ABSTRACT

Title of Thesis: SEQPROC: TOWARD A DECLARATIVE GEOMETRY SPECIFICATION LANGUAGE FOR PROTOCOL-AGNOSTIC READ PREPROCESSING FOR GENOMICS

Elan Fisher
Master of Science, 2026

Thesis Directed by: Associate Professor Rob Patro
Department of Computer Science

Modern single-cell genomics protocols encode critical metadata, such as cell barcodes, unique molecular identifiers, and linker sequences, directly into sequencing reads. Accurately identifying and extracting these elements is a prerequisite for all downstream biological analyses, yet the task is complicated by the proliferation of distinct library chemistries. Each chemistry features its own read geometry (i.e., the configuration and encoding of this technical data) and is subject to sequencing and PCR errors that can corrupt the expected structure. This thesis presents `seqproc`, a general-purpose sequence preprocessing tool built around a declarative domain-specific language called the Extended Fragment Geometry Description Language (EFGDL). EFGDL allows users to specify read structures without prescribing how the parsing should be performed. The language is compiled into an efficient execution graph by the `ANTISEQUENCE` library, which processes reads in a batched, multi-threaded fashion with a focus on memory reuse. We describe the design and implementation of EFGDL and `ANTISEQUENCE`, alongside a series of optimizations that produced a $\sim$4.8x speedup and

$O(1)$ memory usage. Furthermore, we detail the addition of edit distance matching and orientation-aware processing for long-read data. On benchmarks spanning four single-cell protocols, `seqproc` is consistently the fastest and most memory-efficient tool, using 3 to 305 times less memory than alternatives while achieving the highest read recovery on long-read datasets.

SEQPROC: TOWARD A DECLARATIVE GEOMETRY SPECIFICATION LANGUAGE FOR PROTOCOL-AGNOSTIC READ PREPROCESSING FOR GENOMICS

by

Elan Fisher

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Master of Science
2026

Advisory Committee:
Professor Rob Patro, Chair/Advisor
Professor Milijana Surbatovich
Professor Erin Molloy

Dedication

To my mother, whom I lost to cancer at the very start of my graduate studies. Her passing motivated me to pursue this research. It is my hope that as we learn to better and more quickly understand the lives of cells, we might grant other loved ones the time she was so tragically denied.

Acknowledgments

First and foremost, I would like to express my deepest gratitude to my advisor, Rob Patro. Thank you for welcoming me into your lab as a Master's student and for your unwavering guidance, patience, and generosity with your time. Your mentorship has been the driving force behind this thesis. To my collaborators, Noah Cape and Daniel Liu, without you both this thesis would not exist. Noah's foundational work and vision were instrumental to the existence of SEQPROC, and Daniel's architectural expertise made the ANTISEQUENCE execution engine a reality. I extend my sincere thanks to Erin Molloy and Milijana Surbatovich for graciously agreeing to serve on my thesis committee. Beyond their committee duties, they are incredible instructors who fundamentally expanded my perspective on how to approach computer science. I also wish to thank the Computer Science graduate staff at the University of Maryland for their administrative support and for working extremely hard to keep the logistics of this thesis organized and on track. Finally, I must thank my friends and family. Who at the very least, have convinced me that they are interested in my research.

Statement of Contributions

The work described in this thesis represents a collaboration among three researchers, whose distinct contributions are delineated here.

Noah Cape conceived the original idea for the EFGDL declarative language, designed the language semantics, and produced the protocol architecture diagrams and read-layout schematics. Figures contributed principally by Noah are marked with a dagger (†) in their captions.

Daniel Liu is the original author of ANTISEQUENCE, the execution engine described in Chapter 5. The core graph-based execution model, DAG construction, and the initial Myers bit-vector pattern matching integration are his work.

Elan Fisher (the author of this thesis) contributed the extensive performance optimizations described in Chapter 6 (batched read recycling, SmallVec internal collections, synchronization tuning, and the I/O path analysis), the annotation system and language additions to EFGDL, the benchmarking infrastructure and evaluation methodology, and the writing of this thesis.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

1.1 Motivation and Contributions

In 1990, sequencing the human genome was a project of national scope, requiring 13 years, over $3 billion, and the coordinated effort of 20 institutions on three continents. By 2022, the same task cost approximately $1,000 and took a single afternoon on a benchtop instrument [1]. This reduction in sequencing cost outpaces Moore's Law[1] by a factor of five.

Consequently, this approximately 1,000,000× decrease in three decades has produced a data generation rate that storage, compute, and analytical workflows are still catching up to.

These innovations in sequencing have yielded extraordinary scientific returns. It has made it possible to identify the genetic basis of rare diseases with a single blood draw [3], to track the clonal evolution of a tumor through treatment [4], to reconstruct the developmental trajectories of individual cells as an embryo forms [5], and to monitor the spread of infectious disease in near real-time during an outbreak [6]. Realizing these applications requires not just generating sequence data but processing it correctly, quickly, and at scale. A typical modern single-cell RNA-seq experiment produces between 50 million and 500 million reads in a single run [7].

Processing those reads correctly before any biological analysis is a prerequisite for every of

[1] Moore's Law predicts roughly $10^3\times$ improvement per decade; sequencing costs dropped $\sim 10^6\times$ in three decades, or $\sim 5\times$ faster [1, 2].

of these experiments. We must strip technical metadata sequences, validate barcodes, and reformat the output for downstream tools. When that step is slow, costly, or error-prone, it adds a significant bottleneck between data and insight.

We introduce SEQPROC to close this gap. We provide two core contributions. First, the Extended Fragment Geometry Description Language (EFGDL) expresses complex read structures concisely. Second, the ANTISEQUENCE execution engine provides high-throughput graph operations and pattern matching. Together, these components constitute a preprocessing system that we demonstrate to be up to 13.6× faster and require up to 305× less memory than existing alternatives. This thesis documents both contributions in full.

1.2 A Brief History of DNA Sequencing

1.2.1 First Generation. Sanger Sequencing

In 1977, Frederick Sanger introduced chain-termination sequencing, a method that produces short DNA fragments of known length by interrupting synthesis at controlled positions with dideoxy nucleotides [8]. The fragment lengths encode the sequence position of the terminal dideoxy nucleotide; reading each band from shortest to longest reconstructs the original sequence one base at a time. We generate all prefixes of the sequence and sort them by size, reading the terminal base of each to reconstruct the original.

Sanger sequencing is accurate but slow as it reads one fragment at a time. The Human Genome Project, completed in 2003, applied it at industrial scale [9]. Thousands of sequencing machines ran in parallel for over a decade, producing a single 3-gigabyte reference sequence at a cost of approximately $1 per base [9]. Each machine could produce approximately 800 bases per run. The logistical

challenge was not the chemistry but the combinatorics. Researchers had to assemble billions of short, overlapping reads into a contiguous genome. After 13 years, the initial findings were published in 2001 [9] and the reference sequence declared complete in 2003.

1.2.2 Second Generation. Next-Generation Sequencing

The key difference from Sanger sequencing is scale. The introduction of Illumina's sequencing-by-synthesis technology in the mid-2000s shifted sequencing from a largely serial process to a highly parallel one [10].

In this architecture, DNA fragments are seeded onto a glass flow cell and amplified into billions of localized clusters. Sequencing occurs through cycles where fluorescently labeled reversible-terminator nucleotides are added. A camera captures the signal from every cluster simultaneously, and then base-calling software then converts these images into sequence data. Since each cycle reads one base from every cluster in parallel, a single 150-cycle run can generate hundreds of gigabytes of raw data to produce billions of 150-base reads.

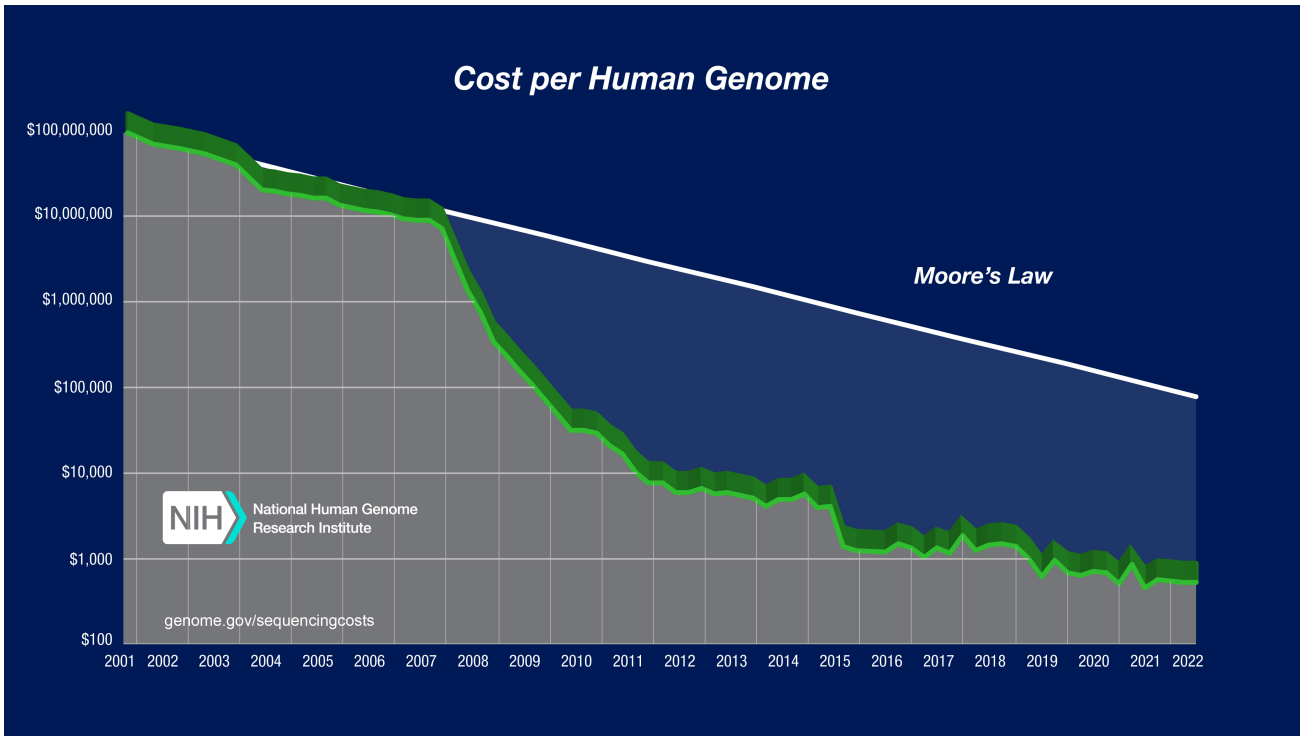

Figure 1.1: **Sequencing cost per human genome, 2001–2022.** Cost data from the NHGRI Genome Sequencing Program [1]. The inflection point around 2008 marks the transition from Sanger-based to short-read (Illumina) sequencing technology.

The impact on cost was dramatic. As shown in Figure 1.1, by 2010, the cost to sequence a human genome had fallen from $1 per base to approximately $10,000 total. By 2022, that number had fallen ten-fold to $1,000 [1]. What resulted was an explosion of short reads (typically 150 to 300 bases). This abundance, necessitated a new bioinformatics toolkit, including specialized short-read aligners, assembly graphs, and variant callers.

1.2.3 Third Generation. Long-Read Sequencing

While second-generation platforms excelled in throughput and cost-efficiency, their inherently short read lengths presented challenges for resolving complex or highly repetitive genomic regions. Third-generation sequencing addresses this limitation by generating reads measured in thousands or

tens of thousands of bases [11, 12].

The underlying mechanisms of these long-read technologies are physically distinct from the sequencing-by-synthesis approach of the second generation. Oxford Nanopore measures the disruption of ionic current as a single DNA or RNA strand is drawn through a protein pore by an applied voltage. Each passing k-mer produces a characteristic current signature that a neural network subsequently decodes into a base sequence. Conversely, PacBio's technology [12] utilizes a HiFi (*High Fidelity*) mode, which generates highly accurate consensus reads by circularizing the DNA template and sequencing the same molecule multiple times.

The engineering trade-off is straightforward. Longer reads simplify genome assembly and enable direct sequencing of repetitive regions and structural variants, but the per-base error rate is higher than Illumina, especially for insertions and deletions (indels) [11, 12]. PacBio HiFi achieves approximately 0.1% per-base error, compared to 0.01% for Illumina [10]. Oxford Nanopore R9.4 achieves approximately 1% to 2% per-base error [11]. While newer Nanopore R10 flow cells have significantly improved raw accuracy and reduced these indel issues [13, 14], handling the complex error profiles of diverse chemistries and legacy datasets remains a core computational challenge. For protocols that use fixed-sequence linkers as positional anchors, a single-base indel within the linker displaces every downstream element, making the choice of matching algorithm a correctness issue as well as a performance one.

1.3 Single-Cell Transcriptomics

Single-cell RNA sequencing (scRNA-seq) measures the transcriptional state of individual cells rather than averaging over a bulk tissue sample. In a bulk assay, a tumor biopsy yields one measure-

ment per gene representing the mean expression across thousands of cells of dozens of types. In contrast, in a single-cell assay, the same biopsy yields one measurement per gene per cell, revealing which subpopulations are actively dividing, which are undergoing apoptosis, and which are transcriptionally quiescent. A single bulk measurement discards the cell-type composition entirely, the single-cell measurement preserves it.

To assign each sequenced molecule to its cell of origin, single-cell protocols attach a short synthetic DNA sequence, called a *cell barcode*, to every cDNA molecule before sequencing. All molecules from the same cell carry the same barcode. Molecules from different cells carry different ones. A second short sequence, the *unique molecular identifier* (UMI), is attached individually to each molecule before amplification, so that duplicate copies generated during PCR can be collapsed to a single count[2]. These two metadata sequences appear in every sequenced read and must be identified and extracted before any biological analysis can begin.

As sequencing technologies have advanced from serial processing to massive parallelization and long-read capabilities, the primary bottleneck in genomics has shifted from data generation to data processing. Assays like single-cell transcriptomics compound this challenge by introducing intricate metadata sequences, such as barcodes and UMIs, that must be extracted precisely even when corrupted by the sequencing errors inherent to these platforms. Resolving these metadata structures at scale requires highly optimized, error-tolerant string matching algorithms.

[2]Without UMIs, all amplified copies of the same molecule are indistinguishable, and the read count conflates biological abundance with PCR amplification efficiency. A gene amplified $100\times$ would appear $100\times$ more abundant than one amplified only once, even if both originated from a single molecule. UMIs decouple molecular counting from amplification depth.

1.4 Outline of Thesis

To address these computational demands, the remainder of this thesis focuses on the design and implementation of robust bioinformatics architectures. Chapter 2 explains how single-cell sequencing encodes cellular identity in the read structure and why extracting that identity is computationally non-trivial. Chapter 3 surveys the existing preprocessing tools and identifies the tripartite trade-off, among generality, performance, and ease of configuration, that none of them resolves satisfactorily. Chapter 4 introduces EFGDL and walks through specifications for all four benchmarked protocols, building from the simplest fixed-length geometry to a long-read, orientation-aware, three-level combinatorial barcode. Chapter 5 describes the ANTISEQUENCE execution engine in full. We detail the compilation pipeline, the zero-copy memory model, the SIMD pattern matching algorithms, and the threading model. Chapter 6 documents the engineering process behind the performance results. We evaluate the optimization passes that succeeded, the experiments that failed and what they taught us, and the head-to-head benchmarks across four single-cell chemistries. Chapter 7 summarizes the contributions, states the limitations honestly, and identifies the most promising directions for future work. Two appendices supplement the main text. Appendix A provides the complete EFGDL language reference, and Appendix B describes the structural validity analyzers used as tool-independent ground truth in the benchmark evaluation.

