symmetry

MDPI

Article

Computational Performance of Programming Languages in Mathematical Biology: A Ten-Language Evaluation Across Six Modeling Regimes

Yi Zheng [1] and Qiuming Luo [2,*]

[1] School of Nano-Tech and Nano-Bionics, University of Science and Technology of China, Suzhou 215123, China; yzheng2026@sinano.ac.cn
[2] School of Computer Science and Software Engineering, Shenzhen University, Shenzhen 518060, China
* Correspondence: lqm@szu.edu.cn

Abstract

Mathematical biology spans multiple computational regimes—from ordinary differential equation models of gene regulatory networks to stochastic simulation of chemical kinetics and reaction-diffusion models of spatial pattern formation—each imposing distinct computational demands on the software infrastructure that executes them. The choice of programming language for implementation of these mathematical models carries quantitative performance consequences that have not been systematically measured across the range of models employed in contemporary mathematical biology. This study provides a computational performance evaluation across ten languages (Python, Julia, Rust, C, C++, C#, F#, Go, Java, and R) and six mathematical biology modeling regimes: deterministic ODE integration, stochastic chemical kinetics, parameter-space exploration, reaction-diffusion spatial modeling, agent-based discrete simulation, and Bayesian parameter inference. Execution time, peak memory footprint, cyclomatic complexity, type-conversion density, and the semantic alignment between data structures and biological state representation were recorded under a uniform experimental protocol. Under a unified hand-coded Dormand–Prince 5(4) integrator, native implementations outperform managed-runtime counterparts by nearly two orders of magnitude for ODE integration, a differential that shrinks sharply once library-delegated solvers are removed from the comparison; the gap contracts below 55× when stochastic kinetics shift the bottleneck from arithmetic throughput to branch resolution. In memory-bandwidth-limited reaction-diffusion modeling, native and just-in-time implementations converge to within a few percent. Agent-based models expose a distinct regime-dependent overhead: immutable-by-default collection semantics impose disproportionate cost during mutation-intensive computation. Peak memory varies by roughly two orders of magnitude across languages, directly affecting deployment density for large-scale simulation. Code-structural measurements confirm that algorithmic form enforces a floor on cyclomatic complexity, irrespective of language, while type strictness and mutation semantics generate substantial differences in per-line cognitive load. These results provide a quantitative foundation for computational tool selection in mathematical biology, challenging universal language recommendations and supporting choices grounded in the algorithmic character of each modeling regime.

Keywords: mathematical biology; computational biology; programming languages; differential equations; stochastic processes; systems biology

Academic Editor: George Karabatsos

Received: 20 July 2026
Revised: 24 August 2026
Accepted: 27 August 2026
Published: 18 September 2026

1. Introduction

Mathematical biology has produced a remarkably diverse set of modeling frameworks for understanding biological systems. Ordinary differential equation (ODE) models capture the continuous dynamics of gene regulatory networks through Hill-type transcriptional kinetics. The chemical master equation, simulated via Gillespie's stochastic simulation algorithm (SSA), describes the discrete molecular fluctuations that drive phenotypic heterogeneity when copy numbers are low. Partial differential equations of the reaction-diffusion type—exemplified by the Gray–Scott model—account for the spatial self-organization of morphogens that underlies developmental patterning [1]. Agent-based models (ABMs) represent biological systems as collections of discrete individuals whose local interaction rules generate emergent tissue-scale behaviors [2]. Bayesian parameter inference via the Markov chain Monte Carlo (MCMC) method closes the loop, estimating the posterior distribution of model parameters from experimental observations [3]. Each of these modeling frameworks has been extensively studied from a mathematical perspective: their stability properties, bifurcation structures, and asymptotic behaviors are well characterized.

What has not been systematically characterized is the computational cost of implementing these models. Every mathematical model must ultimately be translated into executable code, and the choice of programming language for this translation carries quantitative consequences that are rarely subjected to empirical measurement. Identical mathematical formulations—the same ODE system with the same Butcher tableau and the same SSA with the same propensity functions—can exhibit order-of-magnitude differences in execution time depending solely on the language in which they are encoded. These differences are not uniform across modeling frameworks. A language that executes a floating-point-intensive ODE integration efficiently may impose substantial overhead when applied to branch-intensive stochastic simulation or memory-bandwidth-limited spatial modeling. The software implementation layer, often treated as a transparent medium between mathematics and computation, introduces a regime-dependent performance structure that has remained largely unmeasured across the modeling frameworks of mathematical biology.

This gap has practical significance. A systems biology researcher who configures a stiff ODE model for metabolic network dynamics, executes a Gillespie SSA for gene expression noise, and submits a parameter sweep across a computing cluster [4] operates across multiple computational regimes within a single research trajectory. A structural biologist who submits a molecular dynamics trajectory involving tens of millions of force-field evaluations [5], then feeds the output into a Bayesian parameter estimation chain, compounds these demands. Whole-cell models encompassing tens of millions of parameters [6], large-scale structure prediction frameworks [7], and organ-scale digital twins [8] push these requirements further. In each case, the choice of programming language determines the practical reach of the modeling effort: a parameter space that can be explored in hours in one language may require days in another.

The software ecosystem of mathematical biology has evolved around a dual-language architecture. Mainstream simulation engines—GROMACS 5 [9], OpenMM 7 [10], COPASI 4.0 [11], and msprime 1.0 [12,13]—anchor performance-critical numerical kernels in C or C++ and expose user-facing interfaces in Python 3.13.12 or R 4.3.3 (the pinned versions used in this study) [14,15]. This division leverages the complementary strengths of each language tier but carries a documented maintenance burden: Yang et al. (2024) found that 92% of developers in multi-language codebases encountered cross-language interface defects [16], and the ongoing cost of bridging two linguistic systems has been identified as a significant source of software fragility [17].

A newer generation of languages has sought to unify mathematical specification and high-performance execution within a single framework. Julia, through LLVM-based just-in-

time compilation, provides numerical performance approaching that of natively compiled languages while maintaining a development experience comparable to that of Python or R [18–20]. Its SciML ecosystem—DifferentialEquations.jl for adaptive ODE solving and Catalyst.jl for reaction network specification [21]—offers an integrated toolchain from model construction to parameter inference. Rust provides memory-safe ahead-of-time compilation through its ownership model [22], with a growing bioinformatics toolkit spanning sequence analysis [23] and phylogenetic inference [24]. F#, operating on the .NET runtime, represents a functional paradigm whose performance characteristics in mathematical biology have not been systematically evaluated. Alongside these entrants, C, C#, Java, and Go span a range of compilation strategies and runtime designs.

What remains unknown is the quantitative performance relationship between programming language and modeling framework. For the model classes most commonly employed in mathematical biology—deterministic ODE systems, stochastic chemical kinetics, reaction-diffusion PDEs, agent-based models, and Bayesian inverse problems—what is the measured computational cost of each language, and does this cost structure transfer across modeling regimes? A measurement restricted to a single regime, however careful, cannot reveal whether the performance ordering observed there generalizes. ODE integration saturates floating-point units. Stochastic kinetics exercise branch prediction. Reaction-diffusion systems stress memory bandwidth. Agent-based models depend on dynamic allocation throughput. Parameter sweeps reward parallelism. Bayesian inference tests serial convergence speed. The computational character of the model determines which hardware resource becomes the bottleneck, and different language runtimes manage each resource with differing efficiency.

This study presents a systematic measurement framework spanning ten programming languages and six mathematical biology modeling regimes. The framework records execution time, peak memory footprint, cyclomatic complexity, type-conversion density, and the semantic alignment between data structures and biological state representation. These dimensions capture complementary aspects of computational cost: the wall-clock expense of model execution, the memory constraint that governs deployment density, the code-structural burden that language design imposes on implementation, and the translation overhead between biological concepts and their computational encoding. The output is a quantitative performance map: for each modeling regime, the measured runtime behavior of every language, the memory-footprint penalty of each runtime, and the code-complexity signature of each implementation. The goal is not to declare a single best language but to make the performance structure of the implementation layer-visible so that mathematical biology researchers can select computational tools with the same quantitative rigor they apply to model selection.

2. Prior Soundings

Three streams of prior work inform the present measurement campaign. Each stream illuminates part of the cost landscape. None provides the full atlas.

2.1. Cross-Language Runtime Measurement

Systematic measurement of programming-language cost in biological computing begins with Fourment and Gillings (2008), who compared three bioinformatics algorithms across six languages [25]. Their measurements placed managed-runtime implementations at roughly 60× the wall-clock cost of native implementations for the same tasks. Heng Li's biofast survey [26] expanded coverage to 25 languages but remained confined to sequence I/O and interval-query operations. Costanza et al. (2019) introduced the GBh composite metric in a Go/Java/C++17 comparison of NGS pipelines [27]. Bäckström

(2025) jointly assessed runtime cost and developer output in data-intensive settings [28]. What these investigations share is a task domain outside biosimulation regimes. None addressed the cost structure of deterministic ODE integration, stochastic chemical kinetics, parameter sweeping, reaction-diffusion dynamics, agent-based modeling, or Bayesian parameter inference.

Several earlier contributions broadened cross-language measurement beyond raw wall-clock time. Prechelt (2000) incorporated code length and programmer output alongside runtime measurements [29]. Nanz and Furia (2015) introduced the program failure rate as an additional dimension of code quality cost [30]. Bergmans et al. (2021) employed the Gzip compression ratio as a conciseness proxy [31]. Each of these established that cross-language cost comparison benefits from multidimensional assessment, though their task domains remained separate from biological simulation.

On the complementary dimension of energy cost, Pereira et al. (2017, 2021) demonstrated that native implementations draw roughly 5% of the energy consumed by managed-runtime counterparts [32,33]. This finding carries direct budgetary implications for large-scale computational biology, where parameter sweeps and ensemble simulations impose substantial aggregate energy expenses on institutional computing infrastructure.

2.2. Biosimulation-Specific Tool Assessments

Within the narrower domain of biosimulation-specific measurement, several tools have characterized runtime cost on individual compute regimes. Catalyst.jl surveyed ODE and SSA tooling costs within the Julia ecosystem [21]. SciMLBenchmarks has continuously tracked working-precision ODE solver behavior across languages [34]. Cayenne compared Gillespie SSA execution cost among Python, R, and Julia [35]. Each of these efforts covers one regime or, occasionally, two. No prior measurement campaign spans all six regimes (deterministic integration, stochastic kinetics, parameter exploration, spatial dynamics, agent-based models, and inverse inference) within a single, uniformly calibrated framework.

2.3. Software Engineering Costs in Scientific Computing

The software engineering dimensions of scientific computing have attracted increasing attention. Wilson et al. (2014, 2017) codified best-practice frameworks for scientific software construction [36,37]. These guidelines address version control, testing discipline, and modular design. They do not measure the runtime cost implications of language choice. Yang et al. (2024) surveyed the prevalence of cross-language interface friction, reporting that 92% of developers in multi-language codebases encountered interface defects [16]. This finding quantifies a maintenance overhead intrinsic to the dual-runtime architecture prevalent in the field: the interface between C/C++ numerical kernels and Python/R analysis layers acts as a systematic source of software brittleness [17].

Table 1 situates the current measurement campaign within prior empirical literature. Existing investigations sort into two broad groupings. The first grouping, represented by Fourment and Gillings; Li, Costanza et al.; and Bäckström, conducts comparative runtime measurement but does not extend its coverage to biosimulation regimes. The second grouping, represented by Prechelt, Nanz and Furia and Bergmans et al., broadens the measurement dimensions beyond execution time alone (into programmer output, program correctness, and code conciseness) yet applies these dimensions within general algorithmic domains rather than biological simulation. The current measurement campaign merges the two strategies: it applies multi-dimensional cost accounting (execution time, memory, cyclomatic complexity, type-conversion density, and biological-concept alignment) to six distinct biosimulation regimes within a single uniformly calibrated framework spanning ten

languages. The inclusion of Prechelt (2000) [29], Nanz and Furia (2015) [30], and Bergmans et al. (2021) bergmans2021conciseness in Table 1 acknowledges important prior work that expanded cross-language cost measurement beyond pure execution time, although those investigations differ from the present one in task domain or in the specific measurement dimensions assessed.

Table 1. Empirical measurement literature relevant to computational biology, tabulated by coverage dimensions. Columns indicate whether each investigation addressed biosimulation regimes (Sim. regimes), memory consumption, energy usage, code-complexity metrics, or data-structure-to-biological-concept mapping.

