalisation, all described in the Methodology chapter; with those in place, the construction is faithful given identical inputs. This element-wise check covers the one dataset with a complete released set of intermediates; the end-to-end comparison below covers all five.

Element-wise agreement on shared inputs shows the construction is faithful; the end-to-end comparison shows that the complete pipeline, run from raw data, reproduces the original's quality. This is measured with RAGI on the two axes introduced in Section 4.1, and against the Corrected GAGAM baseline rather than the published values, for the reason given next.

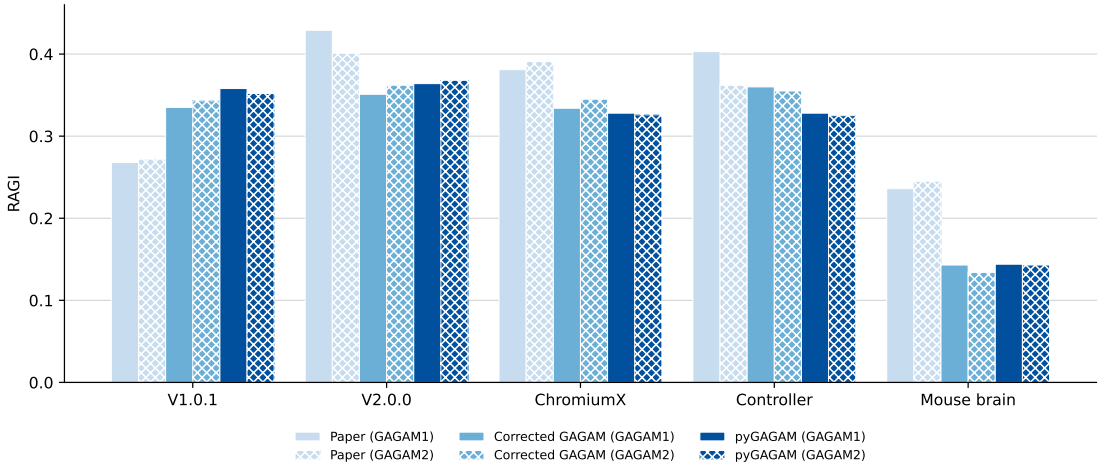
The published RAGI values cannot be compared directly with a clean re-run, for two reasons: a bug in the original scoring script, and a different clustering method.

The scoring bug is in how the marker-gene list is built: each comma-separated entry is split without trimming whitespace, so every marker after the first keeps a leading space and no longer matches the gene names in the matrix. Many markers are silently dropped, and RAGI is then computed over the small, unusually cell-type-specific subset that remains, which inflates the marker Gini; on its own this raises the score by roughly 0.02 to 0.05 on the common axis. The effect is on the score, not the matrix, which is unchanged. The second difference, and the larger one, is the clustering: the paper's self-clustering axis uses Monocle3, whereas the like-for-like baseline uses pyGAGAM's clustering throughout. Because the Paper column carries both effects at once, it is not uniformly inflated: it diverges from the corrected values by as much as 0.14 on V1.0.1 (common axis) in one direction and about 0.08 on the mouse brain in the other, far more than the scoring bug alone could account for. It is therefore not a clean baseline. Re-scoring the paper's own released matrix with corrected marker parsing and pyGAGAM's clustering removes both differences and gives the Corrected GAGAM baseline, against which every comparison below is made.

Figures 4.1 and 4.2 report the two axes, each pairing a grouped bar chart with the exact values beneath it.

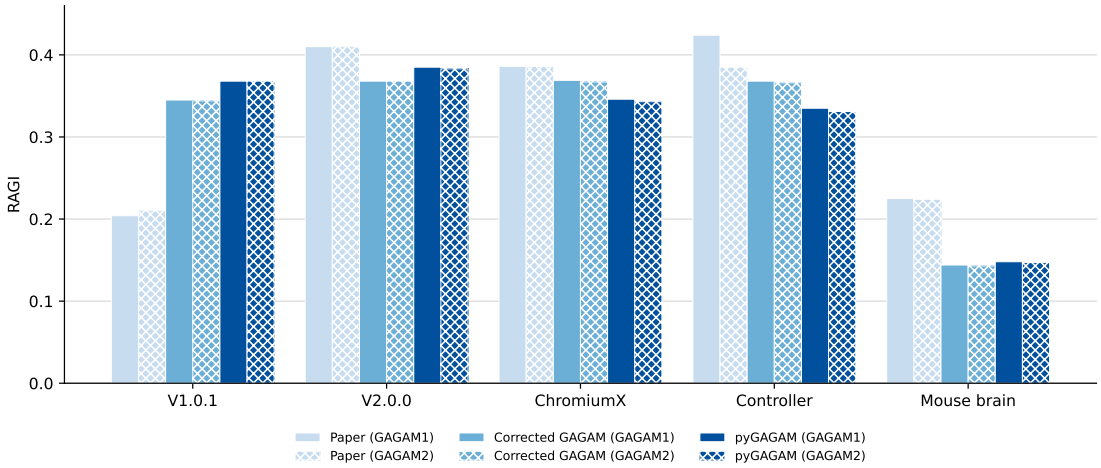
On the like-for-like Corrected GAGAM baseline, pyGAGAM stays close to GAGAM on every dataset and both axes: the difference ranges from about -0.035 to $+0.024$ RAGI, and is within ± 0.025 on four of the five datasets. Set against the seed-to-seed noise (standard deviation ≤ 0.011), the small positive deltas on the two PBMC scATAC sets and the mouse brain are within that noise, while the Controller deficit of about -0.033 is several times larger and a real, if small, difference. Because every matrix is clustered the same way, these differences reflect the matrices themselves rather than the number of clusters. pyGAGAM is slightly higher on the two PBMC scATAC sets, close on the mouse brain, and lower by up to about 0.035 on the two Multiome sets. Part of that deficit is pyGAGAM's lighter from-raw quality control: on the Controller Multiome, given identical accessibility input, the self-clustering difference narrows from about -0.032 to -0.018, so quality control accounts for roughly half, and the residual -0.018 is pyGAGAM's CIRCE co-accessibility differing slightly from the paper's Cicero, not the construction. The mouse brain sits much lower in absolute terms, about 0.14 for both methods; that low value comes partly from the shared clustering applied to mouse accessibility, not from a defect in either matrix, since the paper reports a higher figure on the same released matrix under its own clustering. What matters for reproduction is that the two track each other closely, as close as 0.001, the tightest agreement of the five.