Chapter 2: From Cells to Reads: The Technical Structure of Single-Cell Libraries

2.1 The Shift to Single-Cell Resolution

Bulk RNA sequencing treats a tissue sample as a homogeneous mixture. Every mRNA molecule in the lysate, regardless of which cell it came from, is pooled, amplified, and sequenced together. The resulting count matrix, with one row per gene and one column per cell, reports only one number per gene. This is essentially an average expression across every cell in the sample. This averaging is not just an approximation; it actively hides biological structure [15].

For example, a bulk count for the *CD8A* gene measures the mixture of near-zero expression in the majority of cells and high expression in the small $CD8^+$ T-cell population, collapsing biologically meaningful heterogeneity to a single number [15, 16].

Single-cell RNA sequencing resolves this by measuring the transcriptional state of each cell individually. Early methods required the physical isolation of individual cells by flow cytometry or laser capture before library preparation [17]. While accurate, these methods are throughput-limited. The key innovation that drove the field to profile millions of cells per experiment was the development of *barcoded library construction*. By attaching a unique synthetic identifier to every molecule before pooling, molecules from millions of different cells can be sequenced together and demultiplexed computationally afterward.

2.2 The Problem of Technical Metadata

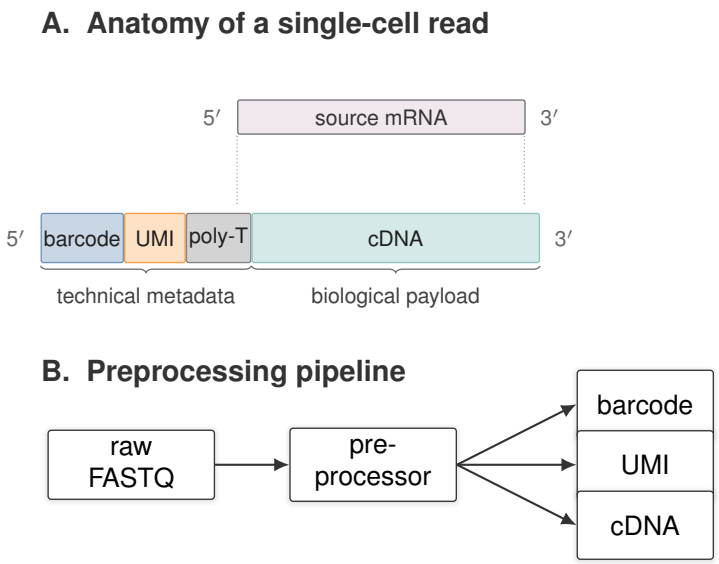

Figure 2.1: **Technical metadata structure of a single-cell sequencing read.** Panel A shows the anatomy of a typical single-cell read. Pictured is a cell barcode identifying the cell of origin, a unique molecular identifier (UMI) for PCR deduplication, a poly-T stretch from oligo-dT priming, and the cDNA payload derived from the source mRNA. Panel B shows the preprocessing task addressed by this thesis: a raw FASTQ stream is consumed by a preprocessor that separates barcode, UMI, and cDNA into distinct downstream outputs.

Every single-cell sequencing read contains two distinct sequence categories. The first is the *biological payload*, which is the cDNA derived from the original mRNA molecule. The second is the *technical metadata*. These metadata sequences have no biological origin. They are synthetic oligonucleotides designed by the assay protocol and physically ligated onto the library molecules during preparation. They must be identified, validated, and stripped before any biological analysis can begin.

2.2.1 Cell Barcodes

A cell barcode is a short synthetic DNA sequence, typically 8 to 16 bases long, whose sequence identity encodes the specific cell from which a molecule originated. Because all molecules from a single cell receive the same barcode during library preparation, downstream computational tools can group billions of sequenced reads back into their respective single-cell expression profiles.

2.2.2 Unique Molecular Identifiers

A unique molecular identifier (UMI) is a short random sequence, typically 8 to 12 bases, attached individually to each cDNA molecule before any amplification steps occur [18]. The purpose of the UMI is to correct for PCR amplification bias. Transcript abundance varies continuously, ranging from thousands of copies for housekeeping genes to only one or two copies for lowly expressed transcripts. During library preparation, a single abundant transcript might be amplified 1,000-fold, while a rare transcript might only be amplified 10-fold. Without UMIs, this stochastic difference in PCR efficiency is indistinguishable from true biological variation. Because the UMI sequence is assigned randomly at the single-molecule level, all PCR duplicates of the same original molecule will share the exact same UMI. Downstream quantification tools [19, 20, 21] use this to collapse all reads with matching barcodes, gene alignments, and UMIs into a single count, replacing a biased read count with an accurate molecular count.

2.3 The Evolution of Geometric Complexity

While the functional goals of barcoding are universal, the *physical layout of these elements on a sequencing read varies drastically between chemistries*. The design space of single-cell protocols

spans several independent axes of geometric complexity.

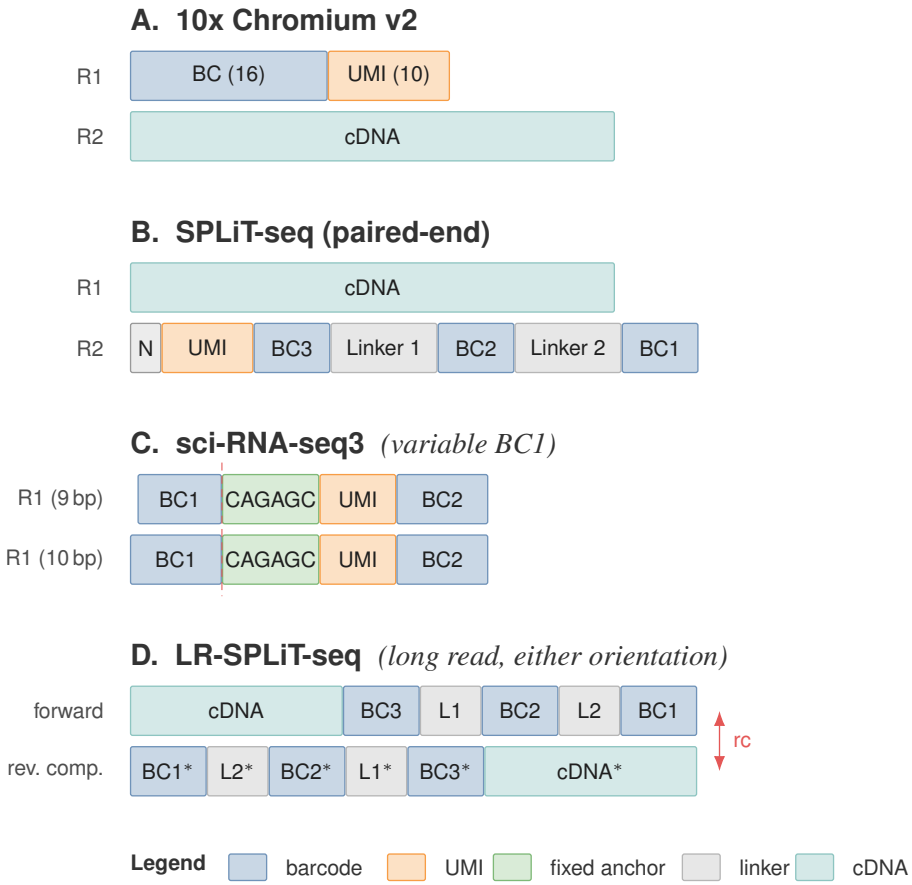

Figure 2.2: **Read geometry across four benchmarked single-cell chemistries.** The four protocols span three independent axes of geometric complexity. Seen here are the number of barcode levels and intervening linkers, the presence of variable-length elements requiring anchor-relative extraction, and the need for orientation-aware search. **Panel A:** 10x Chromium v2 has fixed 16 bp cell barcode and 10 bp UMI on R1, cDNA on R2. **Panel B:** SPLiT-seq paired-end encodes three levels of barcode separated by linkers on R2. **Panel C:** sci-RNA-seq3 has a variable-length first barcode (9 or 10 bp) resolved by the CAGAGC anchor; the dashed line marks the anchor-aligned column. **Panel D:** LR-SPLiT-seq reads are generated without strand selection, so the barcode cassette may appear in either the forward orientation or its reverse complement.

2.3.1 Level 1: Fixed Geometry

The simplest read structure is produced by droplet-based systems like 10x Chromium v2 [7]. Individual cells are encapsulated in nanoliter emulsions with a single barcoded bead. The resulting

library features a fixed 16-base cell barcode at the start of Read 1, followed immediately by a fixed 10-base UMI, with Read 2 carrying the cDNA payload. The positions of every element are deterministic. A preprocessing tool only needs to slice the read at hardcoded byte offsets, a process that is fast and simple.

2.3.2 Level 2: Combinatorial Barcodes and Linkers

Combinatorial indexing platforms, such as SPLiT-seq [22], distribute barcode information across multiple levels to achieve scale without microfluidics. Cells are cycled through successive rounds of split-pool barcoding in 96-well plates. In SPLiT-seq, Read 2 encodes a three-level barcode structure separated by linkers: short, fixed DNA sequences appended by the ligase enzyme during each round. These linkers serve as positional anchors, but ligation chemistry is prone to single-base insertions or deletions. An indel at a ligation junction shifts every downstream element. A tool relying on hardcoded byte offsets will fail here; extracting a barcode that begins at position 21 instead of position 20 silently corrupts the cell-of-origin annotation.

2.3.3 Level 3: Variable Lengths and Anchor Search

The sci-RNA-seq3 protocol [5] introduces variable-length geometry. The first barcode is either 9 or 10 bases depending on the indexing plate used. Consequently, the correct approach is to locate the fixed 6-base *anchor sequence* (`CAGAGC`) that follows the barcode, and then infer the barcode length from the anchor's position via *anchor-relative extraction*. Because the anchor itself may contain sequencing errors, this requires approximate string matching.

2.3.4 Level 4: Strand Orientation in Long Reads

Lastlty, long-read chemistries introduce a complication that is absent in short-read protocols, arbitrary strand orientation. PacBio HiFi reads from the LR-SPLiT-seq protocol [23, 24] do not enforce strand selectivity during library preparation. This means the barcode cassette can appear in either the forward orientation or its reverse complement. A tool that searches only the forward strand will recover roughly half the valid reads and will fail silently on the remainder.

The progression from fixed-length single barcodes to multi-level combinatorial barcodes, variable-length anchor-resolved sequences, and arbitrary strand orientations defines the design space that an expressive tool (possibly, language) must cover. The next chapter examines the existing computational tools that occupy this space and explains why legacy approaches struggle to handle the full scope of modern geometric complexity.

Chapter 3: Prior Work: Evaluating Legacy Preprocessing Architectures

3.1 The Parsing Challenge

The preprocessing task appears straightforward at this point. First, identify the barcode and UMI, then discard the protocol-specific scaffolding, and output the biological payload. If sequencing were error-free, this would be a trivial string-slicing operation. However, biological strings are noisy, and the error models of modern sequencing chemistries interact poorly with naive extraction strategies.

Substitution errors represent the simplest case. A tool can tolerate them by comparing a read window against an expected sequence and accepting the match if the Hamming distance remains within a predefined threshold. This is well-understood and efficiently implementable. The real difficulty arises from insertions and deletions (indels).

As established in Chapter 2, an indel displaces every downstream element. This displacement is not a theoretical edge case but instead a routine failure mode. Ligation chemistry, used in SPLiT-seq to assemble combinatorial barcodes, introduces indels at the ligation junctions at a substantial rate [22, 25]. Similarly, PacBio HiFi sequencing exhibits an inherent per-base indel error rate of approximately 0.1% [12]. Consequently, a standard 30-base linker sequence has a roughly 3% chance of containing a frame-shifting error ($1 - (10.001)^{30} \approx 0.03$). At the scale of tens or hundreds of millions of reads per experiment, failing to account for these shifts results in the silent corruption or outright loss of massive amounts of valid biological data.

The formal problem we must solve is approximate string matching under the full edit-distance metric. The software must locate multiple structural patterns of varying lengths, relative to one another, at massive throughput. Classical dynamic programming solutions exist for edit-distance matching, including the Wagner-Fischer algorithm [26], Ukkonen's banded approach [27], and Myers' bit-vector algorithm [28]. However, the engineering challenge lies in integrating these algorithms into a pipeline that is both computationally efficient and flexible enough to handle diverse protocol geometries.

3.2 Architectural Limits of Existing Tools

Tools designed for barcode extraction generally fall into three architectural categories. Each makes a distinct engineering trade-off that ultimately limits its applicability to the full spectrum of modern single-cell protocols.

3.2.1 Fixed-Position Extractors

Tools such as READSTRUCTURE [29], UMITOOLS [30], and the coordinate-based mode of SPLITCODE [25] extract elements at hard-coded byte offsets. The CELLRANGER pipeline from 10x Genomics heavily relies on this approach.

For deterministic structures like 10x Chromium v2, this is the optimal architecture. Position-based slicing is an $O(1)$ operation, allowing these tools to achieve maximum possible throughput. However, this architecture is profoundly brittle. A single-base indel upstream of any element instantly invalidates every subsequent fixed-position assumption. Furthermore, protocols with variable-length elements, such as sci-RNA-seq3, break this model entirely. A fixed-position tool requires a single, static integer offset; it provides no mechanism to evaluate alternative starting coordinates or anchor

relative to a discovered motif.

3.2.2 Regular Expression Engines

Tools like INTERSTELLAR [31] utilize regular expressions to describe read geometry. The primary advantage of this approach is familiarity; the syntax is universally understood and highly expressive for simple, deterministic patterns.

However, standard regular expressions face massive limitations at scale. First, most regex-based implementations lack the application-level parallelism required to process millions of reads efficiently. Second, POSIX and PCRE standards were not designed for approximate matching. To accept a sequence with up to k edit-distance errors, the user must explicitly enumerate every possible erroneous variant of the pattern. For a 30-base linker with up to 3 errors, this results in a combinatorial explosion. Finally, the backtracking behavior of non-deterministic finite automaton (NFA) regex engines can degrade to exponential time complexity when processing the highly ambiguous, multi-wildcard patterns required by complex chemistries such as SPLiT-seq.

3.2.3 Heuristic and Path-Based Search

A more advanced paradigm is found in SPLITCODE [25] when utilized outside of its rigid coordinate mode. This architecture treats the read as a series of possible paths between known "tag" sequences (such as expected barcodes or linkers). By searching for these tags within a defined positional range and connecting them based on a configuration graph, it can handle combinatorial indexing and local positional shifts more robustly than a regex engine.

While significantly more flexible than fixed-position tools and often faster than imperative DSLs,

this architecture remains highly specialized. Its heuristic search is heavily optimized for the split-pool protocols it was designed to parse. Adapting this path-based logic to highly unconventional geometries or addressing the dual-strand orientation required by long reads requires either manual software updates or overwhelmingly complex configuration graphs, or more bespoke pipelines to merge execution branches making it difficult to generalize.