Study	Year	Languages	Sim. Regimes	Memory	Energy	Code Complexity	Data Struct. Mapping
Fourment & Gillings [25]	2008	6	–	✓	–	–	–
biofast (Li) [26]	2020	25	–	✓	–	–	–
Costanza et al. [27]	2019	3	–	✓	–	–	–
Bäckström [28]	2025	6	–	–	–	–	–
Pereira et al. [32]	2017	27	–	✓	✓	–	–
Gmys et al. [38]	2020	6	–	–	–	–	–
Prechelt [29]	2000	7	–	✓	–	✓‡	–
Nanz & Furia [30]	2015	8	–	–	–	✓†	–
Bergmans et al. [31]	2021	multiple	–	–	–	✓	–
This study	**2026**	**10**	✓	✓	–	✓	✓

† Nanz and Furia (2015) [30] introduced the program failure rate as a supplementary code quality dimension. ‡ Prechelt (2000) [29] compared code length, runtime, and developer productivity, serving as an early pioneer of cross-language comparison beyond pure execution time. Code complexity encompasses multi-dimensional software quality metrics including control flow and cognitive load. Data structure mapping refers to the analysis of how biological concepts map to programming data structures. This table is non-exhaustive. A checkmark (✓) indicates that the investigation addressed the corresponding dimension; a dash (–) indicates that it did not. The bold row identifies the present study.

Building on ground established by Catalyst.jl [21], SciMLBenchmarks, Cayenne, and related prior efforts, the measurement framework extends multi-language cost accounting from isolated compute regimes to six categorically distinct regimes while sustaining broad language coverage: ten languages spanning six execution paradigms. The multi-dimensional measurement perspectives (cyclomatic complexity, cognitive complexity, type-conversion density, and biological-concept alignment) are recorded alongside execution time and memory consumption, constructing a systematic framework for quantifying how language choice shapes the computational, structural and semantic costs of biosimulation implementations.

3. Measurement Apparatus

3.1. Regime Taxonomy

Every simulation sits inside a compute class. The class dictates which hardware resource saturates first. Floating-point units run dry. Branch predictors mis-step. Memory channels stall. Knowing the regime before choosing the language is the central organizing principle of this apparatus.

Six compute classes were selected. Each isolates one dominant bottleneck on a contemporary out-of-order core. The taxonomy is not exhaustive of systems biology. It is exhaustive of the bottleneck types that differentiate language runtimes. Low-dimensional models were chosen deliberately, owing to their modest state vectors and lack of stiffness-driven adaptive step rejection cascades and ill-conditioned Jacobians. Numerical confounds are drained from the comparison. What remains is the runtime itself, measured against bare silicon.

Table 2 presents the field guide. Each entry records three diagnostic signals. The dominant bottleneck names the hardware resource under heaviest demand. The expected native-to-managed cost ratio gives the approximate multiplicative factor separating minimal-runtime native code from managed-runtime execution based on prior cross-language

measurements [25,26,32]. Sensitivity to implementation quality captures how strongly naive versus idiomatic coding shifts the measured cost. A regime marked "high" sensitivity will punish careless data-structure choices, irrespective of the language tier.

Table 2. Regime-taxonomy field guide. Six compute classes are each characterized by the dominant hardware bottleneck, approximate native-to-managed cost ratio, and sensitivity to implementation craft.

Compute Class	Dominant Bottleneck	Native/Managed Ratio	Implementation Sensitivity
Deterministic ODE (Repressilator)	FP throughput	∼10×–100×	Low
Stochastic kinetics (Gillespie SSA)	Branch prediction	∼30×–60×	Medium
Parameter sweep	Parallel scheduling + FP throughput	∼50×–150×	Medium
Spatial dynamics (Gray–Scott)	Memory bandwidth	∼30×–70×	Medium
Agent population (ABM)	Dynamic allocation + branch density	∼20×–40×	High
Inverse inference (MCMC)	FP throughput (amplified)	∼50×–100×	Low

All models are low-dimensional, analytically tractable reference cases. For larger or higher-dimensional models, the reported cost ratios represent lower bounds. Native/managed ratio ranges span observed values across the language coverage matrix under the standard protocol. Implementation sensitivity: low = the algorithm's computational structure dominates regardless of coding style; medium = idiomatic patterns measurably shift cost within a given tier; high = data-structure selection and allocation discipline can alter cost by an order of magnitude or more.

The six classes originate from a single biological reference point: the Repressilator, a synthetic three-gene network with Hill-type transcriptional repression [39]. From this kernel, the taxonomy follows the natural sequence of questions in systems biology. Deterministic kinetics lead to stochastic noise exploration [3,40]; parameter-space characterization, exposing robustness properties [41]; spatial pattern formation, tracing Turing mechanisms [1,42]; discrete agent interactions at tissue scale [2,43]; and, finally, the inverse problem of estimating parameters from observed phenotypes. Each transition alters the bottleneck profile. The measurement apparatus was constructed to capture these transitions cleanly. In practical systems biology, model exchange relies on formal representation languages such as SBML [44], CellML [45], and Antimony [46]. The measurement protocol operates at the level of the numerical solver and its host-language runtime; representation-layer infrastructure falls outside the measured cost envelope.

3.1.1. Deterministic ODE Integration

The Repressilator [39] supplies the floating-point-intensive regime. Three transcription factors with mutual Hill-type repression evolve under mass-action kinetics. Hand-coded implementations (C, Rust, Go, Java, C#, and F#) uniformly adopt the Dormand–Prince 5(4) method. Library-calling languages invoke Python (SciPy RK45), Julia (DifferentialEquations.jl Tsit5), R (deSolve lsoda), and C++ (Boost.ODEint `runge_kutta_dopri5`). The relative tolerance was set to 10^{-6}, and the absolute tolerance was set to 10^{-8}. The full mathematical formulation is provided in Supplementary Material S1.

3.1.2. Stochastic Kinetics

The Gillespie direct method [47] drives a simplified gene expression model with positive auto-regulation across four reaction channels [3,40]. The direct method was chosen for its minimal algorithmic core: approximately ten scalar operations and multiple conditional branches per event, forming a stress test for branch-dense, non-vectorizable code. All implementations share a PRNG seed (42) and simulate 50 trajectories to $T_{\max} = 1000$. The full mathematical formulation is provided in Supplementary Material S1.

3.1.3. Parameter Sweep

A two-dimensional scan across the transcription rate (α) and Hill coefficient (n) generates 2500 independent ODE integrations. Each trajectory runs to $t \in [0, 200]$ and undergoes peak-detection classification after discarding the first 50% as transient. Systems biology models are generally sloppy [41]; this regime captures the computational cost of parameter robustness characterization. The embarrassingly parallel structure exercises each language's preferred concurrency framework on 16 threads. The full parameter ranges are provided in Supplementary Material S1.4.

3.1.4. Spatial Dynamics

The Gray–Scott model [48] serves as the memory-bandwidth-limited regime. Two chemical species evolve on a two-dimensional grid under local reaction and diffusion terms. All implementations use the explicit Euler method, periodic boundary conditions, a double-buffering strategy, and 5000 steps on a 128×128 grid. The five-point stencil keeps arithmetic intensity low. Memory subsystem cost dominates. The full mathematical formulation is provided in Supplementary Material S1.

3.1.5. Agent Population

Cells move and divide in a two-dimensional continuous domain under three rules: Brownian motion, contact-inhibition check, and probabilistic division [2,43]. Starting from 200 cells, the population saturates near 600 under contact inhibition. All implementations adopt a unified spatial hash-grid strategy, reducing pairwise distance complexity from $O(N^2)$ to $O(N)$. This regime stresses dynamic allocation and branch density simultaneously. The full design justification is provided in Supplementary Material S1.6.

3.1.6. Inverse Inference

The Repressilator ODE is embedded within a Metropolis–Hastings chain [49] to jointly infer three parameters ($\theta = (\alpha, n, \beta)$). Synthetic observations are generated from the ODE at true parameter values with additive Gaussian noise under log-normal priors. The chain runs $M = 2000$ iterations, with the first 500 discarded as burn-in. Each iteration performs one complete ODE integration; the multiplicative amplification of per-integration cost across the chain constitutes the measurement target. This regime serves strictly as a compute class for quantification of cumulative runtime differentials across languages. Convergence diagnostics (the $\hat{R}$ statistic, for instance) remain unverified. The full mathematical formulation is provided in Supplementary Material S1.

The measurement design adheres to the reporting standards of Hoefler and Belli (2015) [50] and the guidelines of Weber et al. (2019) [51], with additional reference to systematic measurement methodology [52]. No low-level manual optimization (SIMD intrinsics, assembly-level tuning, or compiler-flag adjustments targeting specific microarchitectures) was applied in any language. The comparison rests at the level of competent practitioners writing idiomatic code in their respective languages.

A methodological asymmetry exists in the ODE regime. Python, Julia, R, and C++ delegate adaptive integration to ecosystem libraries refined across decades: Python's SciPy calls FORTRAN ODEPACK; Julia's DifferentialEquations.jl integrates PI step-size control, Gustafsson acceleration, and Hermite interpolation; R's deSolve dispatches to the FORTRAN lsoda multistep solver; and C++ uses Boost.ODEint's Dormand–Prince stepper. Rust and F# implementations hand-code the integrator from the Butcher tableau coefficients. Both paths are ecologically valid: a Python practitioner would not hand-code RK45, just as a Rust developer would not hand-code RK45 once a mature library arrives. The apparatus therefore records practitioner-attainable cost, not the abstract cost of the language in isolation.

3.2. Language Selection

Ten languages populate the measurement matrix. Grouping by execution model, the approach of prior work [25,26], conflates categories that diverge sharply in measured cost. A more diagnostic grouping organizes languages by the runtime overhead they impose on the compute class. Three tiers emerge.

3.2.1. Low-Overhead Tier (C, Rust, and C++)

These languages generate native machine code directly, with no garbage collector and no runtime dispatcher interposed between the algorithm and the instruction stream. C operates with a minimal libc runtime. Rust adds ownership-based memory safety enforced at build time, with zero runtime cost outside allocation. C++ layers RAII and template-based abstraction onto the same bare-metal execution model. Cost differences within this tier arise from code-generation quality and standard library design, not from managed-runtime mechanisms.

3.2.2. Moderate-Overhead Tier (Go and Julia)

Go builds ahead of time to a self-contained native binary but embeds a concurrent garbage collector and goroutine scheduler inside every process [18]. The goroutine model delivers efficient concurrency but adds per-operation overhead in single-threaded hot loops. Julia translates via LLVM just in time on first invocation; subsequent calls execute native code directly. The JIT translation cost is paid once per function signature. The garbage collector, while generational and latency-tuned, introduces periodic pauses absent from the low-overhead tier.

3.2.3. Managed-Overhead Tier (Java, C#, F#, Python, and R)

Java executes on the HotSpot JVM with multi-tier optimization (C1/C2). The runtime layer provides dynamic class loading, bound checking, and a generational garbage collector. C# and F# execute on the .NET CLR with Tier-0/Tier-1 progressive optimization and dynamic profile-guided optimization [53]. The CLR's managed heap and method-dispatch infrastructure impose structural overhead distinct from that of JVM, though the two belong to the same overhead tier. Python operates through the CPython interpreter, with an adaptive specializing interpreter introduced in version 3.11; the bytecode dispatch loop remains the fundamental cost driver. R executes via the GNU R runtime with default immutable vector semantics that trigger full-vector copies on each mutation.

Table 3 maps each language to its overhead tier and runtime infrastructure. This tiered grouping, rather than the conventional native-versus-managed binary, proves predictive of measured cost patterns across the regime taxonomy.

The three-tier organization exposes patterns invisible to the execution-model grouping. C, Rust, and C++ sit together as the low-overhead reference. Go occupies a distinct intermediate position: its single-threaded cost approaches managed-overhead levels on branch-dense regimes, while its goroutine-based parallelism outperforms all other languages on embarrassingly parallel parameter sweeps. Julia straddles the moderate-overhead and managed-overhead boundaries depending on the regime: on floating-point-dense ODE integration, its JIT-specialized code approaches low-overhead tier cost, while on thread-scheduling-limited sweeps, its cost profile aligns with managed-overhead languages. Java, C#, and F# cluster together in measured cost across regimes, confirming that the managed runtime, not the execution model or syntax, dominates.

Table 3. Language coverage organized by runtime overhead tier. Languages within a tier share comparable per-operation cost profiles despite divergent syntax and build strategies.