One dataset needed extra care. V1.0.1 is distributed on the hg19 assembly, whereas the gene annotation and cCRE registry are hg38. Without lifting the peaks first, few of them overlap the annotation and the matrix collapses to about 3,100



	Paper		Corrected GAGAM		pyGAGAM		Δ (pyGAGAM − Corrected)	
Dataset	G1	G2	G1	G2	G1	G2	G1	G2
V1.0.1 PBMC	0.268	0.272	0.335	0.343	0.358	0.352	+0.023	+0.009
V2.0.0 PBMC	0.429	0.401	0.351	0.362	0.364	0.368	+0.013	+0.006
Multiome ChromiumX	0.381	0.391	0.334	0.345	0.328	0.326	−0.006	−0.019
Multiome Controller	0.403	0.362	0.360	0.355	0.328	0.325	−0.032	−0.030
Mouse brain	0.236	0.245	0.143	0.134	0.144	0.143	+0.001	+0.009

Figure 4.1: RAGI on the self-clustering axis (the original study's Figure 7): each matrix is clustered on itself and scored on that clustering. G1 and G2 denote GAGAM1 (promoter, exon, enhancer) and GAGAM2 (promoter, enhancer), shown as solid and hatched bars; higher RAGI is better. *Paper* reproduces the published values; *Corrected GAGAM* re-scores the paper's own released matrix with pyGAGAM's clustering and RAGI (the like-for-like baseline); *pyGAGAM* is the library's own matrix built from raw data (mean over four random seeds, standard deviation ≤ 0.011). The Paper column uses the original scoring and clustering and is not directly comparable; the comparison of record is Corrected GAGAM versus pyGAGAM.



	Paper		Corrected GAGAM		pyGAGAM		Δ (pyGAGAM − Corrected)	
Dataset	G1	G2	G1	G2	G1	G2	G1	G2
V1.0.1 PBMC	0.204	0.210	0.345	0.344	0.368	0.368	+0.023	+0.024
V2.0.0 PBMC	0.410	0.410	0.368	0.368	0.385	0.383	+0.017	+0.015
Multiome ChromiumX	0.386	0.386	0.369	0.367	0.346	0.343	−0.023	−0.024
Multiome Controller	0.424	0.385	0.368	0.366	0.335	0.331	−0.033	−0.035
Mouse brain	0.225	0.224	0.144	0.143	0.148	0.147	+0.004	+0.004

Figure 4.2: RAGI on the common-clustering axis (the original study's Figure 8): every matrix is scored against one shared clustering derived from the raw accessibility data, which isolates the matrix values from differences in clustering. Columns and bar conventions are as in Figure 4.1.

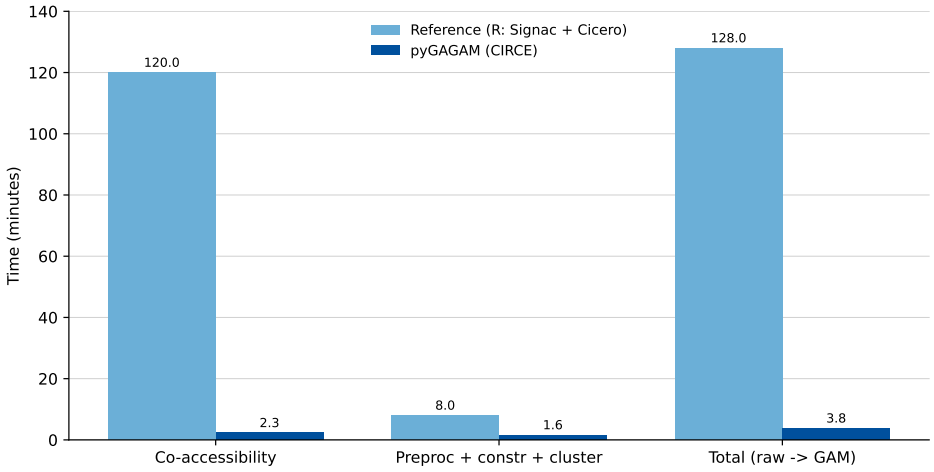
genes with a negative RAGI. Lifting the peaks to hg38 with `pp.liftover_peaks` before annotation restores a full matrix of about 12,800 genes and the values shown in the tables, so V1.0.1 is not a special case once its coordinates are on the right assembly.

4.3 Engineering for scale

The reference pipeline is slow enough to be a practical obstacle. On the V2.0.0 dataset, going from raw peaks to the gene activity matrix takes about 128 minutes, almost all of it in Signac preprocessing and Cicero co-accessibility. pyGAGAM performs the same from-raw run on the same machine in about 3.8 minutes, roughly 30 times faster (Figure 4.3); these are the same runs whose quality is reported in Section 4.2, within ±0.035 RAGI of the corrected baseline. That run is single-threaded and predates the acceleration levers; most of the speedup is the co-accessibility step, where the CIRCE backend replaces Cicero, with pyGAGAM's own preprocessing and construction contributing the rest. Figure 4.4 breaks the timing down across the five datasets: a complete from-raw run takes about 4 to 17 minutes, and co-accessibility is roughly half of each (about 47 to 63 per cent). The two levers designed in Section 3.6 take the pipeline further, to atlas scale.

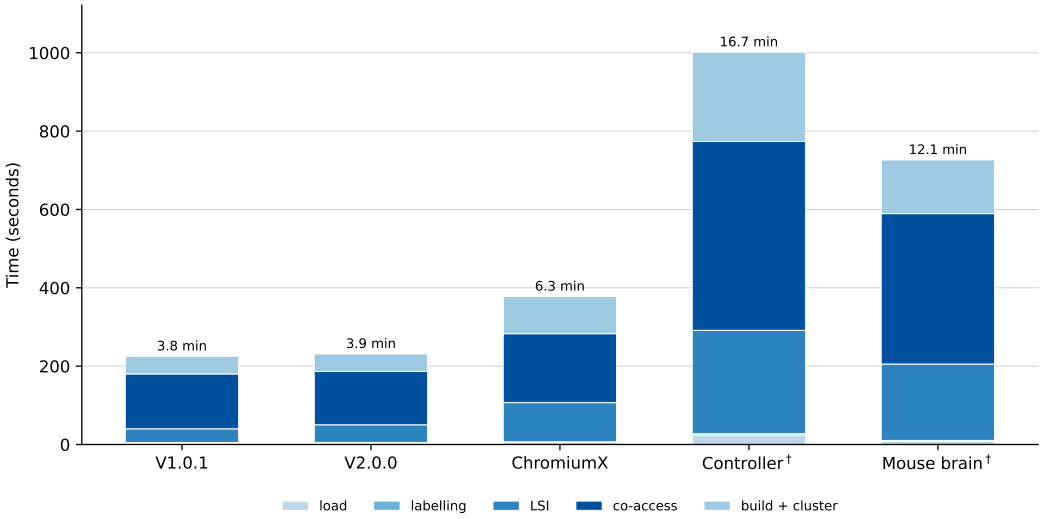
The library has two cost centres, co-accessibility and clustering, and each has an accelerated path that leaves the result unchanged. The first is a multithreaded metacell construction. The reference groups cells with a single-threaded nearest-neighbour search that dominates co-accessibility; replacing it with a multithreaded brute-force search returns the same neighbours bit for bit while using every core. This alone speeds co-accessibility by up to about 21 times at 500,000 cells (the nearest-neighbour step itself by about 30 times), needs no GPU, and so benefits every install. The second is an optional GPU backend for the two arithmetic-heavy clustering steps, LSI and UMAP, which at the 500,000-cell scale run about 26 and 74 times faster on a GPU and make the clustering stage as a whole about 6 times faster. The k-nearest-neighbour graph, the Leiden step, and RAGI stay on the CPU deliberately, because keeping the clustering faithful depends on their exact behaviour, the graph edge weights and Leiden's CPM objective.