3.2.4 Imperative Domain-Specific Languages

The imperative DSL approach, exemplified by MATCHBOX [32], solves many of the problems mentioned above. It provides a specialized pattern-matching language featuring explicit control flow, native edit-distance tolerances, and a highly optimized Rust implementation. It successfully navigates variable-length geometries by allowing the user to write separate execution branches for different barcode lengths, and it achieves processing speeds competitive with native C++ implementations.

The critical limitation of this architecture is configuration scaling. The DSL forces the user to explicitly define the procedure for matching the read, rather than simply describing the read's geometry. For example, the MATCHBOX specification for SPLiT-seq requires approximately 20 lines of imperative control flow with explicit output statements mapped to each branch. If a user wishes to make the specification orientation-aware for long-read data (such as LR-SPLiT-seq), they must duplicate the entire script, manually apply reverse-complement operations to every element, and manage the resulting logical forks. As the underlying chemistry grows more complex, the configuration burden scales non-linearly, making the tool hostile to rapid protocol iteration.

3.3 The Capability Gap

The landscape of existing tools reveals a persistent capability gap. Fast tools lack the flexibility to handle indels or variable geometries. Expressive tools lack the throughput required for industrial-scale datasets. Powerful DSLs handle the computational complexity but push the burden of algorithmic routing entirely onto the user.

To resolve these architectural limitations, we require a framework that separates the biological intent from the execution mechanics. We introduce SEQPROC, a declarative, general-purpose sequence pre-processing tool. By allowing the user to state strictly *what* the read architecture looks like, SEQPROC, along with its backend ANTISEQUENCE, automatically compiles the geometry into an execution plan. SEQPROC dynamically selects the optimal string-matching algorithm, providing both high-throughput execution and robust error tolerance without requiring an imperative definition.

Chapter 4: SEQPROC: A Declarative Approach to Read Preprocessing

4.1 The Declarative Paradigm

4.1.1 The Configuration Burden

As established in earlier chapters, the shift from hardware-driven microfluidics (like 10x Chromium) to complex biochemical routing (like SPLiT-seq and sci-RNA-seq3) drastically altered the geometry of sequencing reads. The simple, fixed-coordinate structures of the past have been replaced by highly interspersed reads featuring variable-length barcodes, error-prone ligation linkers, and arbitrary strand orientations.

At the level of a single read, however, the fundamental preprocessing task remains unchanged. A tool must locate the relevant elements (cell barcodes, UMIs, and the cDNA payload), validate them against known references, extract them into a standardized format, and discard the structural scaffolding. The goal is to reduce the massive diversity of protocol-specific geometries into a single, clean format that any downstream quantification tool can consume.

This framing exposes the primary architectural deficiency of legacy imperative tools: they conflate the *specification* of what to keep with the *implementation* of how to find it.

For example, a SPLITCODE configuration for SPLiT-seq must enumerate all 96 valid sequences for each of the three barcode rounds. This results in 288 lines of configuration code just to define

the barcode vocabulary before a single extraction rule is written. Similarly, a MATCHBOX script for sci-RNA-seq3 must contain two explicit, duplicated branches to handle the 9 bp and 10 bp barcode variants, because the language lacks the vocabulary to express ”either length” in a single statement. In both cases, the configuration forces the user to describe the mechanical procedure of matching rather than the geometric reality of the read. This results in configurations that are verbose, difficult to audit, and highly resistant to rapid protocol iteration.

4.1.2 Declarative Specifications are Succinct

The central claim of SEQPROC is that a *declarative* specification (one that states the geometry of a read without prescribing the matching algorithm) is significantly shorter, more maintainable, and less error-prone than an imperative one.

To make this concrete, Figure 4.1 shows the complete configuration needed to preprocess sci-RNA-seq3 reads across three different paradigms.

The SEQPROC specification handles both barcode lengths with a single `b[9-10]` interval; the variable length is resolved automatically using the anchor as a right-side delimiter. MATCHBOX requires the user to duplicate the entire pattern block. SPLITCODE requires a separate `@extract` directive for each field and relies on an indirect coordinate system that is difficult to read without deep familiarity with the tool’s specialized syntax. Declarative specifications are succinct because they allow the user to state biological *intent* (e.g., ”a barcode of 9 or 10 bases”) and delegate the complex case analysis entirely to the compiler.

A. SEQPROC (8 lines)

seqproc.geom

```
# definitions
#[edit(1)] anchor = f[CAGAGC]
brc1 = norm(b[9-10])
brc2 = b[10]
umi  = u[8]

# read structure
1{<brc1><anchor><umi><brc2>}
2{r<read>:}
-> 1{<brc1><brc2><umi>}2{<read>}
```

B. MATCHBOX (18 lines)

matchbox.mb

```
# definitions
primer = CAGAGC

if read.r1 matches {
  [ bc1:|9| primer~0.17 umi:|8| bc2:|10| _ ] =>
    { bc1.concat(bc2.concat(umi))
        .out!('mb_r1.fq')
      read.r2
        .out!('mb_r2.fq')
    }
  [ bc1:|10| primer~0.17 umi:|8| bc2:|10| _ ] =>
    { bc1.concat(bc2.concat(umi))
        .out!('mb_r1.fq')
      read.r2
        .out!('mb_r2.fq')
    }
}
```

C. SPLITCODE (8 lines + ext)

splitcode.config

```
@extract <splitcode_sci_rna_seq3[9-10]>{{anchor}}
@extract {{anchor}}8<splitcode_sci_rna_seq3[10]>
@extract {{anchor}}<splitcode_sci_rna_seq3[8]>
group   tag     distance  next  maxFindsG  location
anchor  CAGAGC  1         -     1          0:9:16
```

Figure 4.1: **Tool configuration comparison for sci-RNA-seq3 [5].** All three configurations produce equivalent output but vary significantly in verbosity and clarity. **(A)** SEQPROC's declarative `b[9-10]` interval natively handles variable barcode lengths in a single, concise statement. **(B)** MATCHBOX requires an imperative approach, forcing two explicit branches for the variable length and resulting in duplicated logic. **(C)** SPLITCODE relies on an indirect `@extract` coordinate mechanism requiring a separate directive for each field.

4.2 The Extended Fragment Geometry Description Language

4.2.1 Design Goals

EFGDL was designed with two co-equal goals. First, it must *specify* a sequencing protocol completely enough that the tool can validate, match, and transform reads without further algorithmic instruction. Second, it must *document* that protocol in a format that a biologist can easily read and a bioinformatician can rapidly verify. Both goals require the language to be concise, unambiguous, and close to the standard vocabulary utilized in genomics. A third design goal was *extensibility without backward incompatibility*. New sequencing technologies routinely introduce novel matching requirements, including orientation-aware processing for long reads and complex edit distance tolerances for newer flow cell chemistries. EFGDL accommodates these through an *annotation* system (detailed fully in Appendix A) that attaches specific matching behaviors to individual interval definitions. Adding a new annotation type does not alter existing specifications.

Furthermore, annotations draw a clean boundary between *sequential* processing (the left-to-right positional geometry) and *non-sequential* matching behaviors. Operations like orientation reversal, sliding-window searches, and approximate matching via `#[edit]` or `#[hamming]` are non-sequential; they require inspecting an interval in multiple configurations. Factoring these behaviors into annotations keeps the read-structure block purely declarative while accommodating chemistries that require fuzzy, error-tolerant logic.

4.2.2 Language Structure

An EFGDL specification comprises three parts, though only the middle section is strictly required.

The first is a *definitions block*, where named intervals are declared with a specific label and an interval expression. These names serve as readable handles that can be referenced later and carry any annotations that dictate how the interval is matched.

The second is the *read structure*, where those named intervals are chained within `1{...}` and `2{...}` blocks to describe the physical layout of Read 1 and Read 2.

The third is an optional *transformation clause* introduced by the `->` operator, which defines the desired output geometry. Intervals can be reordered, removed, padded, or corrected within this clause. Crucially, any interval not explicitly included in this clause is safely discarded.

The idiomatic style is to give every interval a meaningful name in the definitions block, use those names in the read structure to make the geometry self-documenting, and apply any necessary format normalizations in the output clause.

4.2.3 A Progression of Examples

We will build an understanding of EFGDL through examples drawn from the four protocols benchmarked in Chapter 6. These span the full range of geometric complexity, progressing from trivial fixed-length structures to multi-level combinatorial barcodes requiring fuzzy anchor search and orientation-aware matching.

4.2.3.1 The Simple Case (10x Chromium v2)

The 10x Chromium v2 short-read protocol places a fixed 16 bp cell barcode followed by a fixed 10 bp UMI at the start of Read 1, while Read 2 carries the cDNA. The complete EFGDL specification fits in six lines. A comprehensive EFGDL language reference is available in Appendix A.

10x Chromium v2

```
bc  = b[16]   # 16 bp cell barcode (Gel Bead Oligonucleotide)
umi = u[10]   # 10 bp unique molecular identifier
bio = r:      # remainder of R2 is the cDNA payload

1{<bc><umi>} 2{<bio>}
-> 1{<bc><umi>} 2{<bio>}
```

The read structure states that Read 1 is exactly a 16 bp barcode concatenated with a 10 bp UMI, and Read 2 is an unbounded interval (`r:`), meaning it matches all remaining bases to the end of the read. The transformation clause here is the identity operation, listing the exact same intervals in the same order as the read structure. Its presence is deliberate, in that any interval omitted from the output clause is silently dropped, so an explicit clause makes the output contract highly visible. Because of its simplicity, a molecular biologist familiar with 10x Chromium can read this specification and immediately verify it against official protocol documentation.

4.2.3.2 Variable-length Barcodes and an Anchor (sci-RNA-seq3)

sci-RNA-seq3 [5] introduces positional variability. Its first barcode (labeled `brc1`) is either 9 or 10 bp depending on the indexing plate used. A rigid fixed-position extractor will fail for any read containing the unexpected variant. EFGDL handles this natively with a variable-length interval and an anchor that pins the boundary.

```
sci-RNA-seq3
#[hamming(1)] anchor = f[CAGAGC]  # 6 bp ligation anchor; 1 mismatch tolerated
brc1 = norm(b[9-10])              # variable barcode, padded to 10 bp for output
brc2 = b[10]                      # second barcode, fixed 10 bp
umi  = u[8]                       # 8 bp UMI

1{<brc1><anchor><umi><brc2>}
2{r<read>:}
-> 1{<brc1><brc2><umi>} 2{<read>}  # reorder: both barcodes before UMI
```

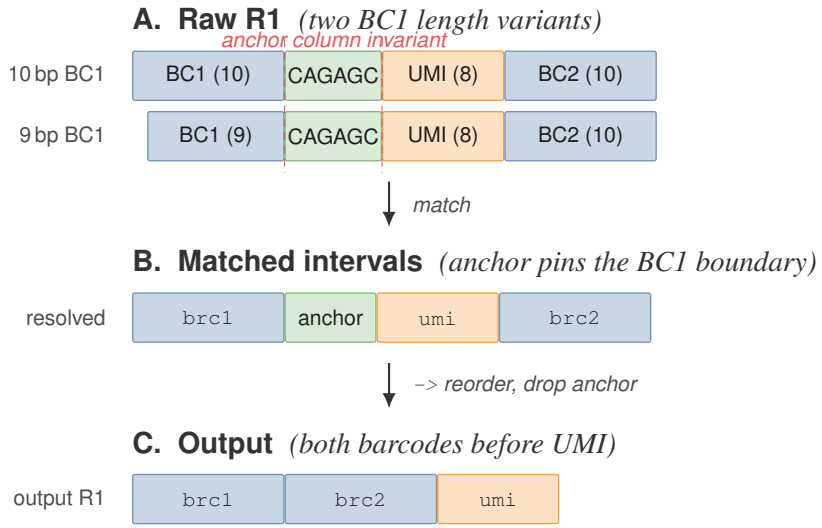

Figure 4.2: **sci-RNA-seq3 read layout and SEQPROC transformation.** Panel A shows the two BC1 length variants (9 bp and 10 bp) stacked with their CAGAGC anchor columns aligned (dashed line): only the left boundary of `brc1` shifts between the variants, while the positions of the downstream anchor, UMI, and `brc2` remain invariant relative to the anchor. Panel B shows the matched intervals after SEQPROC processing, with the anchor pinning the `brc1` boundary regardless of length. Panel C shows the output read after the transformation clause reorders both barcodes before the UMI and silently drops the anchor, yielding a uniform structure for downstream quantification.

The anchor is declared as `CAGAGC` and annotated with `#[hamming(1)]`. This instructs the compiler to accept the sequence even if one base differs due to a sequencing error. Because the anchor must appear immediately after `brc1`, SEQPROC dynamically recovers the correct barcode length. The `norm()` transformation pads shorter matches to a canonical width so that downstream tools receive a

uniform field. Finally, the output clause reorders the structure to place both barcodes before the UMI, matching the standard format expected by downstream quantification tools.

4.2.3.3 Linkers and Whitelist Filtering (SPLiT-seq paired-end)

SPLiT-seq [22] heavily relies on complex linker sequences. Because sequencing errors frequently corrupt these ligation junctions, absolute coordinate extraction is unreliable. The annotation `#[search(relative)]` instructs SEQPROC to scan the read for the best-matching linker position rather than assuming a rigid offset. Stacking `#[edit(6)]` on the same definition allows the engine to tolerate up to six substitutions or indels, effectively absorbing the ligation artifacts common to this protocol [25].

SPLiT-seq paired-end

```
read1 = r:                                          # R1 is the cDNA payload
skip2 = x[2]                                        # 2 bp ligation artifact
umi   = u[10]                                       # 10 bp UMI
bc3   = filter_within_dist(b[8],
            'r3.txt', 1)                            # Round-3 barcode whitelist

#[search(relative)]                                 # Search relative to the read
#[edit(6)]                                          # (up to 6 edits)
l1    = f[GTGGCCGCTGTTTCGCATCGGCGTACGACT]           # Linker 1
bc2   = filter_within_dist(b[8],
            'r2.txt', 1)                            # Round-2 barcode whitelist
#[search(relative)]
#[edit(6)]
l2    = f[ATCCACGTGCTTGAGAGGCCAGAGCATTCG]           # Linker 2
bc1   = filter_within_dist(b[6],
            'r1.txt', 1)                            # Round-1 barcode, 6 bp
rest  = r:                                          # Trailing sequence (discarded)

1{<read1>}                                          # Capture all of read 1
2{<skip2><umi><bc3><l1><bc2><l2><bc1><rest>}        # Put the read structure together
-> 1{<read1>} 2{<umi><bc3><bc2><bc1>}              # Drop linkers; canonical order
```

The `filter_within_dist` keyword combines extraction with validation, accepting a barcode only if it matches a known entry in the whitelist within a Hamming distance of 1. Any read with a failing barcode is safely filtered. The discard interval `x[2]` silently absorbs the two leading bases that appear as artifacts of the chemistry. The transformation completely drops both linkers and the trailing sequence.

Despite being the most structurally complex short-read protocol, the SEQPROC specification remains fully declarative. The user never writes a sliding-window loop or manages a distance calculation.

4.2.3.4 LR-SPLiT-seq: Long Reads with Orientation Awareness

PacBio reads from the LR-SPLiT-seq protocol [23] present a unique complication, that is the barcode cassette can appear in either strand orientation. A preprocessing tool that searches only the forward strand will miss roughly half of all valid biological data. EFGDL natively handles this via the `#[match_ori(either)]` read-level annotation.