Overhead Tier	Compilation	Language	Runtime
Low overhead	AOT (GCC)	C	libc
	AOT (GCC)	C++	libstdc++
	AOT (LLVM)	Rust	libc
Moderate overhead	AOT + GC	Go	Go runtime
	LLVM JIT	Julia	Julia runtime
Managed overhead	JVM JIT	Java	HotSpot JVM
	CLR JIT	C#	.NET CLR
	CLR JIT	F#	.NET CLR
	Bytecode VM	Python	CPython VM
	Bytecode VM	R	GNU R

Low-overhead languages execute as native machine code, with no managed-runtime layer. Moderate-overhead languages produce native code but embed a garbage collector (Go) or incur one-time JIT translation latency (Julia). Managed-overhead languages execute atop a virtual machine with dynamic translation, garbage collection, bound checking, and dynamic class loading. The distinction between moderate overhead and managed overhead is quantitative: both tiers carry runtime costs absent from the low-overhead tier, but the magnitude of those costs differs categorically. Exact language versions appear in Table 4.

Table 4. Compiler, interpreter, and dependency versions fixed across the entire measurement campaign.

Language	Compiler/Runtime	Version
Python	CPython	3.13.12
R	GNU R	4.3.3
Julia	Julia	1.11.5
Rust	rustc	1.95.0
C	GCC	13.3.0
C++	GCC	13.3.0
Go	gc	1.24.3
Java	OpenJDK	21
C#	.NET	9
F#	.NET	9

Key dependencies: Python—NumPy 2.4.4, SciPy 1.17.1, and OpenBLAS 0.3.27; R—deSolve 1.40; Julia—DifferentialEquations.jl 7.14.0; Rust—ndarray 0.17.2, rand 0.10.1, and rayon 1.12.0; C++—Boost 1.87.0 and OpenMP 4.5.

3.3. Measurement Protocol

All measurements were collected on a single dedicated machine: an Intel i7-12650H processor (6P+4E hybrid architecture, 10 cores, 16 threads, and 24 MB L3 cache) with 16 GB DDR5-4800 RAM and NVMe SSD storage, running Windows 11 with WSL2 Ubuntu 24.04. During the measurement windows, the system was isolated: there were no other user processes or background services, and the network interface was disabled. These precautions minimize OS scheduling jitter and interrupt-driven measurement noise.

The processor state was controlled to the extent permitted by the platform. Dynamic frequency scaling (turbo boost) and the per-core thermal state could not be locked because the hybrid P-core/E-core laptop processor under WSL2 exposes no stable frequency-lock interface to the measurement harness. Timing stability was instead enforced through three mechanisms: median aggregation over ten independent replicates, a single-threaded numerical environment for serial regimes (OpenBLAS pinned to one thread; Julia thread count set explicitly per regime), and system isolation, as described above. CPU affinity was not pinned in the initial campaign. Because the P-core/E-core topology can shift timing when the OS migrates threads, the revised protocol pins affinity via `taskset` and repeats the

scaling study on homogeneous P-core-only and E-core-only subsets; the resulting bounds on the scheduling artifact are reported in Supplementary Material S2. The effect of turbo boost and thermal drift is treated as residual, regime-independent measurement-noise source absorbed by the replicate-based dispersion statistics (Supplementary Material S6).

Software versions were pinned as recorded in Table 4. All low-overhead code was built with `-O3 -march=native`.

Identical numerical tolerances were maintained across all languages (relative, 10^{-6}; absolute, 10^{-8}). Hand-coded implementations (Rust, C, Go, Java, C#, and F#) uniformly adopt the Dormand–Prince 5(4) Butcher tableau. Library-calling languages (Python: SciPy RK45, and Dormand–Prince 5(4); Julia: DifferentialEquations.jl Tsit5 and Tsitouras 5(4) RK pair; R: deSolve lsoda and the Adams/BDF multistep method; C++: Boost.ODEint `runge_kutta_dopri5`) carry solver-level differences that are faithfully recorded and treated as qualifications in cross-language cost interpretation.

JIT warm-up was completed by executing a full run prior to all timing measurements for Julia, C#, and Java. For runtimes employing multi-tier optimization (HotSpot C2 and .NET CLR Tier-0/Tier-1/PGO), a single warm-up may be insufficient to trigger all optimization tiers. The reported cost for Java, C#, and F# should therefore be interpreted as the cost after the first adequate warm-up, not the absolute steady-state peak.

Each regime was executed in 10 independent measurement replicates (50 trajectories per language in the stochastic regime). The main text reports median values; interquartile range (IQR) and 95% bootstrap confidence intervals for the median are reported in Supplementary Material S6. Statistical claims are restricted to the tier-level comparisons, whose cost gaps exceed one order of magnitude and therefore remain robust to replicate-to-replicate dispersion. Fine-grained inter-language ranking differences *within* a tier (for instance, between C, C++, and Rust in the stochastic kinetics regime or between C# and F# in the ODE regime) are reported as descriptive comparisons only because the widths of their confidence intervals overlap; such orderings should not be read as statistical inferences. The measured cost ratios reported in the main text are point estimates of the median ratio and carry the same replication uncertainty.

Figure 1 provides an overview of the measurement apparatus. The complete implementation corpus and raw measurement data will be released as open source upon paper acceptance, following FAIR4RS principles [54].

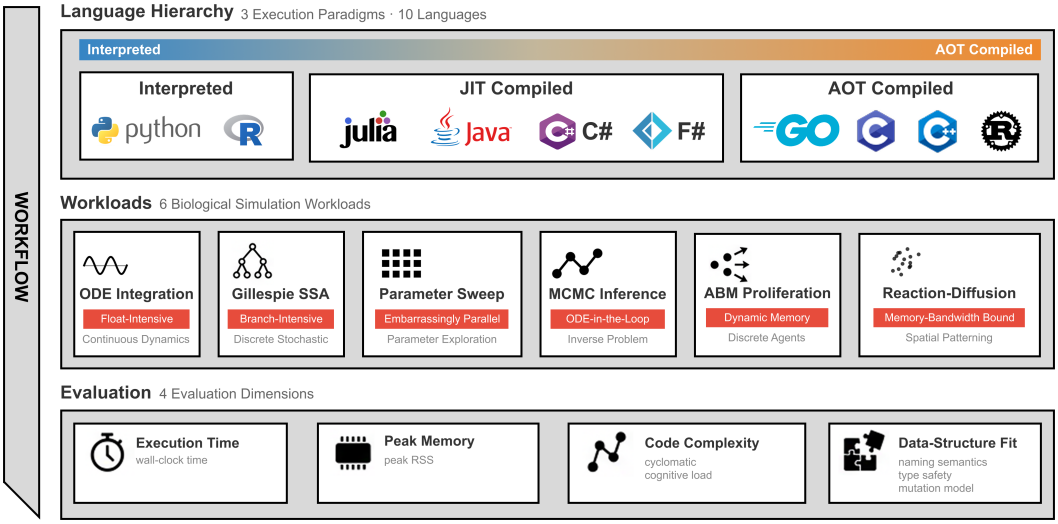


Figure 1. Measurement-apparatus overview. Ten languages cross six compute classes under a uniform protocol capturing runtime cost, memory cost, code-complexity cost, and representation cost.

3.4. Cost Metrics

Four cost dimensions constitute the measurement framework. The first two are quantitative; the latter two are structural and qualitative. Together, they capture not merely how expensive a language is to execute but how expensive it is to reason about and to maintain.

3.4.1. Runtime Cost

Wall-clock duration is measured within the same process as the computation using each language's highest-precision monotonic clock. Process-creation overhead and JIT translation latency are excluded. Measurement anchors the primary quantitative axis.

3.4.2. Memory Cost

Peak physical memory is recorded as the maximum resident set size (RSS) via GNU `time` (`-v`). RSS was chosen over virtual memory to avoid overestimating true memory pressure from reserved but uncommitted address space.

3.4.3. Code-Complexity Cost

Source lines of code (SLOC) is measured using `cloc` v2.00, counting logical lines and excluding auto-generated code and third-party dependencies. SLOC exhibits low correlation with cognitive load and defect density [55,56]. Therefore, three supplementary metrics are integrated. Cyclomatic complexity [57] counts independent control-flow paths in each language's core algorithm function. Abstraction depth records the maximum function-call depth from entry point to innermost computation, distinguishing library-wrapped implementations (Python `solve_ivp` conceals two layers) from hand-coded implementations (Rust's explicit Butcher stepping requires five). Type-conversion density counts explicit integer-to-floating-point conversions per iteration in the stochastic kinetics hot loop (for instance, Rust's `as f64` and C++'s `static_cast<double>`), serving as a proxy for the syntactic overhead imposed by type strictness.

3.4.4. Representation Cost

This qualitative analysis captures how naturally each implementation maps biological concepts onto programming-language constructs. Two sub-dimensions are recorded. Naming semantics track whether state variables are accessed by biological names (e.g., `m1`, `p3`) via destructuring bindings, or only through implicit array indices (e.g., `y[0]`, `y[5]`). Mutation semantics record the alignment between a language's default mutation model and the mutation demands of the algorithm: for the stochastic kinetics regime, the core operation is discrete state mutation, and for the agent population regime, the core operation is dynamic container growth. Julia (the `!` naming convention renders mutating functions explicit) and Rust (the `&mut` annotation enforces distinction of mutable borrows at build time) expose mutation as a semantic construct, aligning naturally with mutation-intensive computation. Python adopts an implicitly mutable model with negligible additional overhead in fine-grained mutation contexts. R's default immutable vector semantics cause each `vec <- c(vec, new_cell)` append to trigger a full-vector copy, producing structural conflict between data structure mutation assumptions and algorithmic requirements. F# occupies a mixed zone: core functional types are immutable by default, but arrays are mutable by default [53]. The full encoding matrix and scoring anchors appear in Supplementary Material S5. This analysis does not rank languages as superior or inferior. It documents code-characteristic differences that carry substantial impact on day-to-day usability beyond raw cost numbers [36].

4. Regime-by-Regime Cost Profiles

Figure 2 summarizes the absolute execution times recorded for all ten languages across the six regimes. The computational cost of biological simulation is not a property of any language. It is a property of the intersection between a language's runtime architecture and the specific hardware bottleneck that a given compute pattern activates. Each of the six regimes examined below stresses a distinct resource. The sections that follow itemize, for each regime, what each language charges in runtime, memory, and code structure.

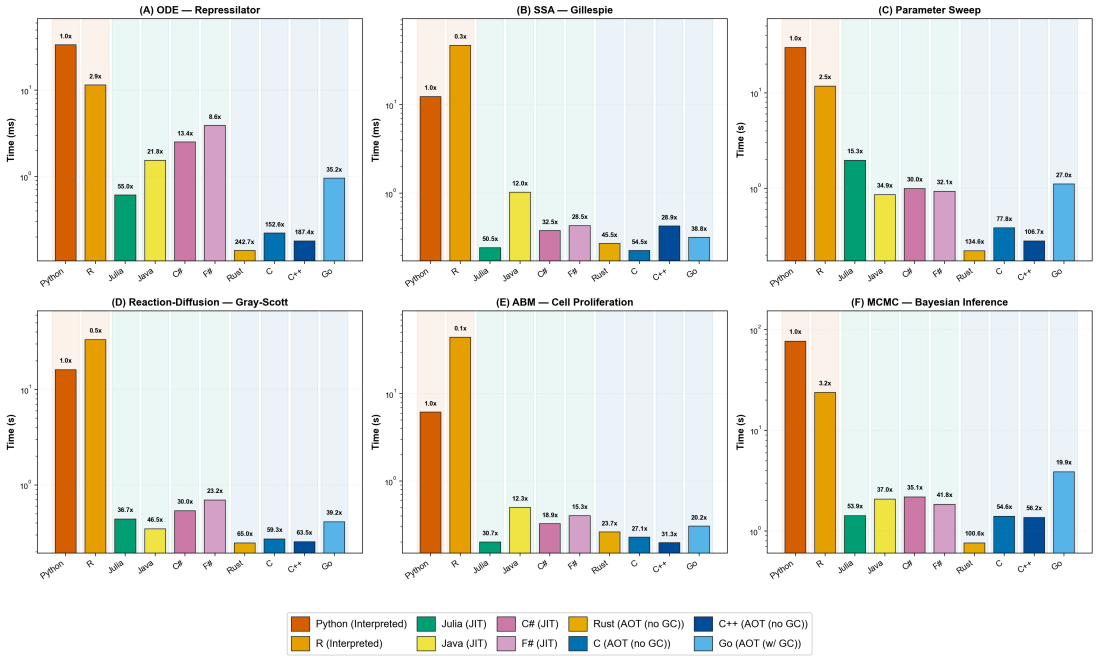


Figure 2. Absolute execution times of ten programming languages across six biological simulation regimes (log scale). Labels atop bars record the cost ratio relative to Python (1.0×). Background shading groups languages by execution model: orange = managed runtimes without JIT (Python and R); green = JIT-compiled managed runtimes (Julia, Java, C#, and F#); blue = native ahead-of-time binaries (Rust, C, C++, and Go). Panels (**A**–**F**) correspond to ODE, SSA, Sweep, RD, ABM, and MCMC.