Figure 4.5 shows the effect on atlas-scale data: CATlas [22], a 717,000-cell human atlas, at cell counts from 100,000 up to its full size, on a single NVIDIA RTX A5000 GPU with a 64-thread host. The end-to-end speedup grows with cell count, from about 4 times at 100,000 cells to about 10 times at 500,000, because the multithreaded lever removes the dominant original cost, metacell construction; of the ten-fold speedup at 500,000 cells, the metacell fix accounts for about 79 per cent and the GPU backend the remaining 21. The full atlas, impractical on the reference pipeline, completes in about 35 minutes.



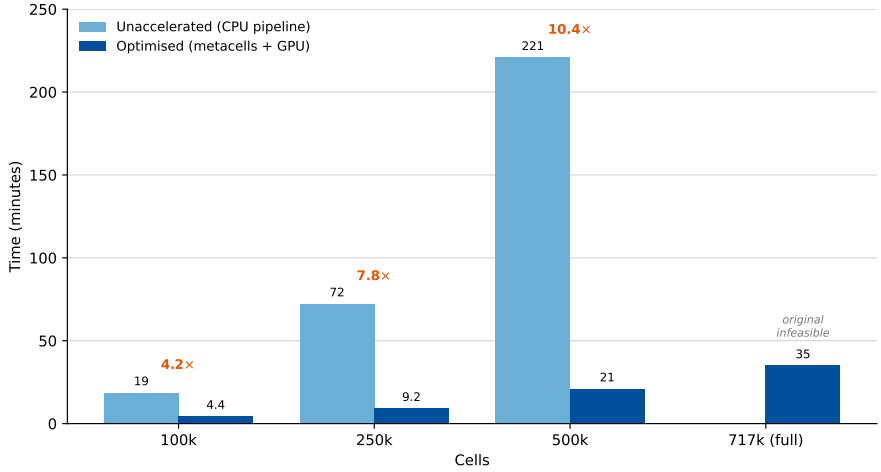
Stage	Reference (R)	pyGAGAM
Co-accessibility	≈2 hours (Cicero)	2.3 min (CIRCE)
Preprocessing, construction, clustering	≈8 min	1.6 min
Total (raw → GAM)	≈128 min	≈3.8 min

Figure 4.3: Runtime of a complete from-raw run (raw peaks to gene activity matrix) on the V2.0.0 dataset: the reference GAGAM R pipeline versus pyGAGAM, on the same dedicated node, single-threaded, with RAGI scoring excluded on both sides. The reference pipeline was benchmarked head-to-head on V2.0.0; it uses the same Cicero-dominated toolchain on every dataset. The reference figure is a single measured run and the pyGAGAM figures are means of four.



Dataset	load	labelling	LSI	co-access	build + cluster	total
V1.0.1 PBMC	3.5	1.6	35	140	45	3.7 min
V2.0.0 PBMC	3.1	2.2	45	136	45	3.8 min
Multiome ChromiumX	5.0	1.9	100	176	95	6.3 min
Multiome Controller†	24	3.5	264	482	228	16.6 min
Mouse brain†	8.0	2.4	195	384	137	12.2 min

Figure 4.4: pyGAGAM from-raw runtime per stage (seconds unless noted), for a complete run from raw peaks to the scored gene activity matrix. Co-accessibility (CIRCE) is the largest stage throughout, and each value is the mean of four runs. "LSI" is the dimensionality reduction that builds the shared clustering axis. Runs marked † were on a shared node and are upper bounds; the others on a dedicated multi-core node.



Cells	Unaccelerated	Optimised	Speedup
100,000	18.7 min	4.4 min	4.2×
250,000	72.3 min	9.2 min	7.8×
500,000	220.9 min	21 min	10.4×
717,000 (full)	—	35 min	—

Figure 4.5: Scaling on CATlas, a 717,000-cell human atlas, from a full from-raw run at each cell count (a single run per size). "Speedup" compares the optimised path (multithreaded metacells + GPU embedding) against pyGAGAM's own unaccelerated CPU pipeline at the same size (not the reference R pipeline); no ratio is shown for the full atlas, which the unaccelerated pipeline cannot run at that size. These are a best case: CATlas here uses a small peak set (about 33,000 cCREs), so the peak-driven graphical lasso is cheap; at a realistic 150,000-peak density the full atlas completes in about 47 minutes (see text).

These end-to-end figures are a best case. CATlas here uses a small peak set, about 33,000 cCREs, whereas typical scATAC has 100,000 or more. Co-accessibility has a cell-driven part (metacell construction, which the multithreaded lever speeds) and a peak-driven part (the graphical lasso, still on a single CPU core), and the graphical lasso grows super-linearly with peak count: from about 25 seconds at 33,000 peaks to about 14 minutes at the full 926,000-cCRE ENCODE registry. At a realistic 150,000-peak density the full atlas still finishes, measured end to end, in about 47 minutes on a comparable single-GPU host, and at the full registry the run becomes bound by data input and the graphical lasso rather than by the accelerated steps. GPU memory is not the constraint: the LSI backend streams the matrix in batches and holds about 3 GB of VRAM even at one million cells, so it stays modest across the whole range. The pipeline therefore reaches atlas scale in under an hour, and the two remaining bottlenecks, a GPU graphical lasso and out-of-core data loading, are the clearest targets for future work.

The acceleration is transparent to the results. The multithreaded metacells reproduce the reference's neighbours and gene activity matrix to a correlation of 1.0; on the paper datasets the accelerated pipeline reproduces the Section 4.2 RAGI within seed noise, with the common-axis matrix bit-identical to the unaccelerated run; and on CATlas the GPU and CPU paths agree to within 0.008 RAGI at every size. Every result in this chapter therefore holds whether or not the acceleration is enabled.