LR-SPLiT-seq (PacBio HiFi)

```
umi      = u[10]                                  # 10 bp UMI
bc3      = b[8]                                   # round-3 barcode
#[search(relative)]                               # Search relative to linker
#[edit(6)]                                        # Up to 6 edits
linker_a = f[GTGGCCGATGTTTCGCATCGGCGTACGACT]      # linker A
bc2      = b[8]                                   # round-2 barcode
#[search(relative)]
#[edit(4)]                                        # Up to 4 edits
linker_b = f[ATCCACGTGCTTGAGACTGTGG]              # linker B
bc1      = b[8]                                   # round-1 barcode
rest     = r:                                     # trailing bases (discarded)

                                                  # Single read structure
#[match_ori(either)]                              # (try both orientations)
1{<umi><bc3><linker_a><bc2><linker_b><bc1><rest>}
-> 1{<umi><bc3><bc2><bc1>}                        # Output: extracted UMI+barcodes
```

This annotation appears on the read-structure block rather than an individual interval because orientation affects the entire read simultaneously. It instructs the engine to attempt forward matching first and, if the geometry is not satisfied, automatically retry on the reverse complement. No other structural changes to the specification are required. This composability highlights the advantage of the annotation design, allowing advanced matching behaviors to be layered onto existing structures without requiring an imperative rewrite.

4.3 Architectural Separation: Specification and Execution

A key design choice in SEQPROC is the strict separation between the *language layer* (EFGDL) and the *execution layer* (ANTISEQUENCE).

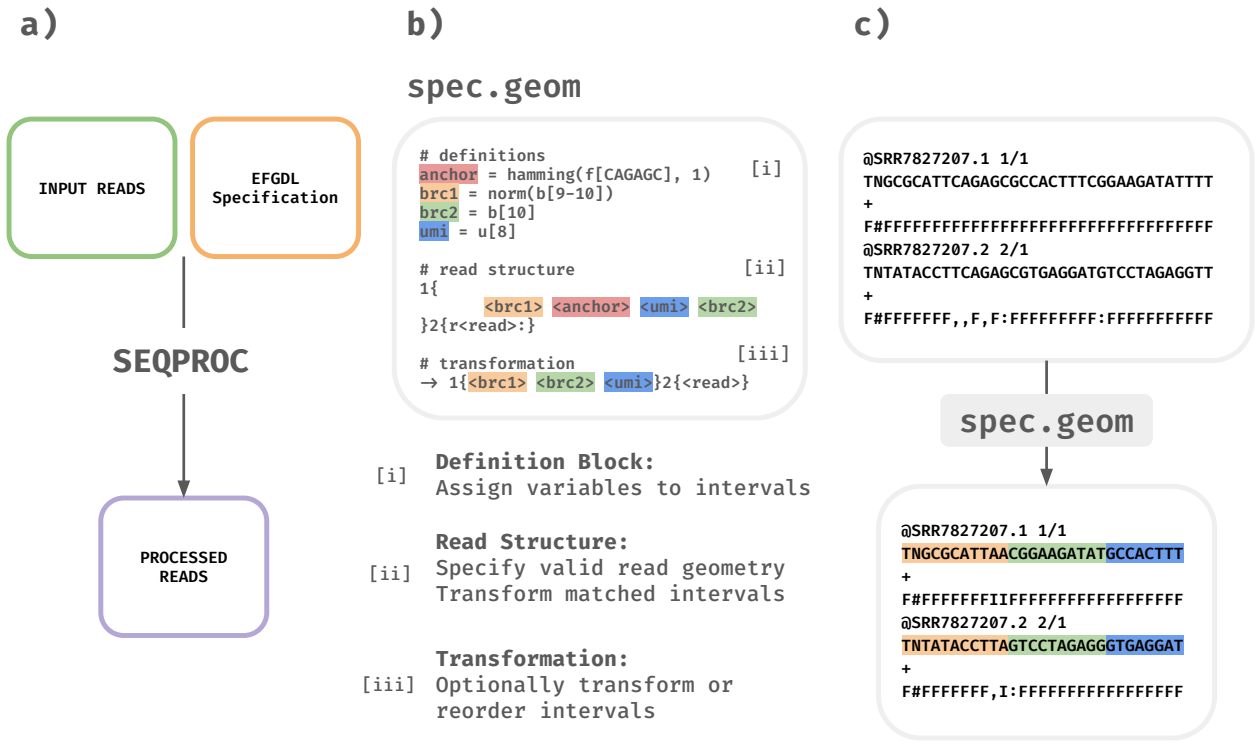

Figure 4.3: **SEQPROC workflow overview † (a)** SEQPROC accepts reads in FASTQ format (single- or paired-end) together with an EFGDL geometry specification. The specification is compiled into an ANTISEQUENCE execution graph, and matching reads are streamed to output. **(b)** An idiomatic EFGDL specification for sci-RNA-seq3 [5]: named intervals are declared in the definitions block, arranged in physical read order in the read-structure block, and reordered into the format required by downstream tools in the transformation clause. **(c)** Two example reads processed by the specification. The anchor `CAGAGC` resolves the length ambiguity in `brc1` (9 or 10 bp); the transformation reorders both barcodes before the UMI in both cases.

An EFGDL specification is initially parsed by the CHUMSKY [33] library. This library uses parser combinators to generate recursive descent parsers capable of processing parsing expression grammars (PEGs) [34] into a typed abstract syntax tree (AST). Chumsky collects all syntax errors in a

single pass, which is highly beneficial in practice as EFGDL is often written by biologists rather than compiler engineers.

The validated AST is then compiled into an ANTISEQUENCE directed acyclic graph (DAG) of operation nodes via a systematic, rule-driven translation. For example, a fixed-length interval becomes a `CutOp`, a fragment annotated with `#[hamming(1)]` becomes a `MatchAnyOp` configured for distance search, and the output clause translates to a sequence of `RetainOp`, `SetOp`, and `TrimOp` nodes. The compiler performs rigorous static analysis to guarantee that every label referenced in the read structure is defined and that all specifier type combinations are valid. A specification that passes these static checks will not produce a per-read architectural error at runtime. Figure 4.4 illustrates the DAG produced from the sci-RNA-seq3 specification.

The execution engine itself, ANTISEQUENCE, is an independent Rust library. Tools that require finer control than EFGDL exposes can program against its public API directly. This architectural separation ensures that performance improvements made to the engine, such as better SIMD routines, smarter batch scheduling, or alternative high-performance memory allocators like mimalloc and jemalloc, propagate automatically to every EFGDL specification without requiring any modifications to the language itself.

4.4 Chapter Summary

SEQPROC directly addresses the algorithmic bottlenecks described in Chapter 3. The Extended Fragment Geometry Description Language provides a concise, declarative notation for specifying complex read geometries. By allowing the user to state what the read looks like and what the output must contain, the compiler absorbs the burden of determining how to achieve that transformation

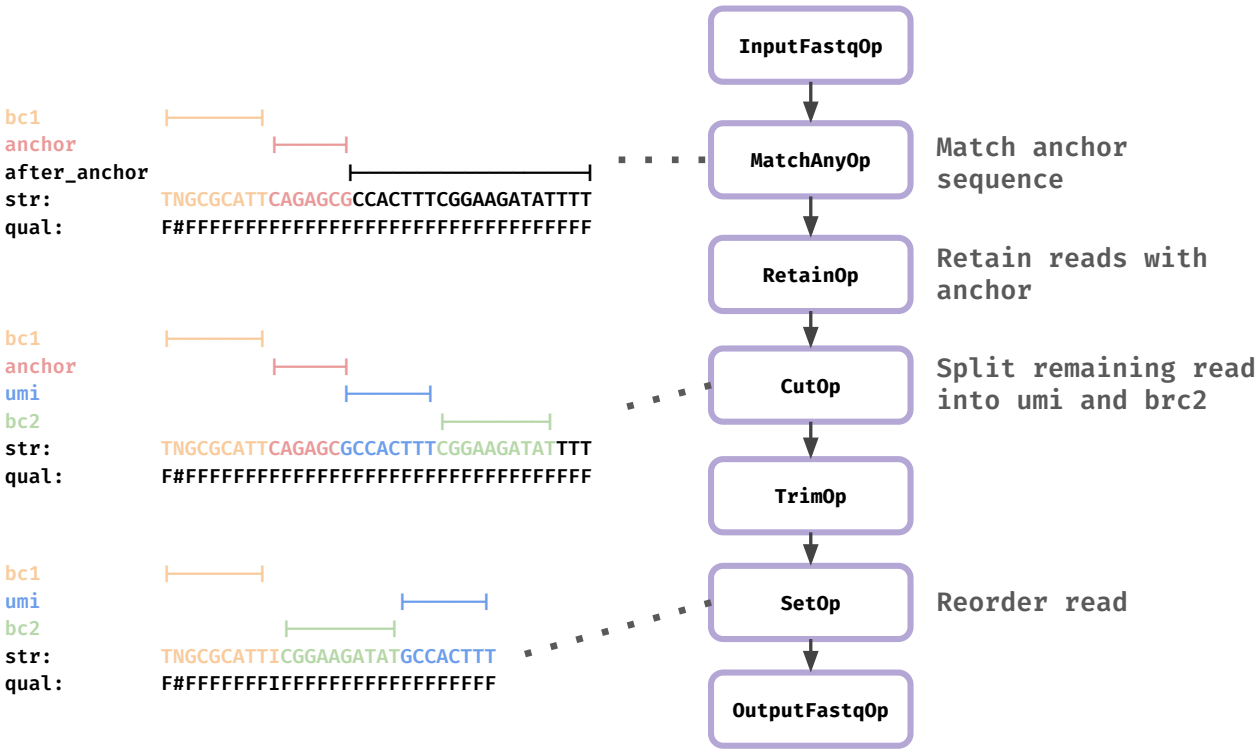

Figure 4.4: **ANTISEQUENCE execution graph for the sci-RNA-seq3 specification.** † The EFGDL specification from Figure 4.3 compiles into this graph of operation nodes. Each node processes a batch of reads and passes matching reads to its successor. Interval labels assigned by earlier nodes are referenced symbolically by later nodes, ensuring the underlying sequence bytes are never copied during matching.

efficiently.

The ANTISEQUENCE execution engine, detailed fully in Chapter 5, provides the robust computational infrastructure that makes this declarative approach practical at industrial scale. The strict separation between the language layer and the execution layer ensures that SEQPROC can dynamically utilize the most appropriate matching algorithms based solely on the user's geometric annotations.

Chapter 6 evaluates SEQPROC empirically against legacy tools on four complex single-cell chemistries. The benchmarks demonstrate that SEQPROC is the fastest tool on three of the four datasets (1.2× to 13.6× faster than the next-best alternative) and requires the least memory in every evaluated scenario. Furthermore, it achieves this computational efficiency while matching or exceeding the valid read recovery rates of its imperative competitors.

Chapter 5: Under the Hood: The ANTISEQUENCE Execution Engine

ANTISEQUENCE is the standalone Rust library that executes every EFGDL specification compiled by SEQPROC. It is also independently usable by tools that require finer programmatic control than a geometry file allows, enabling developers to construct and run ANTISEQUENCE graphs directly through its public API. This chapter details the core engineering decisions underlying the performance and correctness guarantees demonstrated in the benchmark evaluations in section 6.5.

5.1 From Syntax to Execution

An EFGDL geometry file is transformed into a running ANTISEQUENCE graph through a three-phase compilation pipeline. Each phase operates on a distinct intermediate representation and is designed to catch specific classes of errors before any biological data is processed.

The first phase tokenizes the EFGDL source into a flat sequence of typed tokens such as identifiers, integer and string literals, bracket pairs, annotation markers (`#[...]`), and keywords. The lexer is character-level and context-free; it never peeks ahead, ensuring that tokenization errors are reported with exact byte offsets regardless of downstream syntax. This simplicity is deliberate. The grammar is regular at the token level, and pushing complexity into the lexer would complicate error recovery without meaningfully improving expressiveness.

The second phase builds a typed abstract syntax tree (AST) from the token stream using the

CHUMSKY parser-combinator library [33]. A parser combinator encodes each grammar rule as a composable function that returns either a parsed value or an error. CHUMSKY extends this model with robust error recovery, continuing to parse after a failure by dynamically inserting or skipping tokens. This allows the compiler to collect all syntax errors simultaneously rather than halting at the first failure. For a specification language like EFGDL, where a single dropped bracket in a transformation clause might otherwise suppress dozens of downstream diagnostics, this collect-all behavior heavily simplifies the debugging of specifications.

The AST produced by the parser comprises seven node types: definition entries, read-structure blocks, transformation clauses, annotation nodes, filter expressions, map expressions, and literal sequences. Each node securely retains its source span for accurate downstream error reporting.

The third phase lowers the AST into an ANTISEQUENCE directed acyclic graph (DAG) of operation nodes. The translation is rule-driven and highly systematic. For example, a fixed-length interval `b[16]` maps to a `CutOp`; a fragment annotated with `#[hamming(1)]` maps to a `MatchAnyOp` configured for single-mismatch search; a `filter_within_dist` transformation maps to a `SelectOp` that drops reads failing the whitelist lookup; and the output clause translates to a sequence of `RetainOp`, `TrimOp`, and `SetOp` nodes.

The `#[match_ori(either)]` read-level annotation wraps the inner geometry graph in a `TryOrientationOp`. This node runs the full inner graph on the forward read and, upon failure, reverse-complements the read in place and retries. Because the reverse-complement is applied directly to the input buffer before executing the standard geometry, the search semantics, linker ordering, and whitelist checks remain identical in both orientations. This structural property guarantees that intervals extracted from a reverse-complemented read are always reported in forward coordinates, an absolute requirement for correct downstream UMI deduplication.

Compilation also performs rigorous static analysis to enforce type rules. Every label referenced in the read-structure block must resolve to a definition, every specifier and transformation combination must be type-correct, and the right-anchoring rule must be satisfied for every variable-length interval. A specification that passes all three phases of compilation is guaranteed never to produce a per-read structural parse error at runtime.

5.2 The Zero-Copy Memory Abstraction

The dominant computational cost in a naive streaming read processor is memory allocation. Each input read typically triggers the construction of a fresh data structure containing multiple dynamically allocated vectors. At the scale of 100 million reads per dataset, this translates to hundreds of millions of `malloc` and `free` cycles before a single byte of biological payload has been evaluated.

ANTISEQUENCE eliminates this bottleneck through a batched read recycling model. Rather than allocating a fresh `Read` struct for each record, the execution graph passes a `Vec<Read>` from the output of one batch back to the input of the next. The input operator refills the existing `Read` objects by clearing their contents while retaining their allocated buffer capacity. After the first batch of reads is processed, no further heap allocations occur for the remainder of the run. Memory usage becomes $O(1)$ with respect to the total number of input reads.

The `Read` struct itself is engineered to minimize heap pressure. Internally, each `Read` contains typed strings corresponding to the FASTQ record fields, and each string maintains a collection of labeled intervals (referred to as "mappings"). These mappings define subregions via a start offset, an end offset, and an attribute map. Graph operations manipulate these boundary offsets rather than copying or slicing the underlying sequence bytes. The actual bytes are materialized only when the

final output operator serializes the matched intervals into the destination FASTQ file.