4.1. Continuous Integration (ODE)

The ODE regime registers the widest cost spread of any regime in the measurement suite. The floating-point pipeline is the sole bottleneck; branch density is negligible, dynamic allocation is absent, and memory bandwidth is untaxed. Every processor cycle can be dedicated to evaluating the right-hand-side function of the Repressilator system, which contains six `pow` calls and 18 scalar operations per invocation.

The top of the cost hierarchy is occupied by the hand-coded native binaries. Rust delivers a 97× runtime advantage over the hand-coded Python baseline, the single largest cost differential recorded under the unified solver. C++ follows, at 74×, through Boost.ODEint's Dormand–Prince stepper, with C at 68× through a hand-coded integrator. These three languages share a structural property: no garbage collector, no runtime dispatch layer, and no just-in-time compilation thread competing for execution resources. The compiler can schedule floating-point instructions across SIMD lanes, allocate registers across loop bodies, and fuse chains of arithmetic operations into compact basic blocks. Julia occupies the next tier, at 21×, drawing on DifferentialEquations.jl's Tsit5 solver, a Tsitouras 5(4) Runge–Kutta pair with PI step-size control and Gustafsson acceleration. The remaining managed runtimes form a lower tier: Go, at 13×; Java, at 8×; C#, at 4.6×; and F#, at 3.7×. The most consequential result concerns the two library-delegating languages. When R's FORTRAN-

backed lsoda solver is replaced by a hand-coded Dormand–Prince integrator, R drops to 0.22×—slower than the hand-coded Python baseline—while the hand-coded Python integrator, itself (12.5 ms), runs nearly three times faster than SciPy's RK45 (36.2 ms). The absolute execution durations clarify the stakes: Rust completes a single ODE integration in 0.129 ms; hand-coded Python, in 12.5 ms; and SciPy-backed Python, in 36.2 ms.

Memory cost in the ODE regime splits along a clean structural fault line. Native binaries occupy 1.7–4.4 MB. C is the leanest, at approximately 2 MB; Rust is near-constant, at 2.5 MB; and C++ requires 3.8–4.4 MB from dynamic linking of libstdc++. Managed runtimes without JIT fall in the 15–106 MB band, with one instructive exception: the hand-coded pure-Python integrator, which imports neither NumPy nor SciPy, runs in only 12.6 MB—approaching native-binary territory and revealing that the 77 MB footprint of the SciPy-backed variant is dominated by the library ecosystem rather than the CPython runtime itself. JIT-compiled managed runtimes carry a fixed infrastructure tax of roughly 165–172 MB for .NET languages and 43–787 MB for Java. Julia's ODE profile is exceptional: loading the full DifferentialEquations.jl dependency graph pushes the peak resident set size to 763.0 MB. The library-ecosystem cost, not the language runtime itself, dominates Julia's memory footprint in this regime.

Code structural cost in the ODE regime reflects the library-delegation divide. Python, Julia, and R encapsulate solver logic behind two abstraction layers, yielding a user-facing cyclomatic complexity of 4. Hand-coded integrators require four to five abstraction layers (Butcher stepping, error estimation, and step-size control) with CC values of 5. Source lines vary from C#'s 18 (leveraging .NET top-level statements) to Rust's 183 (including explicit Butcher-coefficient array literal transcription of 40 coefficients). The structural price of hand-coded numerical infrastructure is not algorithmic complexity but transcription volume.

4.2. Stochastic Kinetics (SSA)

The SSA regime inverts the cost hierarchy established by ODE. Floating-point throughput recedes, and conditional branching becomes the dominant resource consumer. The Gillespie direct method checks reaction propensities against random thresholds inside a tight per-event loop. Each branch misprediction flushes the processor pipeline at a cost of 15–20 cycles. Vectorization offers no relief because control-flow divergence between trajectories defeats SIMD. The branch predictor becomes the universal bottleneck.

C captures the top position at 54.5×. The runtime footprint underlying this result is the barest in the measurement suite: no garbage collector, no exception-unwinding tables, and no managed-runtime bookkeeping layer. Each event-loop iteration executes nothing beyond the compiled machine code and a direct glibc `random()` call. Julia follows at 50.5×, with its LLVM-based JIT producing compact branch sequences that fit within the instruction cache. Rust drops to 45.5×, a contraction from its ODE standing that reflects the shifting bottleneck: ownership-checking machinery, however lightweight, adds measurable cost when millions of branch-dense iterations replace floating-point throughput as the throughput governor. Go reaches 38.8×, C# 32.5×, C++ reaches 28.9×, and F# reaches 28.5×. The .NET tier's relative strength on SSA compared to ODE constitutes a regime inversion: C# achieves roughly seven times its ODE cost ratio (32.5× versus 4.6×), and F# achieves nearly eight times its ODE cost ratio (28.5× versus 3.7×). The Tier-1 profile-guided optimizer specializes the reaction-selection hot loop efficiently. Java falls to 12.0×, reflecting HotSpot's difficulty with dense per-event branching that resists optimization. R prints the only sub-baseline result at 0.3×, completing each SSA trajectory at greater absolute cost than Python.

Memory profiles in the SSA regime are the most compressed of any regime. Native binaries occupy 1.7–2.5 MB. Managed runtimes show their lowest footprints: Java at

43.4 MB, C# at roughly 166 MB and Julia at 303.8 MB (without the DifferentialEquations.jl ecosystem). The low memory pressure reflects the minimal working set of the Gillespie algorithm: four reaction channels, two state variables, no spatial grid, and no solver library.

Code complexity reveals a structural invariant. Every one of ten implementations registers a core-function cyclomatic complexity of exactly 6 on the SSA, forced by the irreducible branching structure of the direct method. The algorithm itself imposes a floor on control-flow complexity that no language can circumvent. Cognitive complexity, however, diverges. F# records the only "High" score, driven by an empty `then () else` branch idiomatic in the match-expression style. Rust's ownership annotations and explicit scope delimiters raise cognitive load to medium. Type-conversion density captures a dimension of static-typing cost unique to this regime: Rust demands six explicit `as f64` casts per Gillespie iteration; C++ requires four `static_cast<double>`; and Go, Java, and F# require four conversions. Python, Julia, and R pay zero because implicit numeric promotion absorbs the integer-to-floating-point transition. Whether these conversions represent a documentation asset or a coding tax depends on the debugging context; an `as f64` annotation flags a precision-sensitive numeric transition that implicit promotion would conceal [58].

4.3. Parameter Exploration (Sweep)

The Sweep regime unbundles the ODE cost from serial execution and redistributes it across a parallel framework. Two thousand five hundred independent ODE integrations execute on 16 threads with no inter-task data dependencies. Single-core throughput and parallel-scheduler efficiency jointly determine the aggregate cost. Thus, the regime surfaces a dimension absent from the purely serial measurements: thread-coordination overhead.

Rust leads, at 142.4×, followed by C++, at 94.7×, and C, at 72.5×. These three native binaries preserve their relative ordering from the ODE regime but widen the differential: Rust's rayon work-stealing scheduler extracts higher parallel efficiency than the OpenMP-based dispatch used by C and C++. Java reaches 32.5× through ForkJoinPool, followed by F#, at 31.0×, and C#, at 27.0×, through the .NET Task Parallel Library. Go delivers 23.1× but compensates with the most favorable parallel scaling of any language: goroutine-based work distribution achieves a 6.4× scaling factor at 16 threads (normalized time 15.5), which is the highest parallel efficiency recorded. Julia collapses to 6.6×, representing a roughly 3× drop from its ODE standing. The `@threads` cooperative scheduler encounters thread-yield checks at a frequency (approximately 480 µs per integration) where coordination overhead consumes a substantial fraction of the per-task time budget. Julia reaches a normalized parallel time of 71.8 at 16 threads, corresponding to a 1.4× scaling factor. The hand-coded R sweep does not finish within the measurement window (exceeding 400 s), whereas its deSolve lsoda implementation completes in 0.042 s—a four-order-of-magnitude penalty that exposes the cost of interpreted R loops once the FORTRAN library crutch is removed.

Memory cost in the Sweep regime exposes the structural burden of parallel task infrastructure. Java surges to 787.1 MB, the second-highest value recorded, driven by ForkJoinPool's per-thread work queues and task-object allocation under 16-way parallelism. Julia reaches 846.4 MB, the maximum across all language-regime combinations, reflecting the combined load of the DifferentialEquations.jl dependency graph and parallel thread stacks. Native binaries remain in the 2.4–8.6 MB range, with Go's 8.6 MB elevated by goroutine stack pre-allocation (2500 goroutines each, with an initial 2 KB stack). The optimized .NET variants, C#-opt and F#-opt, which pre-allocate buffers to eliminate garbage collection inside parallel hot loops, improve normalized parallel time from 33.5 to 17.0 and from 35.5 to 16.7, respectively. The 16.5-point and 18.8-point differentials quantify the structural tax of GC in parallel floating-point contexts.

Figure 3 records the parallel scaling behavior of all ten languages across 1 to 16 threads. The measured data confirm that the parallel-framework architecture constitutes a differentiating dimension equal in weight to single-core throughput. The non-monotonic fluctuations in the 12–16-thread region (Rust rises slightly from 19.1 at 12 threads to 19.6 at 16) reflect the hybrid P-core/E-core scheduling dynamics of the i7-12650H processor. Code complexity in the Sweep regime bifurcates by implementation strategy: languages delegating to library solvers report user-facing CC of 3–5, while hand-coded integrators expose nested adaptive-step logic, driving CC to 8–12. This split captures the structural trade-off between ecosystem leverage and implementation transparency.

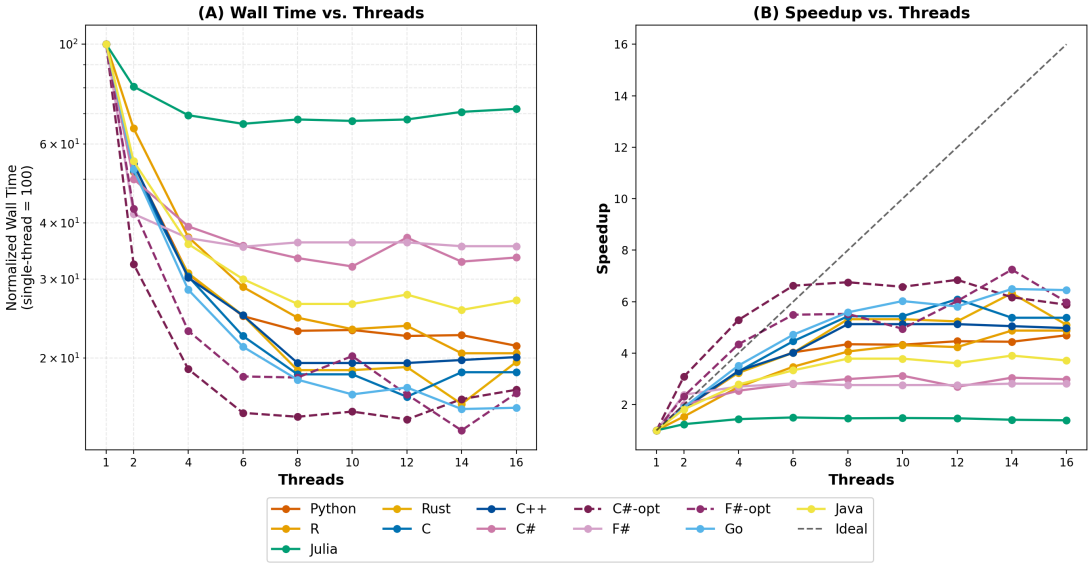


Figure 3. Parallel scaling efficiency of the parameter exploration regime (2500 ODE integrations) on i7-12650H (6P+4E cores, 16 threads). Panel (**A**): normalized wall-clock time versus thread count (single-thread = 100, log Y-axis). Panel (**B**): speedup versus thread count (dashed line = ideal n-fold speedup). C#-opt/F#-opt (dashed) represent optimized variants with pre-allocated memory to eliminate GC in hot loops. The complete data table is presented in Supplementary Material S2.

4.4. Spatial Dynamics (RD)

The reaction-diffusion regime operates under a different governor than the regimes examined above. The Gray–Scott model's five-point stencil on a 128 × 128 grid exhibits arithmetic intensity low enough that the memory subsystem, not the processor core, controls throughput. Every stencil update reads five cells and writes one. The processor spends the majority of its execution budget waiting for cache-line fills from main memory. This bandwidth constraint functions as a universal ceiling that compresses cost differentials across languages.