4.4 Packaging, testing, and availability

Reproducing the science is one objective; delivering GAGAM as usable, maintainable software is the other. The outcome is an installable, tested, and documented package that follows scverse conventions, summarised in Table 4.1. pyGAGAM is published on a public package index (TestPyPI) and installs with `pip`. It operates natively on `AnnData` and exposes the scanpy-style `pp/tl/pl` API catalogued in the Methodology chapter (Table 3.1). Its behaviour is checked by 185 unit and integration tests across thirteen modules, covering every public step from genome loading and peak annotation through co-accessibility, construction, clustering, and scoring, and reaching about 92 per cent line coverage (88 per cent under the stricter branch measure; per-module figures in Appendix A.4). The GPU acceleration backends are excluded from this figure: they need CUDA hardware the continuous-integration environment does not provide, and are instead validated by the scaling results in Section 4.3, where the GPU and CPU paths agree to within 0.008 RAGI. The suite runs under a GitHub Actions continuous-integration workflow across the supported Python versions. A documentation site carries the API reference and a worked end-to-end tutorial. The package was generated

from the community `cookiecutter-scverse` template, so it meets the ecosystem's structural expectations for layout, typing, and metadata.

Table 4.1: The software-engineering deliverables of the second objective and the evidence for each.

Deliverable	Evidence
Installable package	`pygagam` on TestPyPI (v0.3.0), installable with `pip`
scverse-native API	`AnnData` in and out; `pp`/`tl`/`pl` (Table 3.1)
Test suite	185 unit and integration tests across 13 modules; ≈92% line coverage (88% branch)
Continuous integration	GitHub Actions workflow across the supported Python versions
Documentation	API reference and a worked tutorial on the documentation site
scverse conformance	generated from the `cookiecutter-scverse` template

4.5 GRAIGH results

The same foundation reimplements GRAIGH, the companion method introduced in the Background and Methodology chapters. Where GAGAM summarises accessibility over genes, GRAIGH summarises it over GeneHancer regulatory elements, giving a gene-regulation view of the same data. As an additional contribution it is validated more lightly than GAGAM, on three points: that the reference loader is faithful, that the element matrix recovers the cell grouping the peak matrix produces, and that element-level accessibility is comparably cell-type-specific to gene activity.

Loader fidelity. The GeneHancer database is licensed and user-supplied rather than bundled, so the first requirement is simply that the loader reads it correctly. `pp.load_genehancer` parses the full GeneHancer v5.15 release, the same release used by the original GRAIGH study, into **393,464 elements and 2,408,198 element-to-gene associations**, matching the counts reported for that release [14].

Clustering. If the element matrix carries the same cell-type structure as the underlying peaks, clustering it should recover the same groups. Clustering the GeneHancer-element matrix with the recipe used throughout this work (LSI, UMAP, Leiden) and comparing the result to the peak-based clustering gives, on the V2.0.0 PBMC dataset at about eighteen clusters, an **Adjusted Mutual Information**

of 0.80 and an Adjusted Rand Index of 0.62 (a single run). Both indices measure how well two groupings of the same cells agree, corrected so that a random grouping scores near zero and an identical one scores one; 0.80 and 0.62 mean the element matrix recovers most of the peak-level structure, though the agreement is looser on the stricter Rand index. The original GRAIGH study [14] reports 0.87 and 0.80 on its own PBMC sample; our figures are somewhat lower, and the comparison is indicative rather than matched, as it is a different sample clustered at a different resolution. The number of peaks overlapping each element follows a similar distribution to the original study, with most elements covering one or a few peaks.

Specificity. GRAIGH's motivating claim is that regulatory-element accessibility is more cell-type-specific than gene activity. `tl.feature_specificity` computes the original study's per-feature, per-cell-type specificity score, and because it is feature-agnostic it runs identically on a GeneHancer-element matrix and on a GAGAM gene activity matrix, which makes the two directly comparable. On the NeurIPS BMMC multiome dataset (about 20,000 cells, seven lineages), across ten canonical markers, the result is neutral overall: the mean specificity difference is about +0.006, and the element matrix is the more specific of the two on five of the ten markers. The direction holds for lymphoid markers, for example CD8A (element 0.99 versus gene-activity 0.93), CCR7, and CD4, but not for the rest. This does not reproduce the large advantage the original study reported, for two reasons. First, that study compared against a broad gene-body-based (Signac) gene-activity matrix whose specificity is far lower (about 0.45 to 0.75), leaving room to improve on it, whereas the promoter-and-exon GAGAM matrix is already highly specific (0.93 to 0.99) and leaves little headroom. Second, the comparison here is on a different tissue, bone marrow rather than the study's PBMC. Because the tool scores both representations the same way, the finding is a fair, neutral one: against a matrix as sharp as GAGAM's, element-level accessibility is comparably specific rather than clearly more specific.

4.6 Extensibility case studies

A library is only as useful as the analyses it makes easy. Three short studies, each built on the public API, show that the composable contribution model supports real downstream work; the two regulatory-element studies among them are included for what they demonstrate about the framework, their point being methodological rather than biological.

Inspecting a gene's activity. A gene activity value is a single number, which makes it hard to see why a gene scored as it did. `pl.plot_gene_connections` (with `tl.gene_peak_links`) turns that number back into a picture: for one gene it draws the promoter peak, the exon peaks, and the co-accessible distal enhancers that GAGAM sums into the gene's activity, with each peak's cCRE class shown beneath (Figure 4.6). For the B-cell marker MS4A1 on the V2.0.0 PBMC data, the promoter fans out to the distal `dELS` enhancers co-accessible with it, exactly the peaks that build MS4A1's activity. The plot is assembled from the same `pp` and `tl` outputs used throughout the pipeline, so any run can be inspected this way without extra machinery.

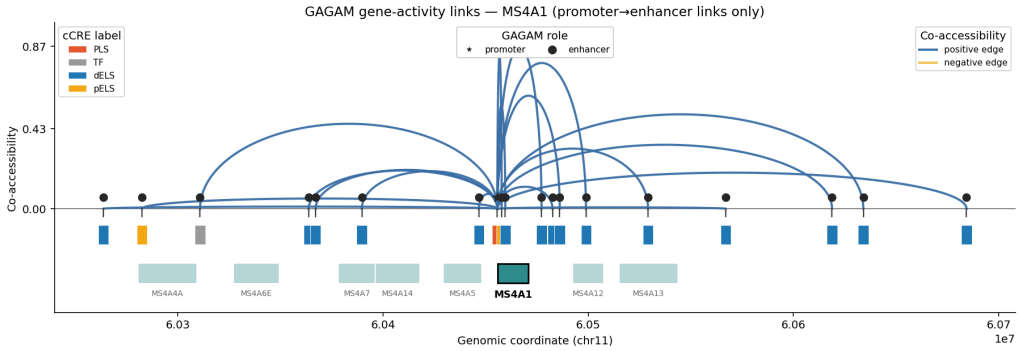


Figure 4.6: Gene-activity connections for the B-cell marker MS4A1 (V2.0.0 PBMC), drawn by `pl.plot_gene_connections`. The promoter peak (⋆) links by co-accessibility to the distal `dELS` enhancer peaks (•) that GAGAM sums into MS4A1's activity; the coloured row beneath the peaks is each peak's cCRE class. The figure is built from ordinary `pp` and `tl` outputs, so any run can be inspected the same way.