The mapping collection utilizes the `SmallVec` data structure, which holds up to four mappings inline without triggering a heap allocation. For the vast majority of read geometries, including all benchmarked protocols, the number of labeled intervals per read rarely exceeds four, meaning the common case is entirely stack-allocated. This optimization alone yielded a consistent 10% throughput improvement during engine profiling.

5.3 Hardware Acceleration and the Block Aligner

Approximate string matching forms the computational core of ANTISEQUENCE. The processing cost scales directly with the chosen distance metric, the pattern length, and the search space. The engine supports both Hamming distance (substitutions) and edit distance (substitutions, insertions, and deletions), dynamically selecting the underlying search algorithm based on the requested tolerance and pattern constraints.

Hamming distance. Hamming distance matching employs a bitwise lookup table that encodes each pattern as a 64-bit bitmask. For patterns up to 8 bytes, a single XOR and popcount instruction can compute the distance against any 8-byte read window. An early-exit scan halts immediately once a position within the requested tolerance is found. This highly optimized approach is $O(n)$ relative to read length and requires no heap allocation beyond the initial lookup table construction.

Myers' bit-vector algorithm for edit distance. Hamming distance is insufficient for protocols where ligation chemistry introduces frame-shifting indels near linker sequences. ANTISEQUENCE addresses this using Myers' bit-vector algorithm [28] for patterns up to 64 bases. Myers' algorithm

encodes the dynamic programming (DP) column as a pair of 64-bit bitvectors, updating an entire column in $O(1)$ time using minimal bitwise operations. For a standard 30 bp SPLiT-seq linker, the entire DP column fits in a single word. This continuous $O(n)$ pass avoids the constant window restarts of a naive Hamming scan, producing a measured 2.3$\times$ speedup over the Hamming-distance baseline on the LR-SPLiT-seq dataset.

Block-aligner for longer patterns. For structural patterns exceeding 64 bases, ANTISEQUENCE falls back to the `block-aligner` library [35]. This library provides SIMD-accelerated banded dynamic programming, utilizing AVX2 instructions on x86-64 architectures and NEON instructions on ARM/aarch64 platforms. While current benchmark protocols do not trigger this path due to shorter linker lengths, this fallback ensures that EFGDL specifications remain compatible with emerging long-read chemistries featuring extended structural motifs.

Adaptive search strategy. For scenarios involving four or more active patterns, exhaustive Myers' search becomes inefficient. ANTISEQUENCE employs an adaptive seeded search based on the pigeonhole principle. Explicitly stated, if a match within edit distance d exists, at least one of $k = \lceil (m-d)/(d+1) \rceil$ non-overlapping k-mers from the pattern must appear in the text exactly. A small k-mer index rapidly identifies candidate start positions for subsequent Myers' verification. The necessity of this adaptive selection is evident when benchmarking naive implementations; applying seeded search unconditionally to the LR-SPLiT-seq dataset resulted in a 2.4$\times$ performance regression. Dynamically switching to the adaptive selection model recovered this performance deficit and maximized overall throughput.

5.4 Thread Pooling and Batch Coherence

ANTISEQUENCE parallelizes read processing through the Rayon [36] work-stealing thread pool. The input operator groups reads into discrete chunks and enqueues each chunk as a unified task. Worker threads dequeue these tasks and process the entire chunk through the full DAG before requesting additional work, ensuring the amortized synchronization cost per read remains negligible.

The granularity of this chunking interacts non-linearly with thread count. While the default batch size is 512 reads, systematic profiling on the 10x dataset revealed that fine-grained batches processed across multiple threads could trigger contention regimes where the lock acquisition overhead exceeded the trivial processing time of the simple geometry. Tuning the batch size dynamically eliminates this contention, though returns diminish beyond 4 threads. We speculate that this is because the working set per thread shrinks below the CPU's L2 cache size. More detailed analysis is required to verify this claim.

Batch coherence and split batches. A structural challenge in the DAG execution model is maintaining batch coherence. Nodes with conditional outgoing edges, such as a `SelectOp` routing reads based on whitelist validation, fragment a batch across multiple downstream paths. Consequently, the effective batch size reaching terminal nodes is smaller than the initial input batch. While dynamic batch buffering at edges (accumulating reads until a critical mass is reached) represents a future optimization avenue, the current implementation performs efficiently for existing protocols where filter rates do not degrade terminal batch sizes into contention-sensitive thresholds.

Output ordering. Multi-threaded execution inherently disrupts the sequential order of reads across batches. Because many downstream quantification tools require paired FASTQ files to remain in strict

lock-step, SEQPROC provides a `--preserve-order` flag. When enabled, the output operator tags each incoming batch with its initial sequence number and utilizes a priority-queue buffering system to emit the processed batches in their original input order. In practice, because all worker threads process data at roughly equivalent rates, this reordering buffer introduces only a minor bounded latency without negatively impacting the maximum throughput of the pipeline.

Chapter 6: Results, Optimizations, and Lessons Learned

This chapter documents the empirical evaluation of the performance results reported throughout this thesis. We detail the methodology used to measure each architectural change, the experiments that succeeded, the optimizations that failed, and the quantitative lessons drawn from both.

6.1 Development Process and Iterative Optimization

6.1.1 Evaluation Methodology

Every optimization was evaluated using `Criterion.rs` [37], a statistical benchmarking framework for Rust that collects 50 to 100 samples per benchmark, reports medians with confidence intervals, and performs a Welch's t-test against a saved baseline. The workflow for each proposed change was strict:

1. Save a Criterion baseline before the change.
2. Implement the change.
3. Run the full benchmark suite against the saved baseline.
4. If any benchmark regressed beyond the noise threshold, revert the change.
5. Run the complete test suite to confirm algorithmic correctness.

6. Commit only if all benchmarks improved or held steady and all tests passed.

The benchmark suite spanned multiple scales (1 K, 10 K, 1 M, and 10 M synthetic reads) and evaluated two distinct protocol complexities: a trivial geometry (10x Chromium v2) designed to isolate per-read memory overhead, and a highly complex geometry (sci-RNA-seq3 with anchor matching and variable-length barcodes) selected to stress the pattern-matching engine. Additionally, real compressed FASTQs sourced from the European Nucleotide Archive [38] were included to prevent algorithmic overfitting to synthetic data artifacts.

To accurately attribute runtime to specific subsystems, several targeted isolation tools were integrated into the evaluation framework. A purpose-built graph node, `NullOutputOp`, was developed to accept and silently discard reads, allowing the baseline cost of the output path to be measured directly by substitution. For stricter isolation, the `ANTISEQ_STUB_OUTPUT` environment variable was introduced to disable FASTQ serialization entirely, cleanly separating raw compute cycles from I/O overhead. Finally, CPU sampling profiles were routinely generated via `samply` and analyzed within the Firefox Profiler to pinpoint hot functions prior to initiating any optimization pass.

6.1.2 Correctness Guard Rails

Performance optimization introduces strict correctness requirements. The test suite grew from 8 tests in September 2025 to 402 tests by March 2026, achieving 80.2% line coverage in SEQPROC. Key test categories included per-byte output matching, multi-thread consistency validation (ensuring 1-thread versus 4-thread outputs remain identical), distance threshold enforcement, and end-to-end verifications for all benchmarked chemistries.

6.2 Empirical Impact of Architectural Optimizations

As detailed in Chapter 5, the ANTISEQUENCE engine utilizes specific architectural paradigms to achieve high throughput. This section quantifies the empirical impact of those decisions.

6.2.1 Measuring the Zero-Copy Abstraction

The pre-optimization profile of ANTISEQUENCE was bottlenecked by heap allocation. At one million reads, flame graphs indicated that roughly 48% of CPU samples resided in condition-variable waits and allocator functions.

Implementing the batched read recycling architecture (detailed in Section 5.2) yielded the single largest performance gain of the development cycle. On the 10x geometry at 1 M reads using a single thread, the cumulative speedup relative to the pre-optimization baseline was 4.8× (1.21 s dropping to 252 ms). This improvement scaled positively with dataset size. At 10 M reads, the recycling pass yielded an additional 38% reduction in runtime beyond the already-optimized state, as the amortization of the initial buffer allocation improves at larger scales. Supporting optimizations, such as migrating internal collections to `SmallVec`, produced a consistent secondary throughput improvement of approximately 10%.

6.2.2 Measuring Synchronization and Parallelism

ANTISEQUENCE parallelizes processing by dispatching batches of reads to worker threads. A systematic sweep over chunk sizes (256, 512, 1,024) and thread counts (1, 2, 4, 8) revealed a highly non-linear interaction between batch granularity and lock contention.

While a chunk size of 1,024 reads with 2 to 4 threads proved optimal (yielding an approxi-

mate 35% improvement over single-threaded execution on 1 M reads), reducing the chunk size to 512 reads with 4 threads produced a catastrophic 2× performance regression. At 512 reads, the per-chunk processing time (roughly 125 μs for the 10x geometry) was shorter than the combined overhead of acquiring the input lock and waking from condition variables.

Scaling beyond 4 threads exhibited diminishing returns. Moving to 8 threads offered zero measurable improvement on the 10x geometry and only marginal gains on the sci-RNA-seq3 geometry. With 1,024-read chunks processing 1 M total reads, the marginal benefit of additional threads is quickly offset by cache-line invalidation across physical cores.

6.3 Evaluating the I/O Path: The Value of Negative Results

One of the most valuable experiments produced no performance improvement. The `NullOutputOp` isolation test revealed that FASTQ serialization and file output account for only 6% of the total runtime. This finding successfully redirected engineering effort away from the output path and toward the allocation and compute paths where the optimization ceiling was significantly higher.

A subsequent experiment attempted to batch multiple FASTQ records into a thread-local buffer before flushing to disk. This produced a 20 to 30% performance regression. Each record was serialized into an intermediate `Vec<u8>` before being appended to the buffer, introducing a memory allocation that did not exist in the unbatched path. The explicit flush after each batch defeated the standard library's internal buffering, which was already coalescing small writes efficiently. This illustrates a core systems principle: *batching improves performance when it amortizes a fixed per-operation cost (like a lock acquisition) but degrades performance when it adds a variable per-operation cost (like an*

intermediate buffer copy).

6.4 Quantifying Algorithmic Improvements

6.4.1 Adaptive Edit Distance Matching

Chapter 5 outlines the adaptive search strategy that dynamically selects between Myers' bit-vector algorithm and seeded k-mer search. This adaptive selection was motivated by a failure during initial benchmarking.

When seeded search was applied unconditionally to the LR-SPLiT-seq dataset (1 M read subset), the edit distance implementation was 2.4× slower than standard Hamming distance (38.7 s versus 16.2 s). The overhead of building the k-mer index completely dominated the single-linker search operation. Switching to the adaptive strategy reversed this penalty, resulting in edit distance becoming 2.3× faster than Hamming distance (2.0 s versus 4.7 s) because Myers' bit-vector processes the entire read in a single $O(n)$ pass.

On total read recovery, edit distance matched or exceeded Hamming distance across all evaluated protocols (Figure 6.1). The improvement was highly pronounced on long-read data, yielding a 15.8% gain on LR-SPLiT-seq where PacBio indels near linker regions frequently cause rigid Hamming-distance matching to fail.

6.4.2 Orientation-Aware Processing

Long reads from PacBio arrive in arbitrary orientations. The implementation of the `TryOrientationOp` node (Section 5.1) ensures the search semantics remain identical in both orientations. The impact on the LR-SPLiT-seq dataset is quantified in Table 6.1.

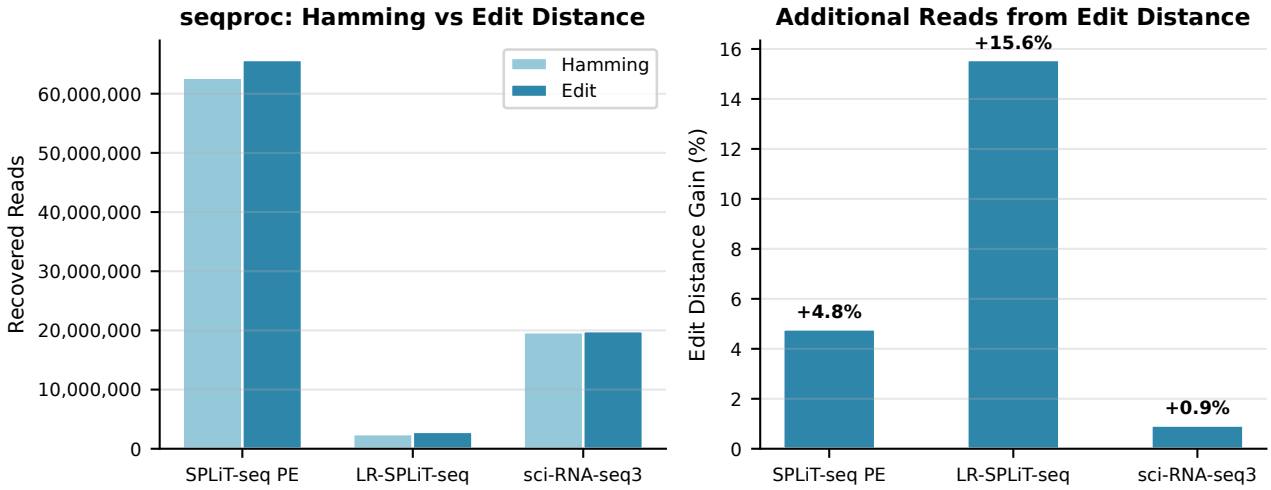

Figure 6.1: **Effect of edit distance matching on read recovery. Left:** Absolute number of reads recovered using Hamming distance versus edit distance across the three protocols that utilize anchor matching. **Right:** Percentage gain from switching to edit distance. The benefit is most pronounced on long-read data (LR-SPLiT-seq, $+15.6\%$), where PacBio HiFi indels near linker sequences cause Hamming-distance matching to miss valid reads. Edit distance is not applicable to 10x Chromium v2, which lacks anchor sequences.

Configuration	**Recovery**	**Time**
Forward-only, Hamming	22.0%	4.5 s
Forward-only, edit distance	25.8%	**2.8 s**
Orientation-aware, Hamming	43.1%	8.4 s
Orientation-aware, edit distance	**49.9%**	5.3 s
MATCHBOX, forward-only	24.0%	3.5 s
MATCHBOX, dual-orientation	39.7%	7.6 s

Table 6.1: **LR-SPLiT-seq recovery progression.** Each row adds one capability on SRR13948564 (1 M reads, 4 threads, 0.2 error rate). Orientation awareness nearly doubles recovery; edit distance adds a further 7 percentage points. SEQPROC with both features recovers 101 K more reads than MATCHBOX's dual-orientation mode.

Investigating why annotation-based orientation detection outperforms a manually written reverse-complement geometry revealed a critical architectural insight. A hand-written geometry reverses the linker search order and introduces a leading unbounded wildcard for the genomic prefix, fundamentally altering the search dynamics. The union of the forward and manual-RC geometries re-

covered 406,446 reads, while the exact annotation approach recovered 430,807 reads. The automated annotation approach discovered an additional 24,361 valid reads (a 6% increase) precisely because it applies the exact same search semantics to the reverse-complemented buffer.

6.5 Benchmarking Performance

6.5.1 Speed and Memory

Table 6.2 details the head-to-head performance comparison across all four benchmark protocols.