The three native binaries without garbage collection converge within a remarkably narrow band: Rust at 65.0×, C++ at 63.5×, and C at 59.3×. The compression of compiler-level optimization differences to a range spanning less than 10% constitutes indirect evidence that memory bandwidth, not instruction-level throughput, is the binding constraint. Go occupies a second tier, at 39.2×, consistent with additional pressure on the memory channel from concurrent GC bookkeeping. Java reaches 46.5×, which is the highest cost ratio of any managed runtime on any regime, reflecting HotSpot's capacity to compile the stencil loop's repeated array-indexing pattern into near-native sequences. Julia registers 36.7×, C# registers 30.0×, and F# registers 23.2×. Python serves as baseline, with a 16.131 s wall-clock time. R falls below the baseline, at 0.5×, completing at 33.325 s.

Memory footprints in this regime are minimal across all languages. C occupies 1.7 MB, corresponding to the leanest profile recorded. Rust holds at 2.5 MB. The stencil's working set (two 128 × 128 double-precision arrays, approximately 256 KB) is negligible relative to runtime overhead. Deployment implications follow directly: when memory bandwidth, not core count, is the bottleneck, language selection pivots from raw throughput toward factors orthogonal to execution speed (ecosystem maturity, team familiarity, and visualization bindings). The code complexity in the RD regime shows a second structural invariant: every implementation falls within the CC = 11–15 band, reflecting the flat inlined stencil approach adopted uniformly across languages. The algorithmic structure, not language choice, determines the control-flow fingerprint.

4.5. Discrete Agents (ABM)

The ABM regime shifts the cost bottleneck from arithmetic throughput and memory bandwidth to dynamic memory allocation and data-structure semantics. Cells are created through division events, appended to a growing container, and spatially indexed through a hash grid that maps continuous positions to discrete bins. Thus, the regime stresses three distinct resources simultaneously: allocation frequency, the container resizing policy, and spatial-lookup efficiency.

C++ occupies the top position, at 31.3×, exploiting `std::vector`'s amortized-constant push-back semantics. Julia follows, at 30.7×, as one of only two regimes where a JIT-compiled language breaches the top tier. C reaches 27.1×. Rust drops to 23.7×, corresponding to a 10.2× contraction from its ODE standing, as the bottleneck migrates from floating-point throughput to dynamic `Vec::push` operations that trigger reallocation and copying. Go records 20.2×, C# records 18.9×, F# records 15.3×, and Java records 12.3×. R lands at 0.14×, which is the single lowest cost ratio in the entire measurement suite, completing the ABM workload in 44.539 s versus Python's 6.174 s.

The R result warrants isolation because it documents a structural mismatch between data-model assumptions and algorithmic requirements, not a generalized statement about computational capacity. R's default immutable vector semantics cause every `vec <- c(vec, new_cell)` call to instantiate a complete vector duplication. Across hundreds of division events, the cumulative cost transitions from inconvenience to infeasibility. The outcome exposes a regime where mutation semantics become a feasibility constraint rather than a throughput consideration. Implicitly mutable languages (Python, Java, C#, C, C++, and Go) pay no equivalent penalty for the same append pattern.

Memory footprints in the ABM regime remain modest across native binaries (2.5–4.4 MB) and managed runtimes (43–172 MB), as the working set of approximately 600 cells with spatial coordinates is negligible. Code complexity in the ABM regime produces the widest spread of any regime. Rust's `'outer`-labeled break compresses the contact-inhibition logic to CC=16. Triple-nested conditional checks in Python and C push CC to 29. C++'s RAII-based resource management avoids the explicit allocation/deallocation branching that contributes to C's ABM cyclomatic complexity. Cognitive complexity reaches "High" in three languages: Python (triple-nested conditionals), C (manual `malloc/free` branching), and F# (conflict between functionally immutable defaults and cell-list append operations). Thus, the ABM regime exposes a regime where implementation style, rather than language capability per se, drives structural complexity.

4.6. Inverse Inference (MCMC)

The MCMC regime amplifies per-call overhead through iteration. Each of 2000 Metropolis–Hastings steps embeds a complete ODE integration, evaluating the Repressilator system from $t = 0$ to $t = 100$ and computing the log likelihood against synthetic obser-

vations. The per-solve cost, which is negligible in a single invocation, is multiplicatively amplified across the chain. Cross-language bridging overhead, which is invisible in the ODE regime, becomes the dominant cost component.

Rust completes all 2000 iterations in 0.764 s, corresponding to a 120.7× advantage over the hand-coded Python baseline (92.2 s). C++ follows, at 67.4× (1.367 s), followed by C, at 65.5× (1.407 s), and Julia, at 64.6× (1.427 s). The compression between Julia and the native tier relative to the ODE regime reflects a structural property: Julia's DifferentialEquations.jl Tsit5 solver, which contributed a substantial advantage in the single-solve ODE measurement, delivers proportionally less benefit when the dominant cost shifts to the per-iteration chain infrastructure. F# records 45.0×, which is the highest cost ratio of any managed runtime on this regime, benefiting from pattern matching over discriminated unions that maps directly onto the Metropolis–Hastings accept/reject decision structure. C# reaches 40.9×, Java reaches 39.2×, Go 21.7×, and hand-coded R reaches 0.83×. The Go result at 21.7× establishes the lower bound of native-binaries-with-GC performance in this regime, consistent with its concurrent GC overhead amplifying across 2000 iterations.

The MCMC regime quantifies the multiplicative tax of the ODE solve with unusual precision. In the single-solve ODE measurement, the hand-coded Python integrator outperformed SciPy's RK45, but the ranking reverses here: across 2000 iterations, SciPy's optimized solver core amortizes its fixed overhead, so the library-delegating Python implementation (76.9 s) finishes faster than the hand-coded implementation (92.2 s). This inversion illustrates that the library-versus-hand-coded trade-off is, itself, regime-dependent. The MCMC acceptance rates (Python, 0.232; C, 0.216; Rust, 0.227) fall within a healthy 0.22–0.23 range, but this masks a deeper limitation: convergence diagnostics computed across four overdispersed chains (Supplementary Material S8) yield $\hat{R}$ values of 159–208 and effective sample sizes of approximately 2.5, indicating that the chains do not converge under the configured proposal covariance. The MCMC regime therefore serves strictly as a compute class for quantification of cumulative runtime, not as a statistically valid inference; obtaining a converged posterior would require adaptive MCMC or a proposal adapted to the posterior geometry (see Limitations and Future Work).

Memory footprints span the full range. Native binaries occupy 2.5–4.4 MB. Python occupies roughly 77 MB. Java swells to 252.5 MB, reflecting HotSpot's metadata structures for 2000 iterations of ODE-solving objects. The memory cost in the inference regime carries deployment implications distinct from those in forward simulation: running multiple Markov chains in parallel for convergence diagnostics multiplies the per-chain memory footprint, and the native-versus-managed memory differential translates directly into how many independent chains can execute concurrently on a single node.

Code complexity in the MCMC regime reflects the deepest library-delegation divide of any regime. Python records a user-facing CC of 1, while Julia records a user-facing CC of 3 and R records a user-facing CC of 4; the solver and proposal logic are encapsulated behind library calls. Rust and C++ expose CC values of 12, reflecting explicit implementation of the Metropolis–Hastings loop, proposal generation, prior evaluation, likelihood computation, and ODE integration. The abstraction-layer metric reinforces this pattern: Python, Julia, and R hide the chain machinery behind two layers, while Rust, C, and C++ expose five layers of explicit stepping, error estimation, and accept/reject logic. The cumulative decision governing structural code cost in the MCMC regime is whether to treat the inference engine as a black-box library call or as a transparent numerical pipeline.

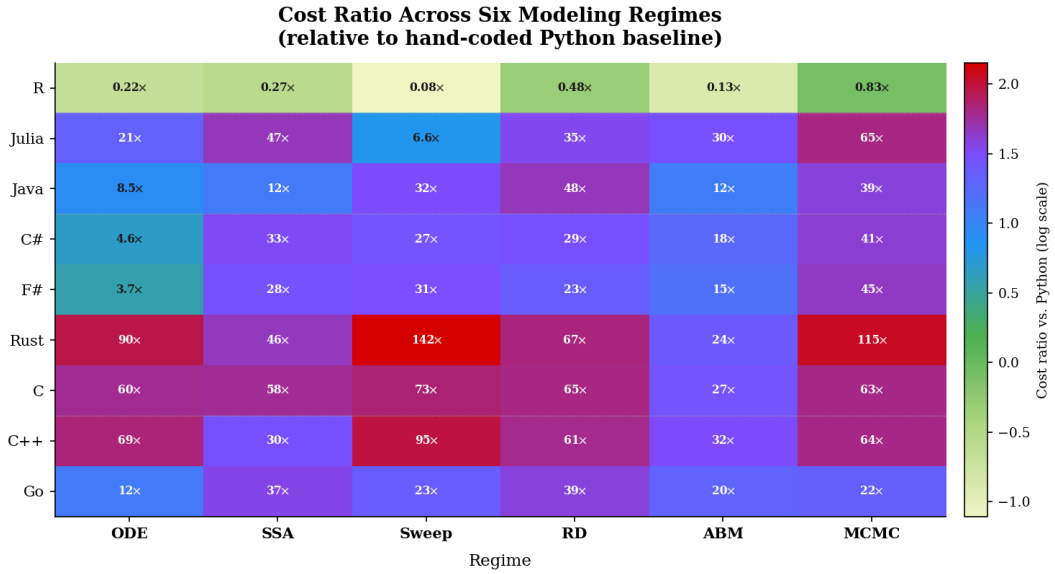
5. The Aggregate Picture

The six regime profiles presented in the preceding section itemize costs at the intersection of language runtime and hardware bottleneck. Stepping back from individual regimes

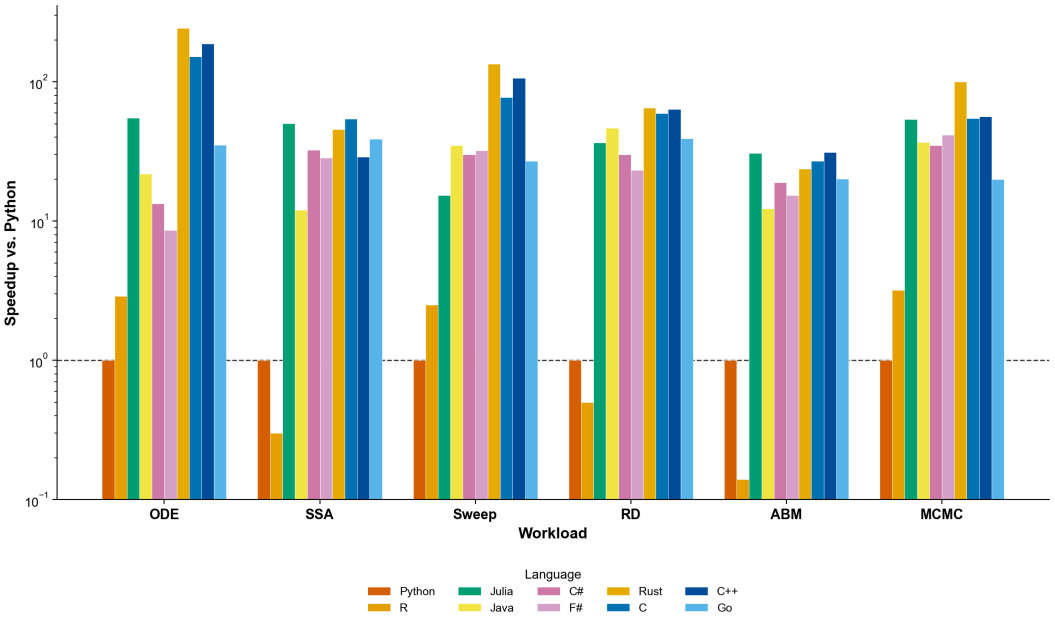
reveals cross-cutting patterns that persist across the measurement suite. Four such patterns organize the analysis below.

5.1. The Cost Gradient

The native-to-managed cost ratio is not a constant. It is a function of the compute regime. Moving across the six regimes traces a systematic gradient driven by arithmetic intensity per unit of runtime. Figure 4 maps the cost ratio of each language relative to Python across all six regimes.



(**a**) Cost ratio heatmap



(**b**) Cost-ratio-grouped bar chart

Figure 4. Cost ratios of ten languages relative to Python across six computational regimes. (**a**) Heatmap: color intensity encodes the runtime cost advantage on a log scale (light yellow = 1×, dark red = highest), with languages grouped by execution model. (**b**) Grouped bar chart (log Y-axis): the horizontal dashed line marks the Python baseline (1.0×); the systematic contraction in bar heights from ODE to SSA tracks the shift from floating-point-dominant to branch-dominant computation.