New distal element classes. ENCODE's SCREEN v4 registry defines two accessibility-based cCRE classes beyond the active enhancers (`dELS`) GAGAM uses: `CA-CTCF` (accessible and CTCF-bound) and `CA-TF` (accessible and transcription-factor-bound). A natural question, and one the advisor raised, is whether feeding these to genes as additional distal contributions, through the same co-accessibility pathway, improves the matrix. Because the contribution API already exposes the distal label set, this needed no new library code: each class is added as its own independently normalised enhancer term and combined with the rest (Listing 4.1).

The manipulation adds each class on top of the standard GAGAM1 matrix (promoter, exon, and the `dELS` enhancer term), varying only the distal enhancer contribution. Two controls are added the same way: a size-matched set of random peaks, and a size-matched subset of `dELS` itself. Across the four human PBMC

```
# each cCRE class becomes its own independently-normalised enhancer term
e_dels = pg.tl.enhancer_contribution(adata, pp, distal_labels=("dELS",))
e_ctcf = pg.tl.enhancer_contribution(adata, pp, distal_labels=("CA-CTCF",))
e_tf   = pg.tl.enhancer_contribution(adata, pp, distal_labels=("CA-TF",))

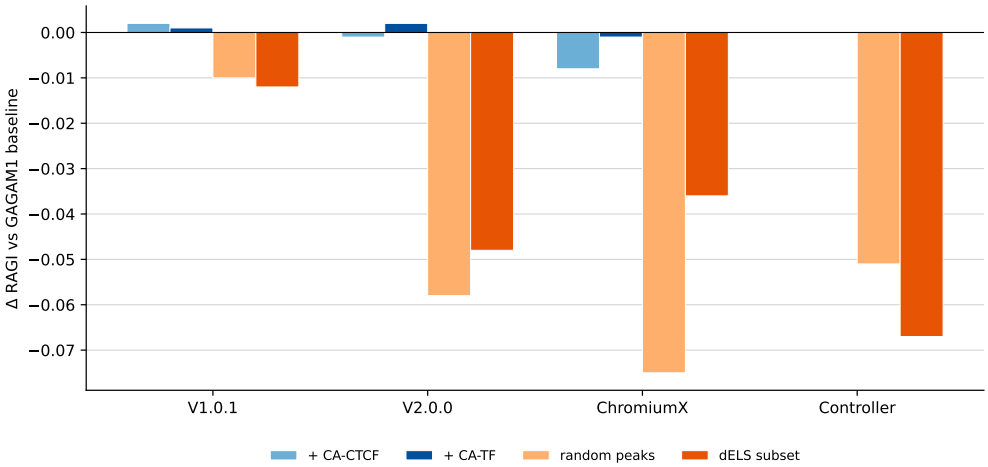
baseline = pg.tl.combine_contributions([prom, exon, e_dels], source_adata=adata)
plus_v4  = pg.tl.combine_contributions([prom, exon, e_dels, e_ctcf, e_tf],
    source_adata=adata)
```

Listing 4.1: Adding the two SCREEN v4 classes as their own enhancer terms, using only the existing API.

datasets, scored with cluster-count-matched RAGI, and on the NeurIPS BMMC multiome scored against paired RNA (per-gene Spearman correlation and agreement with the annotated cell types), the two new classes make no significant change: RAGI stays within ±0.008 of the baseline, inside the roughly ±0.01 seed-to-seed spread, and the multiome correlation and cell-type agreement are flat (Figure 4.7). The controls carry the result: both the random peaks and the `dELS` subset *degrade* the matrix, by up to about 0.075 RAGI, well beyond the seed spread. The `dELS` subset degrades because re-adding peaks GAGAM already uses, as a second independently normalised term, over-weights them and distorts the matrix. So `CA-CTCF` and `CA-TF` are genuine regulatory elements, distinct from `dELS` and unlike random peaks, that nonetheless add nothing to gene activity beyond the co-accessible `dELS` peaks already used. This is biologically sensible: `dELS` carries the active enhancer mark and the two new classes do not. The study rides on a real package parameter and is documented on the project site.

CTCF as insulator boundaries. CTCF also marks the boundaries of three-dimensional chromatin domains. A different idea, then, treats `CA-CTCF` peaks not as extra signal but as insulators, pruning co-accessibility links that cross them, with the caveat that not every CTCF site is a functional boundary. Prototyped in a few lines on top of the co-accessibility network, dropping edges whose two peaks straddle a `CA-CTCF` peak, and tested on V2.0.0 and the multiome, this again left GAM quality unchanged within noise, but produced two clean observations. Removing the cross-boundary links, about 27 per cent of the total, cost no accuracy, whereas removing the same number of random links clearly hurt (RNA correlation down; paired test across genes, $p \approx 5 \times 10^{-55}$); and `CA-CTCF` peaks cut fewer links than random boundaries would (27 against 41 per cent), so co-accessibility already partly respects these domains. This last study is exploratory and is not a shipped feature, only a short script on top of the library, but it makes the section's point: a clean, composable API turns a three-dimensional-architecture idea into a quick, controlled experiment.

The three studies exercise the same contribution model in different ways: one inspects a single gene, one tests a new annotation, and one prototypes an architectural hypothesis, each in a few lines of API calls. For the two regulatory-element studies the biological verdict is the same and honestly modest, that the richer cCRE classes carry real signal GAGAM's existing model already captures; the reusable outcome is the framework that made both cheap to run.



Variant (vs GAGAM1 baseline)	V1.0.1	V2.0.0	ChromiumX	Controller
+ `CA-CTCF`	+0.002	−0.001	−0.008	0.000
+ `CA-TF`	+0.001	+0.002	−0.001	0.000
random peaks (control)	−0.010	−0.058	−0.075	−0.051
`dELS` subset (control)	−0.012	−0.048	−0.036	−0.067

Figure 4.7: Change in RAGI from the GAGAM1 baseline when each variant is added as a separate distal term (cluster-count-matched, mean over the resolution-and-seed sweep; the seed spread is about ±0.01). The two new classes stay within the seed spread on every dataset; both controls degrade well beyond it. On the NeurIPS BMMC multiome, cell-type agreement (ARI) is 0.059 at baseline, 0.058 and 0.061 with `CA-CTCF` and `CA-TF`, and 0.044 and 0.050 for the random-peak and `dELS`-subset controls.

4.7 Discussion and limitations

This chapter set out to test three things, one per objective, and the results speak to each. The reproduction is faithful (O1): given the same inputs, pyGAGAM rebuilds the reference matrix almost exactly, at a per-gene correlation of about 0.99 on the one dataset with released intermediates, and from raw data it matches the like-for-like Corrected GAGAM baseline to within about 0.035 RAGI across all five datasets and both evaluation axes. It is delivered as usable software (O2): the method is now an installable, tested, and documented scverse-compatible package, where it had been a set of research scripts. And it is faster and scales (O3): a full from-raw run, benchmarked head-to-head on V2.0.0, is roughly thirty times quicker than the reference R pipeline, and the accelerated path processes a 717,000-cell atlas in under an hour without changing the result. GRAIGH and the three case studies extend the same foundation to a companion method and to real downstream work.