Dataset	Tool	Time (s)	Memory (MB)	Valid Rec. (%)
SPLiT-seq PE (87M)	SEQPROC	**95.0**	**176**	**64.9**
	MATCHBOX	1,290.9	23,049	61.4
	SPLITCODE	114.7	1,293	**64.9***
LR-SPLiT-seq (4.2M)	SEQPROC	22.1	**158**	**6.2**
	MATCHBOX	**19.1**	639	5.7
	SPLITCODE	27.9	541	**6.2**†
10x Chromium v2 (234M)	SEQPROC	429.5	**140**	**100.0**
	MATCHBOX	635.0	42,585	**100.0**
	SPLITCODE	**416.7**	1,259	**100.0**
sci-RNA-seq3 (22M)	SEQPROC	**26.8**	**138**	89.7
	MATCHBOX	198.4	3,198	**90.1**
	SPLITCODE	32.4	1,850	89.2

Table 6.2: **Performance comparison of SEQPROC, MATCHBOX [32], and SPLITCODE [25].** Mean of 3 replicates, 32 threads, full datasets. Valid recovery reports the fraction of input reads for which each tool produced structurally valid output. *Although SPLITCODE matches SEQPROC's valid recovery, its total raw recovery contains a large fraction of structurally invalid reads (see discordant analysis). †Forward-only search.

SEQPROC consistently peaks between 138 and 254 MB of Resident Set Size (RSS) across all protocols. This represents a 3 to 9× reduction compared to SPLITCODE and a 2 to 305× reduction compared to MATCHBOX. The memory efficiency is most evident on the 10x Chromium v2 dataset:

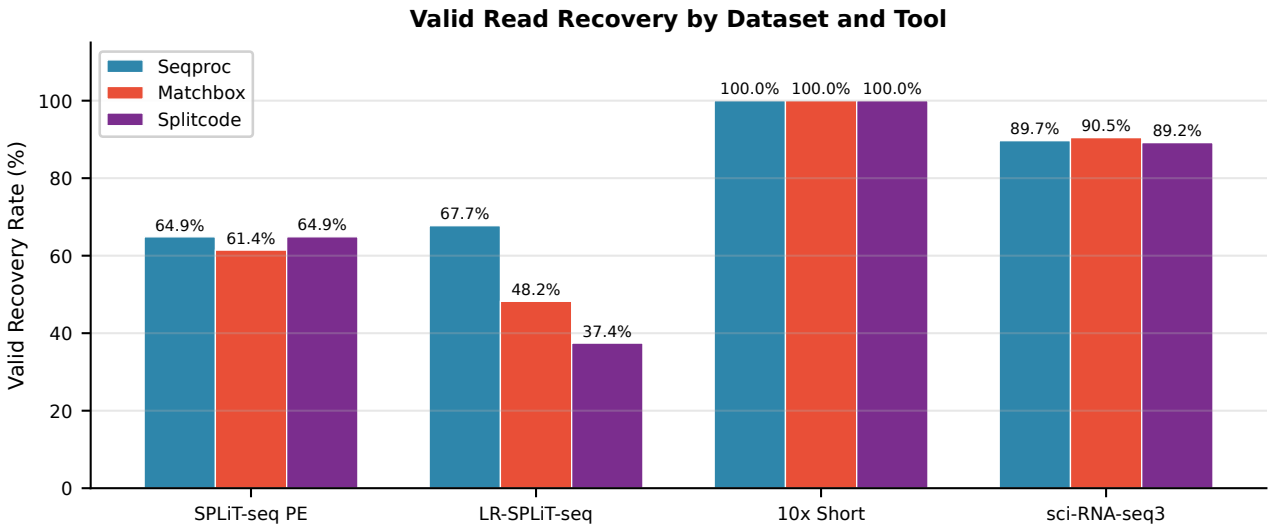

Figure 6.2: **Valid read recovery rates across four single-cell chemistries.** Bars represent the percentage of input reads for which each tool produced structurally valid output. All three tools achieve 100% recovery on 10x Chromium v2. SEQPROC's advantage on LR-SPLiT-seq reflects its dual-orientation edit-distance search capabilities.

MATCHBOX requires 42.6 GB of RAM to process 234 million reads, whereas SEQPROC processes the identical dataset utilizing only 140 MB. This $O(1)$ memory footprint is the direct empirical result of the batched read recycling architecture.

6.5.2 Accuracy and Concordance

Pairwise Jaccard indices between the tools' recovered read sets demonstrate high overall agreement, confirming that performance gains were achieved without sacrificing accuracy (Table 6.3).

Dataset	sp vs mb	sp vs sc	mb vs sc
SPLiT-seq PE	**0.926**	0.916	0.850
LR-SPLiT-seq	**0.709**	0.499	0.413
10x Chromium v2	**1.000**	**1.000**	**1.000**
sci-RNA-seq3	**0.995**	0.991	0.986

Table 6.3: **Pairwise concordance (Jaccard index) across tools on the full datasets.** sp = SEQPROC, mb = MATCHBOX, sc = SPLITCODE. SEQPROC with edit distance acts as a near-perfect superset of MATCHBOX.

6.6 Validity Analysis Methodology

Raw recovery rate alone is an insufficient metric; a tool that blindly outputs every input read would achieve 100% recovery while producing scientifically unusable data. To verify that recovered reads represent genuine biological signal, an independent structural validation framework was developed. This framework operates directly on the raw input reads, completely isolated from the three benchmarked tools.

For each complex geometry, the framework determines which input reads contain a structurally valid barcode configuration. A read is deemed structurally valid if the expected linker and anchor sequences are located and the flanking barcode sequences match known whitelist entries within controlled Hamming tolerances. A tool's *valid recovery* is subsequently calculated as the fraction of its output read IDs that appear in this ground-truth set. The formal per-protocol validity criteria are detailed in Appendix B.

Anchoring validity in the raw input reads ensures that each tool's precision is evaluated against a fixed, objective standard. Pairwise concordance checks revealed that 99.2% of the SPLiT-seq PE reads recovered exclusively by SPLITCODE lacked valid linker structures. This frames SPLITCODE's apparently superior raw recovery rate as predominantly false-positive recovery.

6.7 Benchmark Configuration and Reproducibility

All benchmark results are fully reproducible via an automated pipeline available at:

`https://github.com/COMBINE-lab/seqproc-paper-analysis`

A single entry-point script automatically downloads data, builds binaries from source, installs depen-

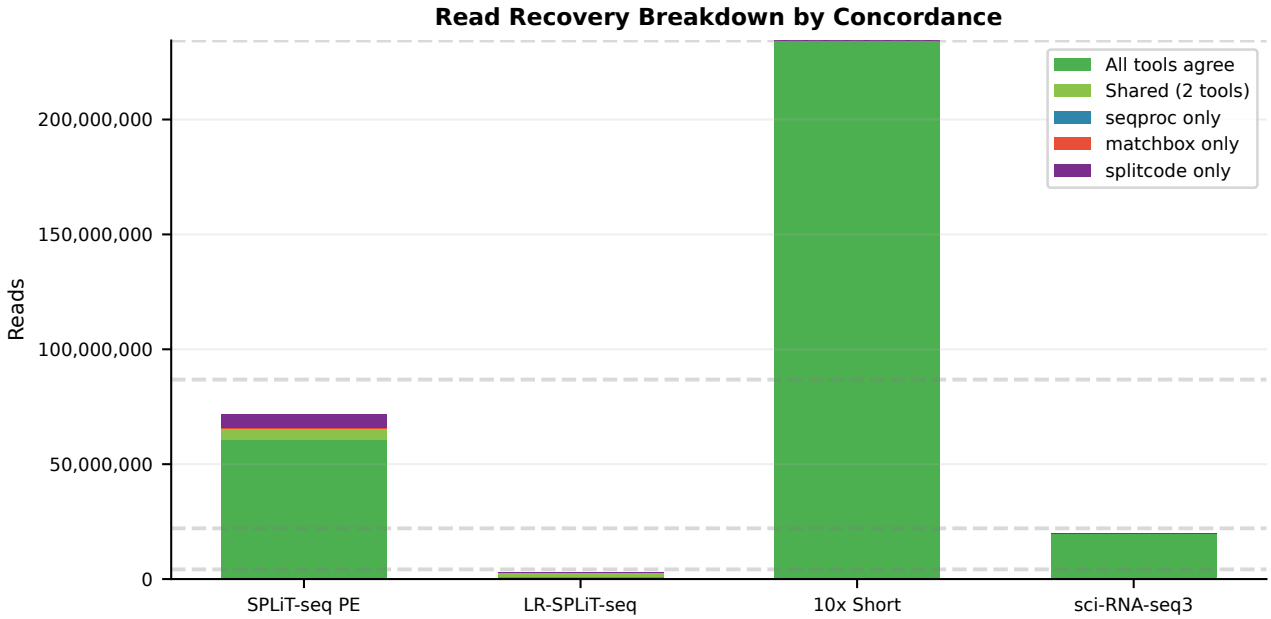

Figure 6.3: **Structural analysis of discordant reads (reads unique to a single tool).** Each bar displays the fraction of tool-unique reads that pass the independent structural validity check. For reads recovered exclusively by SPLITCODE on SPLiT-seq PE, 99.2% lack valid linker structure. By contrast, reads uniquely recovered by SEQPROC overwhelmingly pass structural validity checks.

dencies, and executes the full benchmark suite. Table 6.4 summarizes the public SRA datasets utilized.

Dataset	SRA Accession	Reads	Layout	Read Length
SPLiT-seq PE [22]	SRR6750041	86,820,578	Paired-end	94 bp (R2)
LR-SPLiT-seq [24]	SRR13948564	4,229,250	Single-end	1 kb (PacBio)
10x Chromium v2 [39]	SRR8315379	234,382,218	Paired-end	26 bp (R1)
sci-RNA-seq3 [5]	SRR7827254	22,088,821	Paired-end	66 bp (R1)

Table 6.4: **Benchmark datasets used for evaluation.** All datasets are publicly available from the NCBI Sequence Read Archive.

During configuration, several tool-specific constraints required strict normalization to ensure equivalence. When configuring MATCHBOX for SPLiT-seq PE data, its pattern matcher utilizes a leading wildcard to absorb the 2 bp prefix before the UMI. Specifying `umi:|10|` causes read recovery to drop to 0.3% due to internal heuristics. Utilizing `umi:|8|` allows the wildcard to absorb 4 bp,

successfully restoring normal recovery. Furthermore, executing SPLITCODE with a Hamming distance of 6 on 30 bp linkers triggered an out-of-memory failure with a 25.9 GB peak RSS. A reduced distance of 3 was utilized to maintain system stability. Lastly, the full 30 bp Linker 2 cannot be reliably matched by MATCHBOX on noisy PacBio reads. Truncating this definition to the first 22 bp resolves the matching failure and ensures a fair evaluation.

6.8 Summary of Optimization Contributions

Table 6.5 aggregates the optimization passes, detailing their measured impact on execution time. The dominant architectural theme is memory allocation reduction, driving the bulk of the throughput gains. The secondary theme focuses on algorithmic quality, proving that advanced techniques like edit distance and orientation awareness do not merely accelerate existing operations, but enable qualitatively superior data recovery.

Category	**Optimization**	**Impact**
Allocation	Batched read recycling	-30 to -58%
Allocation	`SmallVec` for mappings	-10%
Allocation	In-place `SetOp` + constant folding	-5%
Synchronization	`parking_lot::Mutex` + lock amort.	-10%
Synchronization	Chunk size tuning (1,024)	optimal
I/O	Thread-local writer cache	-2 to -5%
Algorithm	Adaptive edit distance	$2.3\times$ faster
Algorithm	`TryOrientationOp`	+25.8% recovery
Failed	Output batching	+20 to 30% regr.
Failed	Chunk size 512	up to +110% regr.
Failed	Vectored writes (`IoSlice`)	no effect

Table 6.5: **Summary of all optimization experiments.** Negative impact indicates a reduction in processing time (faster execution). Failed experiments illustrate design boundaries that informed final architectural choices.

6.8.1 Biological and Computational Impact

The performance gap between SEQPROC and legacy alternatives carries direct practical consequences for production bioinformatics. Processing 87 million SPLiT-seq PE read pairs requires 95 seconds using SEQPROC, compared to 1,291 seconds utilizing MATCHBOX. On commodity cloud infrastructure pricing, SEQPROC completes this dataset for under $0.01 of compute cost, whereas MATCHBOX costs roughly $0.14. Scaled to large cohort studies processing hundreds of millions of reads across dozens of samples, this efficiency directly reduces both project costs and wall-clock latency.

The memory profile constitutes a more fundamental advantage. Because SEQPROC maintains an $O(1)$ peak RSS, dataset size is constrained practically by physical storage rather than available RAM. The 10x Chromium v2 dataset contains 234 million reads; processing it with MATCHBOX demands 42.6 GB of RAM, effectively excluding researchers utilizing standard desktop hardware or base-tier cloud instances. SEQPROC processes the exact same dataset utilizing 140 MB of RAM, democratizing access to large-scale data analysis.

Finally, the declarative nature of the EFGDL specification addresses significant configuration complexity. A standard geometry file requires 8 to 15 lines and structurally maps directly to the biological protocol documentation. This limits the potential for the logical configuration errors that frequently plague lengthy imperative scripts, ensuring that high-performance preprocessing remains accessible and auditable.

Chapter 7: Conclusion and Future Directions

7.1 Summary of Contributions

This thesis addresses a concrete and growing problem in high-throughput genomics: the geometric complexity of sequencing protocols has outpaced the architectural capabilities of legacy preprocessing tools. Prior to SEQPROC, existing preprocessors forced users to compromise between execution speed, algorithmic flexibility, and configuration simplicity.

The contributions of this work resolve this tension by separating the biological intent from the execution mechanics. At the language layer, we introduced EFGDL, a declarative notation that allows users to specify the exact geometry of a read without prescribing the underlying search strategy. At the execution layer, we engineered ANTISEQUENCE, a highly optimized, batched directed acyclic graph (DAG) execution engine.

By dynamically mapping declarative annotations to highly optimized SIMD algorithms, SEQPROC demonstrates that high-level abstractions do not require a performance penalty. Empirical evaluations across four diverse single-cell chemistries confirm that SEQPROC is consistently the fastest tool on three of the four protocols, utilizing up to 305$\times$ less memory than imperative alternatives, all while achieving superior valid read recovery on complex, long-read datasets.

7.2 Limitations

The current architecture has several known limitations. First, EFGDL does not yet support quality-score-based filtering. Currently, a read with high-quality barcode bases and low-quality anchor bases is treated identically to one with the reverse profile. For protocols where base quality is a reliable proxy for sequencing error, the inability to reject reads with degraded anchors prior to approximate matching limits potential precision gains.

Second, the transformation clause cannot express conditional logic. A specification cannot dictate that an interval should be output as-is if it matches a whitelist exactly, but replaced with the closest match if it requires error correction. While the `map_with_mismatch` transformation provides a functional workaround, it cannot branch dynamically based on the match quality score.

Third, the ANTISEQUENCE graph is currently executed exactly as compiled. No optimization passes are applied to the DAG structure prior to execution. Techniques such as node fusion, dead-interval elimination, and adaptive batch sizing are highly feasible within the current architecture but remain unimplemented.