In the ODE regime, floating-point density is at its maximum. The right-hand-side function of the Repressilator evaluates six `pow` operations and 18 scalar computations per invocation. Native compilers exploit instruction scheduling, register allocation, and SIMD lanes to fuse chains of floating-point operations into compact machine sequences. A managed runtime processes each operand through a dispatch loop. The cumulative per-operation tax aggregates across millions of hot-loop iterations. The native-to-managed cost ratio spans 97× (Rust) to 3.7× (F#), with the hand-coded Python implementation as the reference baseline. The MCMC regime produces a similar but compressed gradient: 120.7× (Rust) to 21.7× (Go). The iterative amplification of per-solve overhead across 2000 Metropolis–Hastings steps widens the absolute cost but narrows the relative spread because the per-iteration chain infrastructure imposes a floor that managed runtimes cannot circumvent.

The Sweep regime introduces a second dimension to the gradient. Parallel-framework efficiency becomes a differentiating factor equal in weight to single-core throughput. Rust's rayon work-stealing scheduler (142.4×) and Julia's `@threads` cooperative scheduler (6.6×) represent the two extremes of thread-coordination cost. The cost spread between these two results is not a single-core differential but a parallel-efficiency differential: approximately 480 µs per integration crosses the threshold where Julia's thread-yield check overhead consumes the benefit of additional cores. Go's goroutine model achieves 6.4× parallel scaling, which is the highest factor recorded, demonstrating that raw single-core throughput and multi-core efficiency can diverge sharply within the same language.

As arithmetic intensity recedes, the gradient contracts. The reaction-diffusion regime compresses the native-to-managed ratio to a 65.0×–23.2× span. The memory-bandwidth ceiling compresses native-binary differences below 10%. The Gillespie SSA tightens the range further to 54.5×–0.3×, with dense conditional branching as the shared bottleneck. The ABM regime collapses the range to 31.3×–0.14×, as dynamic allocation and container-semantic penalties supplant arithmetic throughput as the cost governor. The gradient traced from ODE through MCMC through Sweep through RD through SSA to ABM is not a smooth monotonic function because the Sweep regime introduces the orthogonal dimension of parallel efficiency, but the overall trend is unmistakable: the cost of choosing a managed runtime over a native binary is highest when the processor spends its budget on arithmetic, moderate when it spends on allocation, and lowest when it spends waiting on the branch predictor or the memory channel.

Prior cross-language surveys that collapsed all regimes into a single cost multiplier therefore underdescribe the landscape that biological simulation practitioners navigate [26,32]. Fourment and Gillings (2008) reported a factor of approximately 60× for bioinformatics measurement regimes [25]. The envelope measured here exceeds both bounds of that figure, confirming that no single number characterizes the native-to-managed divide. The factor is regime-contingent. Hoefler and Belli (2015) established that single-configuration reporting is insufficient for characterizing multi-language cost spaces [50], and Godoy et al. (2023) demonstrated that relative ordering can undergo non-trivial reversals across CPU microarchitectures [59].

5.2. Memory as a Hidden Tax

The cost of runtime infrastructure emerges most starkly in memory. The peak resident set size varies by roughly two orders of magnitude across languages executing identical computational workloads. This variation is invisible to wall-clock measurements but governs deployment architecture and operational economics.

Figure 5 displays the memory landscape across all language-regime combinations. Native binaries occupy the leftmost positions in every panel, clustering within the 1.7–8.6 MB

band. C is the leanest, at 1.7 MB (RD regime); Go reaches 8.6 MB in the Sweep regime from goroutine stack pre-allocation. Rust's 0.4 MB cross-regime variation is the flattest profile recorded, reflecting static linking of the entire program into a single binary with no external shared-library dependencies. Managed runtimes without JIT occupy the intermediate 15–106 MB tier. JIT-compiled managed runtimes carry a fixed infrastructure tax: the .NET CLR imposes 165–172 MB across all regimes for both C# and F#, independent of the computational workload. Java's memory signature is the most regime-sensitive, spanning 43.4 MB (SSA) to 787.1 MB (Sweep), driven by ForkJoinPool's per-thread work queues under 16-way parallelism.

The most extreme memory cost belongs to Julia's library-ecosystem loading: 763.0 MB in ODE and 846.4 MB in Sweep when the full DifferentialEquations.jl dependency graph is loaded. These figures contract to 303.8 MB (SSA) and 285.0 MB (RD) when the solver ecosystem is not required, exposing the library-load tax as the dominant memory component rather than the runtime itself.

These memory differentials translate directly into deployment density [60]. A compute node with 32 GB of RAM can host several thousand native binary processes under fault-isolated multi-process scheduling. The same node accommodates roughly 40–50 heavy managed-runtime instances. Single-process multithreading attenuates the impact of per-instance memory overhead; multi-process isolation, as employed in systems biology calibration workflows scanning thousands of parameter combinations, amplifies it. The deployment-density constraint may be encountered before a single-core throughput limit is reached. For batch-scheduled infrastructure where memory allocation is a fixed per-job budget item, the two-orders-of-magnitude span between native binaries and heavy managed runtimes represents a direct operational cost multiplier.

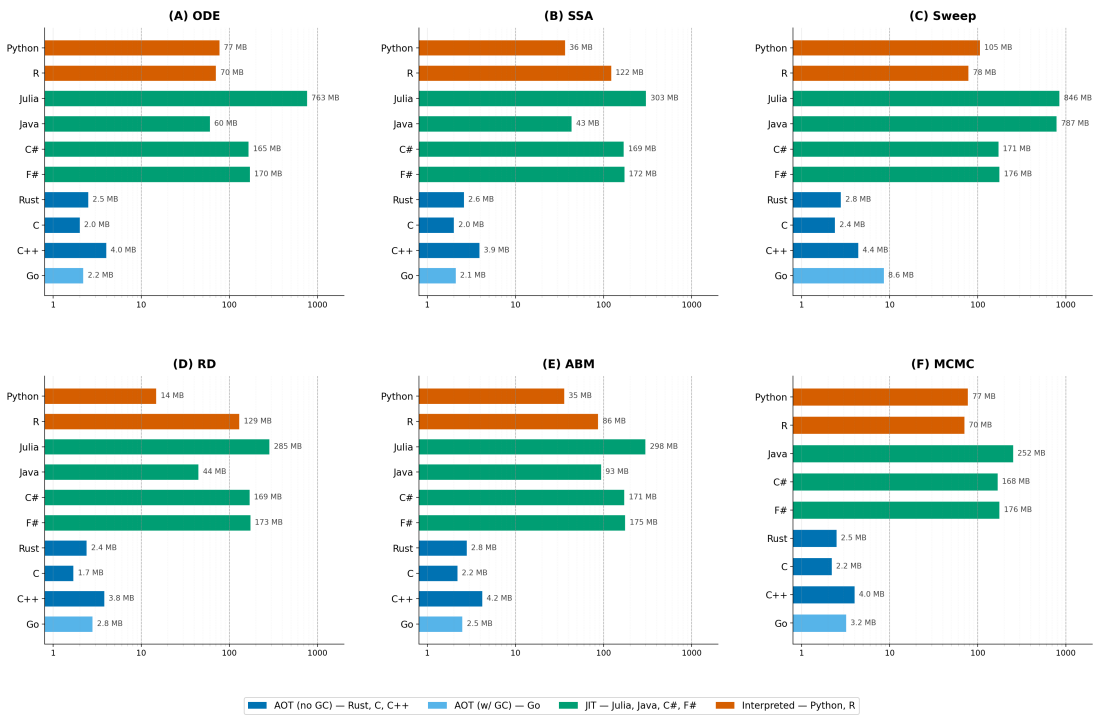


Figure 5. Peak physical memory consumption of ten languages across six computational regimes. A 2×3 panel of horizontal bar charts sharing a log-scaled X-axis (major ticks at 1/10/100/1000 MB). Gray dashed lines mark deployment-relevant thresholds at 10 MB, 100 MB, and 1 GB. The legend is unified by the execution model at the bottom.

5.3. Code Economy

Execution cost and memory cost are runtime observables. Code structure cost is a developer-time observable. The measurement framework captures five dimensions of code economy: cyclomatic complexity (CC), cognitive complexity (CogC), abstraction depth (AL), type-conversion density (TCD), and the memory-management model (MM). Each dimension illuminates a distinct aspect of the implementation tax that a language imposes on the programmer.

Figure 6 employs Gzip-compressed byte counts as a proxy for incompressible semantic content density, with SLOC overlaid as an auxiliary reference. Managed runtimes without JIT produce the smallest compressed footprints: Python spans 700–1740 B, and R spans 700–1460 B. The near-parallel alignment of their SLOC and Gzip polygons indicates a linear mapping between line count and semantic content; a minimal syntactic ceremony means compression algorithms find little redundant structure to eliminate. Native binaries occupy the opposite extreme. Rust's Gzip range of 1085–2540 B is the largest in any subplot, reflecting ownership annotations, error-handling paths, and full Butcher-coefficient transcription. The expansion of the solid Gzip polygon relative to the dashed SLOC polygon reveals syntactic ceremony density: type annotations, access modifiers, and exception declarations contribute incompressible bytes that survive compression.

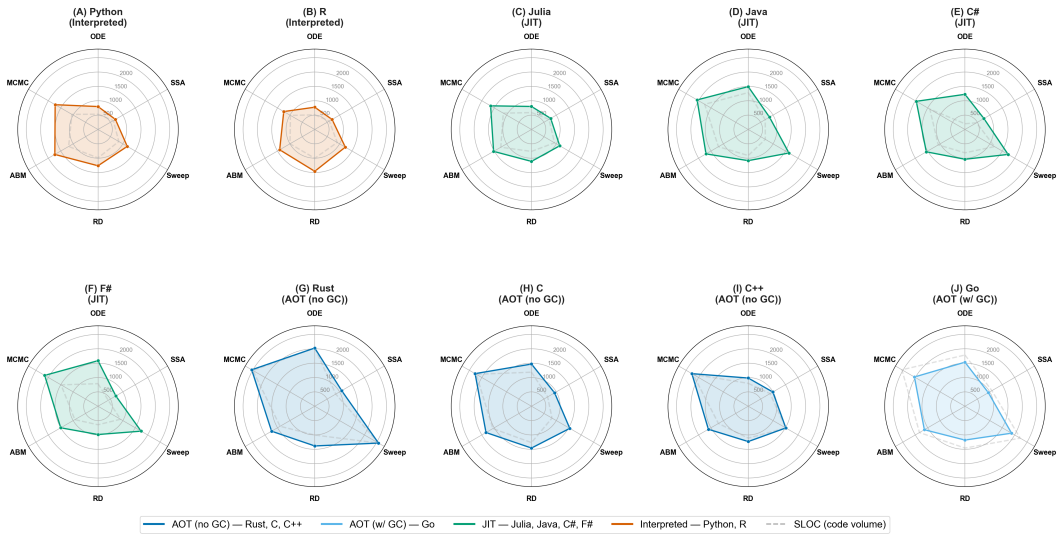


Figure 6. Information density of ten languages across six regimes, measured by Gzip-compressed bytes (colored solid polygons, grouped by execution model). Gray dashed polygons show SLOC as an auxiliary size reference. Languages with minimal syntactic ceremony exhibit the smallest Gzip footprint; those requiring type annotations, error paths, and memory-management code contribute incompressible semantic content. The separation between solid and dashed polygons reveals syntactic ceremony: the solid polygon expanding while the dashed polygon stays near center indicates dense code with high information per line.

Table 5 records code structural characteristics across five dimensions. Two structural forces dominate the CC landscape. The first is algorithmic constraint: every language registers exactly CC = 6 on the Gillespie SSA, forced by the irreducible branching structure of the direct method. Reaction diffusion compresses all implementations into the CC = 11–15 band. The algorithm itself imposes a floor on structural complexity that no language can undercut. The second force is library delegation: languages that offload solver logic to external packages (Python, Julia, R) show user-facing CC values of 1–5 on Sweep and MCMC, while hand-coded integrators expose nested adaptive-step logic, driving the CC to 9–12. This split captures the structural price of implementation transparency: hand-

coded solvers expose Butcher stepping, error estimation, and step-size control as explicit code paths, while library-delegating implementations treat these as opaque infrastructure.