Two points are worth drawing out. First, the reproduction holds only after the paper's scoring-script bug is corrected and one clustering is applied to both sides, the Corrected GAGAM baseline of Section 4.2; measured that way, an independent reimplementation lands on the same numbers, which indicates that GAGAM's published results largely reflect the method and not its particular R toolchain. The qualifier is real: the co-accessibility engine is itself part of that toolchain, and swapping Cicero for CIRCE leaves the small footprint discussed below. Second, the biology the two regulatory-element case studies probed proved real but already captured by GAGAM's existing model. The contribution is a faithful, general framework, and its value is what others can build on.

The evaluation is scoped to the reproduction question, and that shapes what it can and cannot claim. The aim is to show that an independent Python reimplementation reproduces GAGAM, so the test is the one the original study used: the same five datasets, the same RAGI metric, and both of the paper's evaluation axes. Matching the reference on its own terms is what faithfulness means here, and it does not call for an external gold standard the original study never used. RAGI does carry a known caveat, inherited rather than introduced: it is an unsupervised internal proxy that depends on the marker and housekeeping gene panels, so it bounds how much any RAGI-based result, GAGAM's or ours, can say about the underlying biology. That limit belongs to the metric and the method, not to the fidelity of the reimplementation. The main comparison is also made against the paper's own released matrix, re-scored under a single clustering; isolating the gene-activity values from how they happen to be scored and clustered is the point of that baseline, but it is worth being explicit that it is the re-scored reference and not the literal published number.

Two further limits are about coverage rather than metric. The datasets are

narrow in tissue, four PBMC samples and one mouse brain, the same set the original study used, so neither study speaks to how the method behaves on other tissues. And several individual results each rest on a single dataset, because the reference intermediates or a head-to-head runtime were available for only one: the element-wise correlation on ChromiumX, the head-to-head runtime on V2.0.0, the GRAIGH clustering parity on one PBMC sample from a single run, and the GRAIGH specificity comparison on ten markers of a single non-PBMC tissue (bone marrow), where it was neutral and did not reproduce the original study's larger advantage.

Two further caveats concern the method itself. pyGAGAM's co-accessibility uses CIRCE, which reimplements Cicero's approach but is not bit-identical to it, and the from-raw pipeline applies lighter quality control than the original; together these account for the small residual gap on the Multiome datasets, quantified on Controller and likely similar on ChromiumX. The headline speedups, in turn, are a best case at a low peak count: at realistic peak densities the single-threaded graphical lasso and the data input become the bottlenecks, so the largest atlas runs are I/O- and CPU-bound and the accelerated steps are no longer the limit. A GPU graphical lasso and out-of-core loading are the clearest next targets.

One availability gap remains: the package is currently on TestPyPI, a staging index, rather than on production PyPI. Full dataset provenance, reference annotations, software versions, and the hardware used are collected in Appendix A. The Conclusion chapter draws these outcomes together and sets out the directions the limitations open.

Chapter 5

Conclusion and Future Works

This thesis set out to turn GAGAM, a method for building gene activity matrices from scATAC-seq, from a collection of research scripts into a maintainable, interoperable Python library, without changing its results. This chapter summarises what was delivered and outlines where the work can go next.

5.1 Conclusion

The outcome is pyGAGAM, a pure-Python, scverse-compatible library that reimplements GAGAM v1.2 and meets the three objectives set at the start. It reproduces the original method faithfully (Objective 1): given the same inputs its construction rebuilds the reference matrix almost exactly, at a per-gene correlation of about 0.99 on the one dataset with released intermediates, and a complete from-raw run matches the original's quality, scored like for like, to within about 0.035 RAGI across all five published datasets and both evaluation axes, running slightly lower only on the two Multiome sets. It integrates with the scverse ecosystem (Objective 2): it operates on `AnnData` through a scanpy-style `pp/tl/pl` API and ships as an installable, documented package with 185 tests at about 92 per cent coverage, continuous integration, and the layout the `cookiecutter-scverse` template enforces. And it is substantially faster and scales well beyond the original data (Objective 3): a complete from-raw run, single-threaded and before any acceleration, is roughly thirty times quicker than the reference R pipeline on the one dataset benchmarked head-to-head, chiefly because the CIRCE co-accessibility backend replaces Cicero; with optional multicore and GPU acceleration the pipeline processes a 717,000-cell atlas in under an hour, leaving the result unchanged within seed noise. That the reproduction lands on the published numbers at all, once a bug in the original

scoring script is corrected and one clustering is applied to both sides, indicates that GAGAM's results reflect the method itself rather than its particular R toolchain.

Beyond reproducing the method, the work decomposes GAGAM's monolithic construction into small, independently usable contribution units that recombine into the complete matrix. The same decomposition let the library also reimplement GRAIGH, the companion gene-regulation method: its loader reproduces the GeneHancer registry exactly, and clustering its element matrix recovers the peak-level grouping at an adjusted mutual information of 0.80. Three short case studies, a gene-centric visualisation and two regulatory-element investigations, then each took only a few lines of code. pyGAGAM contributes no new algorithm; its value is faithful, efficient, extensible software that puts an established method in the tools the single-cell community already uses and gives others a base to build on. The two regulatory-element studies make the point: they went looking for new biological signal and found the existing model already captures it. The framework that made those studies cheap to run is what lasts.

5.2 Future work

These results come with real boundaries. Faithfulness is established on the reference's own terms, using RAGI, an unsupervised internal proxy, over five mostly-PBMC datasets, so broader tissues and an independent readout remain open; several of the directions below follow directly from those limits.

- **Removing the remaining bottlenecks.** At realistic peak counts the single-threaded graphical lasso (the co-accessibility solver) and data input, rather than the accelerated steps, dominate the largest runs; a GPU graphical lasso and streaming, out-of-core loading are the clearest targets for pushing beyond atlas scale. Clustering is a related case: LSI and UMAP already run on the GPU, but the neighbour graph and Leiden's CPM objective are kept on the CPU to preserve fidelity, and a faithful GPU version of those steps remains open.
- **Closing the from-raw quality-control gap.** About half of the small deficit on the Multiome datasets traces to pyGAGAM's lighter from-raw quality control, the cell and peak filtering applied before the matrix is built; matching the original's filter thresholds there would remove a known, quantified source of that gap.
- **Broader and more independent evaluation.** The reproduction leans on RAGI and on datasets that are mostly PBMC. Adding more tissues, more random seeds, an independent readout such as agreement with matched RNA for the main comparison (not only for GRAIGH), and a comparison against

other gene-activity methods would test generality and reduce the dependence on a single internal metric.