7.3 Future Directions

7.3.1 Compiler Optimizations for the Execution Graph

Classical compiler optimizations apply directly to the ANTISEQUENCE DAG. Node fusion could merge sequential operations that always execute together, eliminating per-node dispatch overhead and improving instruction-cache locality. Dead-node elimination could identify intervals that are matched but never referenced in the final transformation clause, allowing the engine to skip extraction entirely.

Furthermore, adaptive batching could dynamically tune the chunk size based on real-time per-read processing costs, maintaining optimal thread utilization across geometries of varying computational intensity.

7.3.2 Expanding the Annotation Ecosystem

The annotation system was designed specifically with extensibility in mind. Several new annotations could address currently unsupported use cases without breaking backward compatibility. A `#[min_qual(k)]` annotation could require the mean base quality of a matched interval to exceed a threshold k, enabling quality-aware filtering. A `#[prob_match(p)]` annotation could utilize quality scores as Bayesian priors to evaluate the correctness of a match, providing a soft-decision analogue to rigid edit-distance thresholds. Finally, a `#[capture(name, dist)]` annotation could expose the numerical match distance as a named attribute, making it available for downstream conditional logic.

In the current implementation, however, every new annotation must be added in the compiler itself. The lexer tokenizes the `#[...]` form as a `HashBracket` token, the parser lifts it into an `Annotation` AST node, and the compiler routes each annotation name to a concrete handler through a hard-coded string match in the `annotations_to_compiled_functions` dispatch table. Adding a single distance- or search-style annotation therefore touches roughly four source files across SEQPROC and ANTISEQUENCE, including the dispatch table, the supporting utility that validates its arguments, the `CompiledFunction` variant it emits, and the corresponding operation constructor in the execution engine. This places the extensibility of the language entirely in the hands of the tool's maintainers. A natural next step is to replace the fixed dispatch table with a small extension

mechanism driven by a directory of *minimal extension files*. Each file would declare a single annotation through a manifest consisting of the annotation name, the expected argument types, a short validator, and a handler that returns either a `CompiledFunction` or, for parameter-style annotations, a pass-through configuration value. At compile time, SEQPROC would scan this directory, register each manifest in a runtime registry keyed by annotation name, and look annotations up by that name instead of pattern-matching on hard-coded strings. Users could then drop a file such as `phred_min.ext` into their project alongside their geometry to add a bespoke annotation without modifying the core tool, and community-contributed annotations could be distributed as standalone text artifacts curated in the same repository as the EFGDL Exchange.

7.3.3 Pluggable Algorithm Selection

The dynamic algorithm selection described in Chapter 5 currently encodes its routing decisions inline. Inside `MatchAnyOp`, the edit-distance and semi-global edit-search paths each test pattern length against a single 64 bp threshold, sending short patterns to Myers' bit-vector algorithm and longer patterns to a standard dynamic-programming fallback. Full-alignment match types (global, local, prefix, suffix) dispatch through a generic `Aligner` trait to block-aligner-backed implementations, but the Hamming and edit paths bypass this trait and call concrete functions directly. The only condition examined at dispatch time is pattern length. Consequently, introducing a new algorithm today requires editing the operation itself rather than composing a new backend against a stable interface.

Making this layer genuinely pluggable has two components. First, the `Aligner` trait should be elevated into a first-class registration API, uniform across Hamming, edit, and full-alignment match types, so that any algorithm, whether a BWT-based exact matcher, a SIMD Hamming-only kernel, a

banded Smith-Waterman implementation, or a learned index, can be added as a self-contained module that exports only its name, its capabilities (supported match type, maximum pattern length, minimum and maximum edit budget), and a single `align` method. Second, the hard-coded length cutoff should be replaced by a declarative `SelectionPolicy`. Each registered algorithm would advertise a predicate over runtime features, including pattern length, edit budget, number of patterns in the set, alphabet size, and the width of the available SIMD register, and the engine would pick the first algorithm whose predicate matches, falling back to a default registered at engine construction time. The current 64 bp threshold would then become the predicate `length <= 64 && match_type == Edit` attached to the Myers backend rather than an unreachable constant in the dispatch code. Exposing this policy to users would allow workloads that know their inputs are short and exact to install a specialized Hamming-only kernel, and it would let future work integrate accelerator backends such as a GPU matcher that is only profitable for very large pattern sets, all without further modification to `MatchAnyOp`. Combined with the annotation-extension mechanism of the previous section, this turns the most performance-critical piece of the system into a user-extensible surface while preserving the guarantees that the EFGDL compiler relies on.

7.3.4 The EFGDL Exchange

A practical barrier to the adoption of any bioinformatics tool is the effort required to configure it for a novel chemistry. A community-driven repository of verified EFGDL specifications, would lower this barrier substantially. Users could execute a command to pull a geometry file that has been validated against synthetic data and reviewed by the community, eliminating the need to write specifications from scratch. The combination of a concise declarative language and machine-checkable correctness

criteria makes EFGDL uniquely suited for this type of automated, community curation.

7.3.5 Multi-Modal, Multiplexed, and Spatial Protocols

Single-cell technologies continue to rapidly evolve. Spatial transcriptomics platforms attach spatial barcodes that encode physical tissue coordinates, producing reads with three or more distinct barcode levels alongside complex structural elements. Multi-modal assays, such as CITE-seq [40] and REAP-seq [41], append antibody-derived tag libraries to the same read as the transcriptomic payload, requiring tools to separate two distinct payloads based on internal structural features. Multiplexed protocols such as 10x Genomics Chromium Flex extend this axis further by pooling up to 384 biological samples into a single GEM well and encoding the sample of origin as a separate probe barcode on R2, distinct from the cell-identifying GEM barcode on R1. To make this distinction first-class in the language, a dedicated sample-barcode specifier `s` was added to EFGDL (see Appendix A). Mechanically, `s` shares the full set of barcode transformations with `b` and uses the same runtime path, so the addition is free of performance cost; semantically, it separates cell identity from sample identity in the output so that downstream demultiplexers can route each barcode to its correct use without additional metadata bookkeeping. Beyond this concrete extension, both multi-modal and spatial use cases fall within the expressive potential of EFGDL. Multi-level barcodes map directly to multiple interval definitions with separate whitelists, and conditional payload extraction maps to separate read-structure blocks. Validating the current annotation and specifier system against the full diversity of these emerging protocols remains an area for future work.

7.3.6 Integration with Downstream Quantification

SEQPROC currently produces standard FASTQ output designed for compatibility with downstream quantification pipelines such as STARSOLO [42] and Alevin [21]. However, a tighter integration where SEQPROC passes labeled interval metadata directly to the quantification tool in memory would eliminate the I/O bottleneck entirely. The ANTISEQUENCE data model is already structured for this; labeled intervals are named and offset-based, allowing a downstream consumer to receive a batch of processed objects and extract barcodes by name without re-parsing sequence bytes. Whether this integration is practical depends heavily on the memory models of the downstream tools, but it represents a highly promising architectural direction.

Chapter A: EFGDL Language Reference

This appendix provides a formal reference for the Extended Fragment Geometry Description Language (EFGDL). While the examples in Chapter 4 develop the intuition for common specifications, this reference defines the syntax, interval behaviors, and grammar constraints required for handling edge cases and language extensibility.

A.1 Interval Types and Specifiers

Each interval in an EFGDL specification possesses a *type*, governing length behavior, and a *specifier*, declaring the biological role of the sequence.

Fixed-length intervals. The fixed-length type uses an integer bound, such as `b[16]`, to match a sequence of exactly that length. The length is determined at parse time and remains invariant across all reads. This is the standard type for elements occupying known, invariant coordinates.

Variable-length intervals. The variable-length type uses a range, such as `b[9-10]`, to match either specified length. Resolution occurs during matching: SEQPROC identifies the fixed-sequence fragment immediately following the interval to infer the correct length. Variable-length intervals must be anchored on the right by a fragment or the terminal end of the read to remain unambiguous.

Unbounded intervals. The unbounded type, written `r:`, matches all remaining bases to the end of the read. The colon denotes that no upper length limit is imposed. Only one unbounded interval may

appear per read structure, and it must occupy the terminal position. This is the standard type for cDNA payloads.

Fixed-sequence (fragment) intervals. The fixed-sequence type, written as `f[...]`, matches a specific nucleotide sequence. These intervals serve as structural anchors to establish positional context. Fragments are the only interval type permitted to carry `#[hamming]`, `#[edit]`, and `#[search(relative)]` annotations.

Specifiers. Six single-character specifiers are available, each constraining the biological interpretation of the matched sequence and its eligibility for downstream transformations.

`b` designates a cell barcode. Barcode intervals are retained in the output by default and are eligible for `map`, `filter`, and `norm` transformations. `s` designates a sample barcode: mechanically identical to `b`, with the same transformation eligibility and the same runtime code path, but carrying a distinct label so that downstream tools can separate cell identity from sample identity without additional bookkeeping. The motivating use case is multiplexed single-cell protocols such as 10x Genomics Chromium Flex, in which up to 384 biological samples are pooled into a single GEM well and the sample of origin is encoded by a probe barcode in R2 that is distinct from the cell-identifying GEM barcode in R1. `u` designates a unique molecular identifier (UMI), retained by default; UMI intervals are not eligible for barcode-specific transformations. `r` designates the cDNA read payload and is retained. `x` designates a discard interval: the sequence is matched and consumed at its expected position but never written to any output file. The `x` specifier is the idiomatic way to absorb ligation artifacts, primer scars, and other structural elements that must be accounted for positionally but carry no biological signal. `f` is reserved for fixed-sequence fragments; it signals to the parser that this interval is a

structural anchor rather than a biological payload and enables the annotation types that fixed-sequence anchors require.

The specifier system enforces type safety at parse time. Applying `norm`, a barcode normalization transformation, to a UMI interval is a type error. Passing a discard-specifier interval as the argument of `map` is a type error. Because `s` shares the full set of barcode transformations with `b`, the same whitelist, Hamming, edit-distance, padding, and truncation operations apply uniformly to both, and no per-specifier rules distinguish the two at the compile stage. These errors are caught before any reads are processed, following the same philosophy as static typing in compiled languages: invalid programs are rejected at the earliest possible stage rather than producing silently incorrect output at runtime.

A.2 Annotations

Annotations decouple matching behavior from interval identity. They use the syntax `#[name(args)]` and are placed immediately preceding a definition. Annotations are orthogonal and may be stacked on a single definition without interaction.

Matching Annotations. `#[hamming(n)]` accepts a fragment match with up to n substitutions. `#[edit(n)]` is the permissive variant, accepting a match within Levenshtein edit distance n, which includes substitutions, insertions, and deletions.

Positional Annotations. `#[search(relative)]` scans the read for the best-matching position of a fragment rather than requiring a fixed offset. Preceding variable-length intervals are resolved relative to the located anchor position.

Read-level Annotations. `#[match_ori(either)]` is applied to a read-structure block. It instructs SEQPROC to attempt a match in the forward orientation and, if unsuccessful, retry on the reverse complement of the read.

Annotation	Effect
`#[hamming(n)]`	Accept fragment match with up to n substitutions (Hamming distance).
`#[edit(n)]`	Accept fragment match with up to n edits (Levenshtein distance).
`#[search(relative)]`	Scan read for best-matching position; resolve preceding variable-length intervals relative to this anchor.
`#[match_ori(either)]`	Attempt match in forward orientation; retry on reverse complement upon failure.

Table A.1: **EFGDL annotations.** Each annotation modifies a specific aspect of matching behavior. They may be combined freely.

A.3 Transformations

Transformations are applied after intervals are successfully matched. They fall into two categories: output-structure transformations and sequence-modification transformations.

Output-structure transformations. Expressed in the `->` clause, these control which labeled intervals appear in the output. Any interval named in the clause is written to the output in the specified order; unnamed intervals are discarded. This is the primary mechanism for reordering barcodes and removing linker sequences.

Sequence-modification transformations. These reshape the matched subsequence before output. Supported operations include `rev` (reverse), `revcomp` (reverse-complement), and various `trunc`

(truncate) or `pad` functions. `map` and `map_with_mismatch` replace matched sequences with canonical entries from an external file.

Filtering and Normalization. `filter` and `filter_within_dist` discard the read if the matched interval is absent from a reference file. This occurs during the matching phase to conserve I/O bandwidth. `norm(I)` converts variable-length barcodes to a fixed width by appending a deterministic suffix encoding the original length offset.

A.4 Grammar Constraints and Determinism

EFGDL enforces the *right-anchoring rule* to ensure every specification is semantically unambiguous. Every variable-length and unbounded interval must be anchored on the right by either a fixed-sequence fragment or the physical end of the read.

Without this rule, a specification such as `1{b[9-10] r:}` is ambiguous, as multiple assignments are consistent with the grammar. Adding a fragment, such as `1{b[9-10] f[CAGAGC] r:}`, allows SEQPROC to scan for the anchor and resolve the length of the preceding interval definitively.

Two additional constraints ensure the integrity of the grammar. The first restricts the use of unbounded intervals (`r:`) to a maximum of one per read structure, further requiring that it occupy the terminal position. The second mandates that the transformation clause reference only those labels that are both explicitly named in the definitions block and matched within the read structure. Together, these three constraints ensure that every EFGDL specification passing the parser is both syntactically complete and semantically unambiguous, effectively eliminating classes of errors that would otherwise only surface during execution.

Chapter B: Structural Validity Analysis

Design Rationale

A benchmark utilizing a tool's own output to define a correct recovery contains a circular dependency. A tool with permissive matching criteria will produce more output, inflating both its own numerator and the denominator used to evaluate stricter tools. To eliminate this bias, we developed an independent structural validity framework that evaluates reads prior to any tool execution. Given only the raw input reads and the protocol specification, this framework identifies which reads contain a genuine, decodeable barcode structure.

The framework applies a protocol-specific algorithm to every input read for the three protocols with non-trivial geometry: SPLiT-seq PE, LR-SPLiT-seq, and sci-RNA-seq3. These algorithms search for expected structural elements, such as linker sequences and barcode fields, using the same tolerances as SEQPROC (as specified by each geometry file) but implemented independently in Python. A read passing this algorithmic check is added to the *valid input set*. Each tool's *valid recovery* is subsequently measured as the fraction of its output read identifiers that appear in this set.

For the LR-SPLiT-seq dataset, the validity analyzer utilizes exact string matching for the 30 bp linker. This provides a conservative lower bound on the ground-truth set. Approximately 74% of truly barcoded HiFi reads are expected to carry an error-free 30 bp linker based on a 1% per-base error rate. This approach ensures that no false positives are admitted into the ground-truth set. The formal criteria for each protocol are summarized in Table B.1.

B.1 SPLiT-seq Paired-End

The SPLiT-seq PE R2 read encodes the combinatorial barcode structure at approximately fixed positions: `[NN:2][UMI:10][BC3:8][Linker1:30][BC2:8][Linker2:30][BC1:6]`. Three separate whitelists define the valid barcode space for each respective round.

To determine structural validity, the analyzer first scans for Linker 1 within the initial 39 bases of the read using a sliding window. If the best match exceeds a Hamming distance of 3, the read is immediately marked invalid. Next, it scans for Linker 2 within a 40-base window situated 30 bases downstream of the Linker 1 position, again requiring a Hamming distance of 3 or less. Once both linkers are anchored, the algorithm extracts the three barcodes (BC3, BC2, and BC1) from their expected relative offsets. If any extracted barcode is shorter than its expected length, or if it fails to match its corresponding whitelist within a Hamming distance of 1, the read is classified as invalid. Reads passing all these criteria are added to the valid input set.