Because the library-versus-hand-coded divide confounds a single CC number, the ODE regime's complexity is decomposed into two layers that carry distinct meaning. The *user-interface layer* comprises the code a practitioner writes directly: the model's right-hand-side function and the driver that invokes the solver. This layer is nearly language-independent—the Repressilator's right-hand side registers CC = 1 in all ten languages because the biological model is a straight-line formula with no branching. The *numerical-kernel layer* comprises the adaptive-step machinery: Butcher stepping, embedded error estimation, and step-size control. In library-delegating languages, this layer is absent from the user's source entirely (Python's `solve_ivp`, Julia's Tsit5, and R's `lsoda` conceal it behind the library interface). In hand-coded languages, it appears as explicit control flow: measured kernel CC ranges from 11 (C#, with `RK54Step` CC = 7 and `SolveODE` CC = 4) through 12 (C, Go, and Java) to 17 (Rust, whose `rk45_step` at CC = 12 and `integrate` at CC = 5 jointly encode the full error-control state machine). Table 5 reports user-interface CC for library-delegating languages and total CC for hand-coded languages; the decomposed values, which separate the two layers on a uniform basis, are tabulated in Supplementary Material S7.

Table 5. Multidimensional code complexity across ten languages and six measurement regimes.

Metric	Python	Julia	Rust	C	C++	C#	F#	Go	Java	R
CC (ODE)	4	4	5	5	5	5	5	5	5	4
CC (SSA)	6	6	6	6	6	6	6	6	6	6
CC (Sweep)	3	4 †	10	11	8	10 †	12 †	10 †	10 †	5 †
CC (RD)	13	15 †	11	11	13 †	13 †	13 †	13 †	13 †	15 †
CC (ABM)	29	16 †	16	29	20 †	20 †	24 †	16 †	20 †	24 †
CC (MCMC)	1	3 †	12	9	12 †	9 †	9 †	9 †	9 †	4 †
AL (ODE)	2	2	5	4	3	4	4	4	4	2
AL (SSA)	2	2	2	2	2	2	2	2	2	2
AL (Sweep)	2	2 †	5	4	3 †	5 †	5 †	5 †	5 †	2 †
AL (RD)	2	2	2	2	2	2	2	2	2	2
AL (ABM)	2	3 †	4	3	3 †	3 †	3 †	3 †	3 †	3 †
AL (MCMC)	2	2 †	5	5	5 †	5 †	5 †	5 †	5 †	2 †
CogC (ODE)	L	L	M	M	M	L	M	M	M	L
CogC (SSA)	L	L	M	M	M	L	H	M	M	L
CogC (ABM)	H	M †	M	H	M †	M †	H †	M †	M †	H †
TCD (SSA)	0	0	6	0	4	0	4	4	4	0
MM	G	G	A	E	E	G	G	G	G	G

CC = cyclomatic complexity; CogC = cognitive complexity (L = low; M = medium; H = high); AL = abstraction layers; TCD = type conversion density; MM = memory management (E = explicit; A = annotated; G = GC-managed). CC values for Python, Rust, and C were verified with `lizard`; values marked † are estimated from structural pattern equivalence (see Methods). Sweep/MCMC CC for library-based languages (Py, Jl, and R; C++ for Sweep) reports user-facing wrapper functions; for hand-coded languages, CC reports solver-level functions (`rk45_step`/`rk54Step`/`solve_ode`). TCD applies only to SSA; purely floating-point regimes (ODE, RD, Sweep, and MCMC) lack equivalent-density integer-to-float conversions. CogC analyzes only the three regimes with maximal cross-language differentiation (ODE, SSA, and ABM).

The cognitive-complexity dimension registers burdens invisible to cyclomatic counts. F#'s empty `then () else` branch, which is idiomatic in the match-expression style, earns a "High" designation on SSA. The functional paradigm's immutable defaults push ABM cognitive complexity to High as well. Rust's ownership annotations and explicit scope delimiters raise cognitive load to Medium across all three analyzed regimes. The abstraction-layer metric reinforces the library-delegation divide in a complementary manner: Python,

Julia, and R encapsulate solvers behind two layers on Sweep and MCMC, while Rust and C expose five layers of explicit Butcher stepping, error estimation, and step-size control.

Type-conversion density captures a distinct dimension of static-typing cost specific to the branch-intensive SSA regime. Rust demands six `as f64` casts per Gillespie iteration; integer molecule counts must be promoted to floating-point values for propensity-function arithmetic. C++ requires four `static_cast<double>`, and Go, Java, and F# each require four implicit or explicit conversions. Python, Julia, and R pay zero because implicit numeric promotion absorbs the conversion [58]. Whether these explicit conversions constitute a documentation asset or a coding tax is a matter of debugging context. An `as f64` annotation flags a precision-sensitive numeric transition that implicit promotion would conceal. Memory management divides into three models: explicit (C's `malloc/free` and C++'s RAII), annotated (Rust's `&mut` borrows), and garbage-collected (all remaining languages).

The code-economy data collectively argue against single-metric language assessment. SLOC carries limited information value as a standalone metric [55,56,61]. Prechelt (2000) demonstrated that the code-volume advantage of managed runtimes over native languages is the primary adoption driver among domain scientists [29]. Multidimensional frameworks provide a more complete information basis for language selection than any single number [62,63]. Nanz and Furia (2015) introduced the program failure rate as a supplementary quality dimension that runtime measurements alone cannot address [30]. Cognitive complexity [64] and TCD reveal dimensions of developer experience that complement execution-time measurements [31].

5.4. Semantic Fit

The cost of data-structure semantics manifests most acutely when a language's default mutation model is in opposition to the algorithm's operational pattern. The Gillespie SSA and the ABM regime consist of repeated, fine-grained state mutations: molecule counts adjusted by ± 1 per event and cells appended per division. The alignment or misalignment between mutation-model defaults and these algorithmic demands generates measurable runtime consequences and auditability implications.

Table 6 catalogs mutation semantics across languages. The spectrum runs from implicit mutability (objects writable by default, encompassing Python, R, Java, C#, C, C++, and Go), through a mixed model (F#, where functional types default to immutability but arrays permit in-place updates [53]) to explicit mutability (Julia's `!` suffix convention and Rust's `&mut` borrow annotations make mutation a deliberate, compiler-verified act).

Table 6. Mutation semantics classification by language.

Language	Mutation Semantics
Python	Implicitly mutable
R	Implicitly mutable
Java	Implicitly mutable
C#	Implicitly mutable
C	Implicitly mutable *
C++	Implicitly mutable
Go	Implicitly mutable
F#	Mixed
Julia	Explicitly mutable
Rust	Explicitly mutable

* C uses file-scope `static` mutable arrays.

Two of the three semantic-alignment dimensions examined in the measurement framework (naming conventions and type-safety enforcement) do not generate independent differentiating signals at the tested model scales. Naming conventions divide between

named-field access (Python, R, Julia, and Rust) and positional array indexing (Java, C#, F#, C, C++, and Go), which is a distinction reflecting implementation convention rather than language capability. At the Repressilator's six state variables, the cognitive difference between `m1` and `y[0]` is negligible. At genome scale, where metabolic models contain hundreds to thousands of ODEs, maintaining mental index-to-quantity mappings becomes a documented source of error during calibration and debugging. Destructured bindings preserve visual correspondence between the mathematical formulation and the source code. Type safety yields a null result in the current measurement: all implementations represent biological quantities as dimensionless floating-point numbers. F# carries language-level dimensional-type support through its Units of Measure feature [65]. In principle, compile-time tags such as `[<Measure>] type concentration` could prevent dimensional errors. In practice, the feature remains disconnected from the scientific library ecosystem; mainstream libraries expose dimensionless floating-point arrays as their default interface [66].

The mutation-semantics dimension, by contrast, produces a demonstrable causal chain with measurable cost consequences. R's immutable vector semantics cause every `vec <- c(vec, new_cell)` call to instantiate a complete copy in the ABM regime, driving execution time to 44.539 s at a cost ratio of 0.14× relative to Python. This outcome documents an architectural mismatch between data-model assumptions and algorithmic requirements. Implicitly mutable languages pay no equivalent penalty for the same append pattern, illustrating that the cost of mutation semantics emerges only when the semantic default and the algorithm's operational pattern are in opposition.

Languages with explicit mutation models render state transitions as intentional design decisions in the source text. Julia's `!` convention and Rust's `&mut` annotations mark each mutation as a deliberate act, aiding in auditability when simulation results deviate from biological expectations. The debugging implication carries disproportionate weight in biological simulation because errors frequently arise at the interface between the mathematical model and its computational implementation [67]. In implicitly mutable code, anomalous state modification in a Gillespie trajectory occurs silently, and tracing it demands line-by-line backtracking across thousands of iterations. Explicit mutation models mark each state transition as deliberate, helping distinguish model predictions from code artifacts during validation. The cost of implicit mutability is not measured in execution cycles but in debugging cycles; the debugging tax compounds across the model development life cycle.

6. Practical Accounting

1. *Regime awareness as cost avoidance.* Language selection without regime awareness imposes an invisible but recurring overhead on every compute cycle. Floating-point dense regimes expose the full price of managed runtime execution: native implementations eliminate the interpretation layer that multiplies per-call overhead across millions of right-hand-side evaluations. Branch-dense regimes, by contrast, erode this price differential. The branch predictor constrains all execution targets equally, and managed runtimes recover competitiveness because arithmetic throughput ceases to dominate the cost ledger. Memory-bandwidth-limited regimes erase the price almost entirely; all implementations converge to a narrow cost band set by the memory subsystem. Practitioners who select languages without first diagnosing the regime type pay a tax whose magnitude varies by orders of magnitude across the computational landscape [55].
2. *Thread scheduling overhead.* Parallel decomposition carries its own cost structure that interacts with regime type and runtime architecture. Work-stealing schedulers in native implementations distribute fine-grained tasks with minimal coordination overhead, while cooperative thread models impose a scheduling tax that grows with thread

count. The measurement data from the parameter-sweep regime reveals that the price of suboptimal thread scheduling can exceed the price of single-threaded managed runtime execution. Pre-allocating memory buffers to eliminate garbage-collection pressure inside hot loops produces a larger marginal cost reduction than adding supplementary hardware threads in several managed runtime environments [16].

3. *Memory as deployment cost.* Peak memory footprint translates directly into containerized deployment density. A factor spanning roughly two orders of magnitude across languages means a single hardware node can host between tens and thousands of concurrent simulation instances depending solely on language choice. For cloud-hosted parameter sweeps, high-throughput screening campaigns, and continuous integration pipelines, this memory price dominates the total operational cost more decisively than single-instance wall-clock time [36]. Allocation-intensive regimes magnify the memory cost further: mutation semantics that force full copies on append operations impose a structural penalty that renders certain language–runtime combinations economically infeasible for large-scale agent-based modeling [66].
4. *Code volume as a development tax.* Implementation size, measured in logical source lines, varies across languages by roughly an order of magnitude for identical mathematical content. This volume differential constitutes a recurring maintenance tax: every line carries a nonzero probability of harboring a defect, and longer implementations distribute defect risk across a larger surface [61]. Algorithmic form places a floor on structural complexity irrespective of language, yet type-strictness annotations and explicit memory-management directives add inflection to per-line cognitive load. The data-structure-to-biological-concept fidelity introduces a further dimension: compact representations that obscure the mapping between code entities and biological state variables impose a debugging tax that compounds with model scale [58,68].
5. *The unbundling horizon.* The dual-language architecture that has defined computational biology for two decades is approaching an inflection point. Deep learning frameworks and JAX/XLA-compiled kernels are progressively dissolving the boundary between native and managed runtime execution [7]. A new generation of languages carries the potential to collapse the prototyping-to-production cost gradient into a single toolchain: Rust, Julia, and F# each approach this goal from a different direction [4,22]. Realizing that potential demands more than compiler quality. It demands the coordinated construction of biological simulation library ecosystems: ODE solver collections, SBML and CellML parsers, reaction-network domain-specific languages, and MCMC engines, built upon native and JIT-compiled foundations, validated against community reference problems, and documented through biological-model-driven tutorials rather than general-purpose programming textbooks [44,45,69]. Infrastructure investment, not language advocacy, will determine whether the next decade of computational biology achieves a single-toolchain future.

7. Limitations and Future Work

The measurement framework is deliberately scoped, and four boundaries of that scope condition the interpretation of every number reported above. Each is stated here, together with the direction of work that would relax it.

7.1. Hardware Diversity

All measurements were collected on a single x86-64 hybrid processor (i7-12650H). The P-core/E-core topology, cache hierarchy, and thread-scheduling behavior of this platform are specific to it, and prior work has shown that relative language ordering can reverse across microarchitectures [59]. The homogeneous-core-subset scaling study introduced

in this revision bounds the P/E-core scheduling confound on the sweep regime, but it does not substitute for genuinely different platforms. Cross-validation on ARM-based processors (Apple Silicon) and high-core-count server architectures (AMD EPYC) would establish whether the tier structure reported here transfers across instruction-set and memory-hierarchy boundaries and is the highest-priority extension.