- **Extending GRAIGH.** The port currently covers GRAIGH's element matrix and its specificity metric; its remaining component, the co-accessibility extension that intersects co-accessibility links with GeneHancer associations, is a natural next addition on the same foundation, completing the companion method in the library.
- **Wider release and integration.** The package is published to a staging index; a production release, together with the documentation site and a listing in the scverse ecosystem, would complete its availability. The longer-term direction the original proposal envisaged, a single package unifying several of the group's gene-activity and gene-regulation methods (including the Cross-Modality-TF analysis not implemented here), fits the same architecture.

With GAGAM faithfully reproduced, packaged, and accelerated, the nearest of these steps, a production release and a scverse listing, would put the method directly into the hands of the single-cell community.

Appendix A

Reproducibility

This appendix records the datasets, reference annotations, software versions, and hardware behind the Results and Discussion chapter.

A.1 Datasets

All five evaluation datasets are public 10X Genomics samples, the same ones used by the GAGAM v1.2 study [1]. Table A.1 lists their sizes; "Genes" is the number of genes in the constructed gene activity matrix. The scaling study (Section 4.3) uses CATlas [22], a human scATAC-seq atlas (GEO accession GSE184462), about 717,000 cells.

Table A.1: The five evaluation datasets: 10X Genomics sample, size, and genome build. "Genes" is the size of the constructed gene activity matrix.

Dataset	10X sample	Cells	Peaks	Genes	Genome
V1.0.1 PBMC	`atac_v1_pbmc_5k`	5,335	97,119	12,790	hg19 → hg38
V2.0.0 PBMC	`atac_pbmc_5k_nextgem`	4,623	135,317	13,287	hg38
Multiome ChromiumX	`10k_PBMC_Multiome_nextgem_Chromium_X`	10,970	111,680	13,143	hg38
Multiome Controller	`10x_PBMC_Multiome_Controller`	11,898	143,836	13,373	hg38
Mouse brain	`8k_mouse_cortex`	7,749	162,277	12,819	mm10

A.2 Reference annotations

- **cCRE registry:** the UCSC `encodeCcreCombined` track (hg38 for human, mm10 for mouse), used to match the source study. pyGAGAM's own default is the ENCODE4 / SCREEN v4 registry; the benchmarks pass the UCSC track to reproduce the paper's labelling.

- **Gene annotation:** the GAGAM v1.2 gene annotation (hg38 and mm10). `pp.load_genome` is annotation-agnostic and accepts any GTF; the exact GENCODE release was not recorded in the run logs.
- **Liftover:** V1.0.1 is distributed on hg19 and is lifted to hg38 with the UCSC `hg19ToHg38.over.chain` chain via `pyliftover` (`pp.liftover_peaks`).

A.3 Software environment

Table A.2 lists the pinned versions of the main dependencies. The quality and single-threaded timing results were produced with pyGAGAM 0.1.x, the acceleration and scaling results with 0.2.0, and the current released version is 0.3.0; the quality metrics are deterministic and reproduce identically across these versions. The reference pipeline is the original GAGAM implementation in R (Signac, Monocle3, and Cicero) in a package-pinned environment matching the source study.

Table A.2: Pinned versions of the main pyGAGAM dependencies (from the project lock file).

Component	Version
Python	3.11 or later
anndata	0.12.13
scanpy	1.11.5
numpy	1.26.4
scipy	1.17.1
scikit-learn	1.8.0
pandas	2.3.3
umap-learn	0.5.12
pyranges	0.1.4
circe-py (CIRCE)	0.4.0

Hardware. Two Linux compute nodes were used, both running Ubuntu 22.04.5 LTS (kernel 5.15) with CUDA 13.2. The single-threaded from-raw timings in Section 4.3 were measured on a dedicated, idle node: an AMD Ryzen 9 5950X (16 cores, 32 threads), 64 GB of RAM, and an NVIDIA RTX A4000 (16 GB). This node produced the clean V1.0.1, V2.0.0, and ChromiumX figures and the head-to-head reference-pipeline comparison. The Controller and mouse-brain timings, marked as upper bounds, were run on a second, shared node under concurrent load from other users: a dual-socket Intel Xeon Silver 4216 (32 cores, 64 threads), 128 GB of RAM, and an NVIDIA RTX A5000 (24 GB). The acceleration and atlas-scale results were produced on the RTX A5000 node.

Determinism. Clustering uses seeds 42–45, and every pyGAGAM figure is the mean over these four seeds. Given the seeds and the inputs, RAGI, ARI, and AMI are deterministic.

A.4 Test coverage

The test suite is 185 unit and integration tests across thirteen test modules, run with `pytest` and `pytest-cov`. Overall coverage is about 92% of statements and 88% under branch coverage (1,381 statements in total). Table A.3 gives the per-module breakdown. The optional GPU acceleration paths in `tl.cluster_cells` are excluded from the measurement: they require CUDA hardware and the RAPIDS GPU stack, which is incompatible in a single Python interpreter with the `leidenalg`/`igraph` library the tests use. Those paths are validated instead by the scaling results in Section 4.3, where the GPU and CPU paths agree to within 0.008 RAGI. The lower per-module figures are confined to plotting code and a few network and error-handling branches; the core construction, co-accessibility, scoring, and clustering logic is covered at 87–100%.

Table A.3: Per-module test coverage (statements and combined statement-plus-branch coverage). Package `__init__` modules (100% covered) are omitted from the rows; the total is over the whole package. GPU-only code paths are excluded as described above.

Area	Module	Statements	Coverage
Preprocessing (`pp`)	`load_genome`	64	92%
	`classify_peaks`	105	96%
	`peak_coords`	26	95%
	`liftover_peaks`	124	91%
	`load_genehancer`	77	78%
Tools (`tl`)	`build_gam`	248	89%
	`contributions`	57	87%
	`coaccessibility`	124	82%
	`cicero`	75	89%
	`cluster_cells`	57	87%
	`ragi`	52	100%
	`gene_scoring`	37	100%
	`build_gh_matrix`	72	95%
	`feature_specificity`	30	92%
	`gene_peak_links`	59	78%
Plotting (`pl`)	`plot_gene_connections`	115	78%
	GRAIGH plots	33	100%
Total (whole package)		**1,381**	**88%**

Bibliography

[1] Lorenzo Martini, Roberta Bardini, Alessandro Savino, and Stefano Di Carlo. «GAGAM v1.2: An Improvement on Peak Labeling and Genomic Annotated Gene Activity Matrix Construction». In: *Genes* 14.1 (2023), p. 115. DOI: 10.3390/genes14010115 (cit. on pp. 1, 10, 14, 31, 53).