B.2 LR-SPLiT-seq

PacBio HiFi reads are approximately 1 kb in length and randomly oriented. The barcode cassette can appear at any position within the read.

For the LR-SPLiT-seq protocol, the validity algorithm performs an exact string search for Linker 1 across the entire sequence. If the linker is not found in the forward orientation, the read is reverse-complemented and searched a second time. If the linker is absent in both orientations, or if it appears too close to the sequence boundaries to accommodate the flanking barcodes, the read is rejected. Upon successfully anchoring the linker, the algorithm extracts BC3, BC2, and BC1 relative to the exact linker

position. The read is deemed valid only if all three extracted barcodes match their respective whitelists within a Hamming distance of 1.

B.3 10x Chromium v2

The 10x Chromium v2 R1 read features a fixed 26 bp layout consisting of a 16 bp barcode and a 10 bp UMI.

Because the protocol produces fixed-length R1 reads by construction, the validity check is exceptionally straightforward. The algorithm simply verifies that the total read length is exactly 26 bases. Reads meeting this strict length requirement are marked valid, while all others are rejected. This criterion is intentionally permissive; all three benchmarked tools achieve 100% recovery on this dataset, consistent with the trivial validity check.

B.4 sci-RNA-seq3

The sci-RNA-seq3 R1 read contains a variable-length barcode followed by a fixed 6-base anchor sequence (`CAGAGC`).

To validate sci-RNA-seq3 reads, the algorithm explicitly searches for this anchor sequence. It evaluates starting positions 8, 9, 10, and 11 to seamlessly accommodate the variable-length initial barcode. If the anchor is found at any of these specific positions with a Hamming distance of 1 or less, the read is classified as valid. If the anchor is not found within this narrow positional window, the read is marked invalid and discarded from the ground-truth set.

B.5 Summary of Validity Criteria

Table B.1 summarizes the structural validity criteria. The analysis on the SPLiT-seq PE dataset highlights the necessity of this metric: while SPLITCODE reports high raw recovery, 21% of its output reads lack recognizable linker or barcode structures.

Dataset	**Read**	**Validity Criteria**
SPLiT-seq PE	R2	Both linkers found (Hamming $\leq$ 3); three barcodes in whitelist (Hamming ≤ 1)
LR-SPLiT-seq	R1	Linker 1 exact match in either orientation; position constraints met; barcodes in whitelist (Hamming ≤ 1)
10x Chromium v2	R1	Read length exactly 26 bp
sci-RNA-seq3	R1	Anchor `CAGAGC` found at position 8 to 11 (Hamming ≤ 1)

Table B.1: **Structural validity criteria per dataset.** Valid recovery is defined as the fraction of output read identifiers present in the input valid set.

Bibliography

[1] Kris A. Wetterstrand. DNA sequencing costs: Data from the NHGRI genome sequencing program (GSP). `https://www.genome.gov/sequencingcostsdata`. Accessed: 2026-03-01.

[2] Eric S. Lander. Initial impact of the sequencing of the human genome. *Nature*, 470(7333): 187–197, 2011. doi: 10.1038/nature09792.

[3] Murim Choi, Ute I. Scholl, Weizhen Ji, Tiewen Liu, Irina R. Tikhonova, Paul Zumbo, Ahmet Nayir, Aysin Bakkaloglu, Seza Ozen, Sami Sanjad, Carol Nelson-Williams, Anita Farhi, Shrikant Mane, and Richard P. Lifton. Genetic diagnosis by whole exome capture and massively parallel DNA sequencing. *Proceedings of the National Academy of Sciences*, 106(45):19096–19101, 2009. doi: 10.1073/pnas.0910672106.

[4] Marco Gerlinger, Andrew J. Rowan, Stuart Horswell, Marco Math, James Larkin, David Endesfelder, Eva Gronroos, Pierre Martinez, Nicholas Matthews, Aengus Stewart, Patrick Tarpey, Ignacio Varela, Benjamin Phillimore, Sharmin Begum, Neil Q. McDonald, Andrew Butler, David Jones, Keiran Raine, Calli Latimer, Claudio R. Santos, Mahrokh Nohadani, Aron C. Eklund, Bradley Spencer-Dene, Graham Clark, Lisa Pickering, Gordon Stamp, Martin Gore, Zoltan Szallasi, Julian Downward, P. Andrew Futreal, and Charles Swanton. Intratumor heterogeneity and branched evolution revealed by multiregion sequencing. *New England Journal of Medicine*, 366 (10):883–892, 2012. doi: 10.1056/NEJMoa1113205.

[5] Junyue Cao, Malte Spielmann, Xiaojie Qiu, Xingfan Huang, Daniel M Ibrahim, Andrew J Hill, Fan Zhang, Stefan Mundlos, Lena Christiansen, Frank J Steemers, et al. The single-cell transcriptional landscape of mammalian organogenesis. *Nature*, 566(7745):496–502, 2019.

[6] Nathan D. Grubaugh, Jason T. Ladner, Moritz U. G. Kraemer, Gytis Dudas, Amanda L. Tan, Karthik Gangavarapu, Michael R. Wiley, Stephen White, Julien Thézé, Diogo M. Magnini, Luiz Cariou, Janko Landeghem, Morena T. Oliveira, Paulo S. Lemos, Zhaleh Rikhtegaran-Tehrani, Camille A. Coomber, et al. Genomic epidemiology reveals multiple introductions of Zika virus into the United States. *Nature*, 546(7658):401–405, 2017. doi: 10.1038/nature22400.

[7] Grace X.Y. Zheng et al. Massively parallel digital transcriptional profiling of single cells. *Nature Communications*, 8:14049, 2017.

[8] Frederick Sanger, Steven Nicklen, and Alan R Coulson. DNA sequencing with chain-terminating inhibitors. *Proceedings of the national academy of sciences*, 74(12):5463–5467, 1977.

[9] International Human Genome Sequencing Consortium. Initial sequencing and analysis of the human genome. *Nature*, 409(6822):860–921, 2001.

[10] David R Bentley, Srinivas Balasubramanian, Harold P Swerdlow, et al. Accurate whole human genome sequencing using reversible terminator chemistry. *Nature*, 456(7218):53–59, 2008.

[11] Miten Jain et al. Nanopore sequencing and assembly of a human genome with ultra-long reads. *Nature Biotechnology*, 36(4):338–345, 2018.

[12] John Eid, Adrian Fehr, Jeremy Gray, et al. Real-time DNA sequencing from single polymerase molecules. *Science*, 323(5910):133–138, 2009.

[13] Mantas Sereika, Rasmus Hansen Kirkegaard, Søren Michael Karst, Thomas Yssing Michaelsen, Emil Aarre Sørensen, Rasmus Dam Wollenberg, and Mads Albertsen. Oxford nanopore r10.4 long-read sequencing enables the generation of near-finished bacterial genomes from pure cultures and metagenomes without short-read or reference polishing. *Nature Methods*, 19(7):823–826, 2022. doi: 10.1038/s41592-022-01539-7.

[14] Tianyuan Zhang, Hanzhou Li, Silin Ma, Jian Cao, Hao Liao, Qiaoyun Huang, and Wenli Chen. The newest oxford nanopore r10.4.1 full-length 16s rrna sequencing enables the accurate resolution of species-level microbial community profiling. *Applied and Environmental Microbiology*, 89(10):e00605–23, 2023. doi: 10.1128/aem.00605-23.

[15] Evan Z. Macosko, Anindita Basu, Rahul Satija, James Nemesh, Karthik Shekhar, Melissa Goldman, Itay Tirosh, Allison R. Bialas, Nolan Kamitaki, Emily M. Martersteck, John J. Trombetta, David A. Weitz, Joshua R. Sanes, Alex K. Shalek, Aviv Regev, and Steven A. McCarroll. Highly parallel genome-wide expression profiling of individual cells using nanoliter droplets. *Cell*, 161 (5):1202–1214, 2015. doi: 10.1016/j.cell.2015.05.002.

[16] Amit Zeisel, Ana B. Muñoz-Manchado, Simone Codeluppi, Peter Lönnerberg, Gioele La Manno, Anna Juréus, Sueli Marques, Hermany Munguba, Liqun He, Christer Betsholtz, Charlotte Rolny, Gonçalo Castelo-Branco, Jens Hjerling-Leffler, and Sten Linnarsson. Brain structure. Cell types in the mouse cortex and hippocampus revealed by single-cell RNA-seq. *Science*, 347(6226): 1138–1142, 2015. doi: 10.1126/science.aaa1934.

[17] Simone Picelli, Åsa K. Björklund, Omid R. Faridani, Sven Sagasser, Gösta Winberg, and Rickard Sandberg. Smart-seq2 for sensitive full-length transcriptome profiling in single cells. *Nature Methods*, 10(11):1096–1098, 2013. doi: 10.1038/nmeth.2639.

[18] Markus Hafner, Markus Landthaler, Lukas Burger, et al. Transcriptome-wide identification of RNA-binding protein and microRNA target sites by PAR-CLIP. *Cell*, 141(1):129–141, 2010.

[19] Rob Patro, Geet Duggal, Michael I. Love, Rafael A. Irizarry, and Carl Kingsford. Salmon provides fast and bias-aware quantification of transcript expression. *Nature Methods*, 14(4):417–419, 2017. doi: 10.1038/nmeth.4197.

[20] Benjamin Kaminow, Dinar Yunusov, and Alexander Dobin. STARsolo: accurate, fast and versatile mapping/quantification of single-cell and single-nucleus RNA-seq data. *bioRxiv*, 2021. doi: 10.1101/2021.05.05.442755.

[21] Dongze He, Mohsen Zakeri, Hirak Sarkar, Charlotte Soneson, Avi Srivastava, and Rob Patro. Alevin-fry unlocks rapid, accurate and memory-frugal quantification of single-cell RNA-seq data. *Nature Methods*, 19(3):316–322, 2022. doi: 10.1038/s41592-022-01408-3.

[22] Alexander B Rosenberg, Charles M Roco, Richard A Muscat, Anna Kuchina, Paul Sample, Zizhen Yao, Lucas T Graybuck, David J Peeler, Sumit Mukherjee, Wei Chen, et al. Single-cell profiling of the developing mouse brain and spinal cord with split-pool barcoding. *Science*, 360(6385):176–182, 2018.

[23] Elisabeth Rebboah, Fairlie Reese, Katherine Williams, Gabriela Balderrama-Gutierrez, Cassandra McGill, Diane Trout, Isaryhia Rodriguez, Heidi Liang, Barbara J Wold, and Ali Mortazavi. Mapping and modeling the genomic basis of differential rna isoform expression at single-cell resolution with lr-split-seq. *Genome biology*, 22(1):286, 2021.

[24] Anna Kuchina, Leandra M Brettner, Lauren Paleologu, Charles M Roco, Alexander B Rosenberg, Attilio Carignano, Ryan Kibler, Masahiro Hirano, R William DePaolo, and Georg Seelig. Microbial single-cell rna sequencing by split-pool barcoding. *Science*, 371(6531):eaba5257, 2021.

[25] Delaney K. Sullivan and Lior Pachter. splitcode: robust and efficient extraction of combinatorial barcodes from sequencing data. *Bioinformatics*, 40(6):btae331, 2024. doi: 10.1093/bioinformatics/btae331.

[26] Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, 1974. doi: 10.1145/321796.321811.

[27] Esko Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64(1–3): 100–118, 1985. doi: 10.1016/S0019-9958(85)80046-2.

[28] Gene Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM*, 46(3):395–415, 1999.

[29] Nils Homer and Seth Stadick. read-structure: Library for parsing and working with read structure descriptions, 2024. URL `https://github.com/fulcrumgenomics/read-structure`.

[30] Tom Smith, Andreas Heger, and Ian Sudbery. UMI-tools: modeling sequencing errors in Unique Molecular Identifiers to improve quantification accuracy. *Genome Research*, 27(3):491–499, 2017. doi: 10.1101/gr.209601.116.

[31] Yusuke Kijima, Daniel Evans-Yamamoto, Hiromi Toyoshima, and Nozomu Yachie. A universal sequencing read interpreter. *Science Advances*, 9(1), 2023. doi: 10.1126/sciadv.add2793.

[32] Jakob Schuster, Kathleen Zeglinski, Lucinda Xiao, Olivia Voulgaris, Sarahi Mendoza Rivera, Stephin J. Vervoort, Matthew E. Ritchie, Quentin Gouil, and Michael B. Clark. Powerful read processing with matchbox. *bioRxiv*, 2025. doi: 10.1101/2025.11.09.685711.

[33] Joshua Barretto. Chumsky crate. `https://docs.rs/chumsky/latest/chumsky/`. Accessed: 2024-08-27.

[34] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, page 111–122, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 158113729X. doi: 10.1145/964001.964011. URL `https://doi.org/10.1145/964001.964011`.

[35] Daniel Liu and Martin Steinegger. Block Aligner: an adaptive SIMD-accelerated aligner for sequences and position-specific scoring matrices. *Bioinformatics*, 39(8):btad487, 2023.

[36] Niko Matsakis, Josh Stone, and Rayon Contributors. Rayon: A data parallelism library for rust. `https://docs.rs/rayon/`, 2026. Accessed: 2026-03-31.

[37] Brook Heisler and Jorge Aparicio. Criterion.rs: Statistics-driven Microbenchmarking in Rust. `https://github.com/bheisler/criterion.rs`, 2024. Accessed: 2026-03-01.

[38] David Yuan, Alisha Ahamed, Josephine Burgin, Carla Cummins, Rajkumar Devraj, Khadim Gueye, Dipayan Gupta, Vikas Gupta, Muhammad Haseeb, Maira Ihsan, et al. The european nucleotide archive in 2023. *Nucleic Acids Research*, 52(D1):D92–D97, 2024. doi: 10.1093/nar/gkad1067.

[39] Hyun Min Kang, Meena Subramaniam, Sasha Targ, Michelle Nguyen, Lenka Maliskova, Elizabeth McCarthy, Eunice Wan, Simon Wong, Lauren Byrnes, Cristina M Lanata, et al. Multiplexing droplet-based single cell rna-sequencing using natural genetic variation. *Nature Biotechnology*, 36(1):89–94, 2018.

[40] Marlon Stoeckius, Christoph Hafemeister, William Stephenson, Brian Houck-Loomis, Pratip K Chattopadhyay, Harold Swerdlow, Rahul Satija, and Peter Smibert. Simultaneous epitope and transcriptome measurement in single cells. *Nature methods*, 14(9):865–868, 2017.

[41] Vanessa M Peterson, Kelvin Xi Zhang, Namit Kumar, Jerelyn Wong, Lihua Li, Douglas C Wilson, Rene Moore, Thomas K McClanahan, Svetlana Sadekova, and Joel A Klappenbach. Multiplexed quantification of proteins and transcripts in single cells. *Nature Biotechnology*, 35(10):936–939, 2017.

[42] Benjamin Kaminow, Dinar Yunusov, and Alexander Dobin. Starsolo: accurate, fast and versatile mapping/quantification of single-cell and single-nucleus rna-seq data. *bioRxiv*, pages 2021–05, 2021.