[2] Isaac Virshup, Danila Bredikhin, Lukas Heumos, et al. «The scverse project provides a computational ecosystem for single-cell omics data analysis». In: *Nature Biotechnology* 41.5 (2023), pp. 604–606. DOI: 10.1038/s41587-023-01733-8 (cit. on pp. 1, 13, 15).

[3] Jason D. Buenrostro, Paul G. Giresi, Lisa C. Zaba, Howard Y. Chang, and William J. Greenleaf. «Transposition of native chromatin for fast and sensitive epigenomic profiling of open chromatin, DNA-binding proteins and nucleosome position». In: *Nature Methods* 10.12 (2013), pp. 1213–1218. DOI: 10.1038/nmeth.2688 (cit. on p. 4).

[4] Jason D. Buenrostro, Beijing Wu, Ulrike M. Litzenburger, Dave Ruff, Michael L. Gonzales, Michael P. Snyder, Howard Y. Chang, and William J. Greenleaf. «Single-cell chromatin accessibility reveals principles of regulatory variation». In: *Nature* 523.7561 (2015), pp. 486–490. DOI: 10.1038/nature14590 (cit. on p. 5).

[5] Darren A. Cusanovich, Riza Daza, Andrew Adey, Hannah A. Pliner, Lena Christiansen, Kevin L. Gunderson, Frank J. Steemers, Cole Trapnell, and Jay Shendure. «Multiplex single-cell profiling of chromatin accessibility by combinatorial cellular indexing». In: *Science* 348.6237 (2015), pp. 910–914. DOI: 10.1126/science.aab1601 (cit. on p. 5).

[6] Tim Stuart, Avi Srivastava, Shaista Madad, Caleb A. Lareau, and Rahul Satija. «Single-cell chromatin state analysis with Signac». In: *Nature Methods* 18.11 (2021), pp. 1333–1341. DOI: 10.1038/s41592-021-01282-5 (cit. on pp. 7, 14).

[7] Jeffrey M. Granja, M. Ryan Corces, et al. «ArchR is a scalable software package for integrative single-cell chromatin accessibility analysis». In: *Nature Genetics* 53.3 (2021), pp. 403–411. DOI: 10.1038/s41588-021-00790-6 (cit. on pp. 7, 14).

[8] Kai Zhang, Nathan R. Zemke, Ethan J. Armand, and Bing Ren. «A fast, scalable and versatile tool for analysis of single-cell omics data». In: *Nature Methods* 21 (2024), pp. 217–227. DOI: 10.1038/s41592-023-02139-9 (cit. on pp. 7, 14).

[9] Hannah A. Pliner, Jonathan S. Packer, José L. McFaline-Figueroa, et al. «Cicero Predicts cis-Regulatory DNA Interactions from Single-Cell Chromatin Accessibility Data». In: *Molecular Cell* 71.5 (2018), pp. 858–871. DOI: 10.1016/j.molcel.2018.06.044 (cit. on pp. 7, 9, 10, 14, 22, 29).

[10] Jill E. Moore, Michael J. Purcaro, Henry E. Pratt, et al. «Expanded encyclopaedias of DNA elements in the human and mouse genomes». In: *Nature* 583.7818 (2020), pp. 699–710. DOI: 10.1038/s41586-020-2493-4 (cit. on p. 9).

[11] Rémi Trimbour, Julio Saez-Rodriguez, and Laura Cantini. «CIRCE: a scalable Python package to predict cis-regulatory DNA interactions from single-cell chromatin accessibility data». In: *Bioinformatics* 42.3 (2026), btag092. DOI: 10.1093/bioinformatics/btag092 (cit. on pp. 9, 22).

[12] Lorenzo Martini, Roberta Bardini, Alessandro Savino, and Stefano Di Carlo. «GAGAM: A Genomic Annotation-Based Enrichment of scATAC-seq Data for Gene Activity Matrix». In: *Bioinformatics and Biomedical Engineering (IWBBIO 2022)*. Vol. 13347. Lecture Notes in Computer Science. Springer, 2022, pp. 18–32. DOI: 10.1007/978-3-031-07802-6_2 (cit. on pp. 10, 14).

[13] Huidong Chen, Caleb Lareau, et al. «Assessment of computational methods for the analysis of single-cell ATAC-seq data». In: *Genome Biology* 20 (2019), p. 241. DOI: 10.1186/s13059-019-1854-5 (cit. on p. 11).

[14] Lorenzo Martini, Roberta Bardini, Alessandro Savino, and Stefano Di Carlo. «GRAIGH: Gene Regulation Accessibility Integrating GeneHancer Database». In: *2023 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. IEEE, 2023, pp. 343–348. DOI: 10.1109/BIBM58861.2023.10385417 (cit. on pp. 12, 14, 26, 42, 43).

[15] Simon Fishilevich, Ron Nudel, Noa Rappaport, et al. «GeneHancer: genome-wide integration of enhancers and target genes in GeneCards». In: *Database* 2017 (2017), bax028. DOI: 10.1093/database/bax028 (cit. on pp. 12, 26).

[16] Isaac Virshup, Sergei Rybakov, Fabian J. Theis, Philipp Angerer, and F. Alexander Wolf. «anndata: Access and store annotated data matrices». In: *Journal of Open Source Software* 9.101 (2024), p. 4371. DOI: 10.21105/joss.04371 (cit. on pp. 13, 17).

[17] Danila Bredikhin, Ilia Kats, and Oliver Stegle. «MUON: multimodal omics analysis framework». In: *Genome Biology* 23 (2022), p. 42. DOI: 10.1186/s13059-021-02577-8 (cit. on pp. 13, 17).

[18] F. Alexander Wolf, Philipp Angerer, and Fabian J. Theis. «SCANPY: large-scale single-cell gene expression data analysis». In: *Genome Biology* 19 (2018), p. 15. DOI: 10.1186/s13059-017-1382-0 (cit. on pp. 13, 18).

[19] Lorenzo Martini, Roberta Bardini, Alessandro Savino, and Stefano Di Carlo. *GAGAM: source code repository*. URL: https://github.com/smilies-polito/GAGAM (visited on 07/20/2026) (cit. on p. 14).

[20] Junyue Cao, Malte Spielmann, Xiaojie Qiu, et al. «The single-cell transcriptional landscape of mammalian organogenesis». In: *Nature* 566.7745 (2019), pp. 496–502. DOI: 10.1038/s41586-019-0969-x (cit. on pp. 14, 27).

[21] Lorenzo Martini, Roberta Bardini, Alessandro Savino, and Stefano Di Carlo. *GRAIGH: source code repository (GH-Co-Accessibility)*. URL: https://github.com/smilies-polito/GH-Co-Accessibility (visited on 07/20/2026) (cit. on p. 14).

[22] Kai Zhang et al. «A single-cell atlas of chromatin accessibility in the human genome». In: *Cell* 184.24 (2021), 5985–6001.e19. DOI: 10.1016/j.cell.2021.10.024 (cit. on pp. 37, 53).