7.2. Energy Consumption

Energy was not measured. Pereira et al. [32,33] established that native implementations draw a small fraction of the energy of managed-runtime counterparts, but that result was obtained on server hardware and is cited here as context rather than reproduced. An energy dimension is a natural extension for parameter-sweep and ensemble workloads, where aggregate institutional energy cost dominates operational economics; it requires instrumentation (power telemetry or per-component counters) unavailable on the present platform.

7.3. Model Scale and Stiffness

The six regimes employ low-dimensional, analytically tractable reference models (a six-variable Repressilator, a four-channel Gillespie system, and a 128×128 Gray–Scott grid). Stiff dynamics and ill-conditioned Jacobians were deliberately excluded so that adaptive step-size behavior would not confound the language comparison. Two consequences follow. First, the reported cost ratios are lower bounds: larger or stiffer models would amplify the per-step overhead of managed runtimes. Second, the results cannot speak to stiff-solver regimes, where library-quality adaptive step control—rather than the host-language runtime—becomes the dominant cost driver. A genome-scale metabolic model (hundreds to thousands of ODEs) or a stiff reaction system would additionally exercise JIT warm-up behavior and external-memory boundaries that the present models do not reach.

7.4. Replication and Statistical Scope

The statistical claims are restricted to tier-level cost gaps exceeding an order of magnitude, which remain robust to the measured replicate-to-replicate dispersion (Supplementary Material S6). Within-tier rankings are reported as descriptive because their confidence intervals overlap at $n = 10$ replicates. Increasing replication depth and archiving raw replicate arrays uniformly across all six regimes would extend inferential coverage to within-tier comparisons. Processor-state limits (turbo boost and thermal drift under WSL2) contribute a residual, regime-independent noise source that is absorbed by the replicate statistics but not separately resolvable without hardware frequency telemetry.

Supplementary Materials: The following supporting information can be downloaded at: https://www.mdpi.com/article/10.3390/sym18091559/s1. References [70,71] are cited in the Supplementary Materials.

Author Contributions: Conceptualization, Y.Z. and Q.L.; methodology, Y.Z.; software, Y.Z.; validation, Y.Z.; formal analysis, Y.Z.; investigation, Y.Z.; resources, Q.L.; data curation, Y.Z.; writing—original draft preparation, Y.Z.; writing—review and editing, Y.Z. and Q.L.; visualization, Y.Z.; supervision, Q.L.; project administration, Q.L.; funding acquisition, Q.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: Implementation code: https://github.com/Anyee-Lab/benchmark (accessed on 19 July 2026). Raw data will be released upon acceptance (FAIR4RS principles).

Acknowledgments: The authors acknowledge the open-source communities maintaining the compilers, interpreters, and runtimes for all languages measured in this study. Large language models

(DeepSeek-V3.1) were used solely for English-language polishing, grammar correction, and stylistic refinement of the manuscript text. All authors have thoroughly reviewed, edited, and verified the AI-generated suggestions. We take full responsibility for the accuracy, originality, and integrity of the final content of this publication.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

ABM	Agent-Based Model
AOT	Ahead Of Time
CC	Cyclomatic Complexity
CogC	Cognitive Complexity
FP	Floating Point
GC	Garbage Collection
HPC	High-Performance Computing
JIT	Just In Time
MCMC	Markov Chain Monte Carlo
ODE	Ordinary Differential Equation
RD	Reaction–Diffusion
RSS	Resident Set Size
SLOC	Source Lines of Code
SSA	Stochastic Simulation Algorithm
TCD	Type-Conversion Density

References

1. Turing, A.M. The chemical basis of morphogenesis. *Philos. Trans. R. Soc. Lond. Ser. B Biol. Sci.* **1952**, *237*, 37–72. [CrossRef]
2. Bonabeau, E. Agent-based modeling: Methods and techniques for simulating human systems. *Proc. Natl. Acad. Sci. USA* **2002**, *99*, 7280–7287. [CrossRef] [PubMed]
3. Wilkinson, D.J. Stochastic modelling for quantitative description of heterogeneous biological systems. *Nat. Rev. Genet.* **2009**, *10*, 122–133. [CrossRef] [PubMed]
4. Roesch, E.; Greener, J.G.; MacLean, A.L.; Nassar, H.; Rackauckas, C.; Holy, T.E.; Stumpf, M.P.H. Julia for biologists. *Nat. Methods* **2023**, *20*, 655–664. [CrossRef] [PubMed]
5. Kondratyuk, N.; Nikolskiy, V.; Pavlov, D.; Stegailov, V. GPU-accelerated molecular dynamics: State-of-art software performance and porting from NVIDIA CUDA to AMD HIP. *Int. J. High Perform. Comput. Appl.* **2021**, *35*, 312–324. [CrossRef]
6. Karr, J.; Sanghvi, J.; Macklin, D.; Gutschow, M.; Jacobs, J.; Bolival, B.; Assad-Garcia, N.; Glass, J.; Covert, M. A whole-cell computational model predicts phenotype from genotype. *Cell* **2012**, *150*, 389–401. [CrossRef] [PubMed]
7. Abramson, J.; Adler, J.; Dunger, J.; Evans, R.; Green, T.; Pritzel, A.; Ronneberger, O.; Willmore, L.; Ballard, A.J.; Bambrick, J.; et al. Accurate structure prediction of biomolecular interactions with AlphaFold 3. *Nature* **2024**, *630*, 493–500. [CrossRef] [PubMed]
8. Niederer, S.A.; Sacks, M.S.; Girolami, M.; Willcox, K. Scaling digital twins from the artisanal to the industrial. *Nat. Comput. Sci.* **2021**, *1*, 313–320. [CrossRef] [PubMed]
9. Abraham, M.J.; Murtola, T.; Schulz, R.; Páll, S.; Smith, J.C.; Hess, B.; Lindahl, E. GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers. *SoftwareX* **2015**, *1–2*, 19–25. [CrossRef]
10. Eastman, P.; Swails, J.; Chodera, J.D.; McGibbon, R.T.; Zhao, Y.; Beauchamp, K.A.; Wang, L.P.; Simmonett, A.C.; Harrigan, M.P.; Stern, C.D.; et al. OpenMM 7: Rapid development of high performance algorithms for molecular dynamics. *PLoS Comput. Biol.* **2017**, *13*, e1005659. [CrossRef] [PubMed]
11. Hoops, S.; Sahle, S.; Gauges, R.; Lee, C.; Pahle, J.; Simus, N.; Singhal, M.; Xu, L.; Mendes, P.; Kummer, U. COPASI—A COmplex PAthway SImulator. *Bioinformatics* **2006**, *22*, 3067–3074. [CrossRef] [PubMed]
12. Kelleher, J.; Etheridge, A.M.; McVean, G. Efficient coalescent simulation and genealogical analysis for large sample sizes. *PLoS Comput. Biol.* **2016**, *12*, e1004842. [CrossRef] [PubMed]
13. Baumdicker, F.; Bisschop, G.; Goldstein, D.; Gower, G.; Ragsdale, A.P.; Tsambos, G.; Zhu, S.; Eldon, B.; Ellerman, E.C.; Galloway, J.G.; et al. Efficient ancestry and mutation simulation with msprime 1.0. *Genetics* **2022**, *220*, iyab229. [CrossRef] [PubMed]
14. Russell, P.H.; Johnson, R.L.; Ananthan, S.; Harnke, B.; Carlson, N.E. A large-scale analysis of bioinformatics code on GitHub. *PLoS ONE* **2018**, *13*, e0205898. [CrossRef] [PubMed]

15. Huber, W.; Carey, V.J.; Gentleman, R.; Anders, S.; Carlson, M.; Carvalho, B.S.; Bravo, H.C.; Davis, S.; Gatto, L.; Girke, T.; et al. Orchestrating high-throughput genomic analysis with Bioconductor. *Nat. Methods* **2015**, *12*, 115–121. [CrossRef] [PubMed]
16. Yang, H.; Nong, Y.; Wang, S.; Cai, H. Multi-Language Software Development: Issues, Challenges, and Solutions. *IEEE Trans. Softw. Eng.* **2024**, *50*, 512–533. [CrossRef]
17. InMoose. InMoose: Bridging the Gap between R and Python in Bulk Transcriptomic Data Analysis. *Sci. Rep.* **2025**, *15*, 18104. [CrossRef] [PubMed]
18. Bezanson, J.; Edelman, A.; Karpinski, S.; Shah, V.B. Julia: A fresh approach to numerical computing. *SIAM Rev.* **2017**, *59*, 65–98. [CrossRef]
19. Perkel, J.M. Julia: Come for the syntax, stay for the speed. *Nature* **2019**, *572*, 141–142. [CrossRef] [PubMed]
20. Eschle, J.; Gál, T.; Giordano, M.; Gras, P.; Hegner, B.; Heinrich, L.; Hernandez Acosta, U.; Kluth, S.; Ling, J.; Mato, P.; et al. Potential of the Julia programming language for high energy physics computing. *Comput. Softw. Big Sci.* **2023**, *7*, 10. [CrossRef]
21. Persson, S.; Fröhlich, F.; Grein, S.; Loman, T.; Ognissanti, D.; Hasselgren, V.; Hasenauer, J.; Cvijovic, M. PEtab.jl: Advancing the efficiency and utility of dynamic modelling. *Bioinformatics* **2025**, *41*, btaf497. [CrossRef] [PubMed]
22. Köster, J. Rust-Bio: A fast and safe bioinformatics library. *Bioinformatics* **2016**, *32*, 444–446. [CrossRef] [PubMed]
23. Wiewiórka, M.; Khamutou, P.; Zbysiński, M.; Gambin, T. polars-bio—fast, scalable, and out-of-core operations on large genomic interval datasets. *Bioinformatics* **2025**, *41*, btaf640. [CrossRef] [PubMed]
24. Vijendran, S.; Anderson, T.; Markin, A.; Eulenstein, O. Phylo-rs: An extensible phylogenetic analysis library in Rust. *BMC Bioinform.* **2025**, *26*, 197. [CrossRef] [PubMed]
25. Fourment, M.; Gillings, M.R. A comparison of common programming languages used in bioinformatics. *BMC Bioinform.* **2008**, *9*, 82. [CrossRef] [PubMed]
26. Li, H. Biofast: A Small Benchmark for Evaluating the Performance of Programming Languages and Implementations on a Few Common Tasks in the Field of Bioinformatics. 2020. Available online: https://github.com/lh3/biofast (accessed on 19 July 2026).
27. Costanza, P.; Herzeel, C.; Verachtert, W. A comparison of three programming languages for a full-fledged next-generation sequencing tool. *BMC Bioinform.* **2019**, *20*, 301. [CrossRef] [PubMed]
28. Bäckström, J. Performance Evaluation of Programming Languages in Data-Intensive Applications. Bachelor's Thesis, Tampere University, Tampere, Finland, 2025.
29. Prechelt, L. An Empirical Comparison of C, C++, Java, Perl, Python, Rexx, and Tcl. *IEEE Comput.* **2000**, *33*, 23–29. [CrossRef]
30. Nanz, S.; Furia, C.A. A Comparative Study of Programming Languages in Rosetta Code. In *Proceedings of the ICSE 2015*; IEEE: New York, NY, USA, 2015; pp. 778–788. [CrossRef]
31. Bergmans, L.; Schrijen, X.; Ouwehand, E.; Bruntink, M. Measuring Source Code Conciseness Across Programming Languages Using Compression. In *Proceedings of the 2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*; IEEE: New York, NY, USA, 2021; pp. 47–57. [CrossRef]
32. Pereira, R.; Couto, M.; Ribeiro, F.; Rua, R.; Cunha, J.; Fernandes, J.P.; Saraiva, J. Energy efficiency across programming languages: How do energy, time, and memory relate? In *Proceedings of the SLE 2017*; ACM: New York, NY, USA, 2017; pp. 256–267. [CrossRef]
33. Pereira, R.; Couto, M.; Ribeiro, F.; Rua, R.; Cunha, J.; Fernandes, J.P.; Saraiva, J. Ranking programming languages by energy efficiency. *Sci. Comput. Program.* **2021**, *205*, 102609. [CrossRef]
34. SciML. SciMLBenchmarks.jl: Benchmarks for Scientific Machine Learning (SciML) and Equation Solvers. 2016. Available online: https://github.com/SciML/SciMLBenchmarks.jl (accessed on 19 July 2026).
35. Kishore, D.; Chandrasekaran, S. Introducing and Benchmarking the Accuracy of cayenne: A Python Package for Stochastic Simulations. *bioRxiv* **2020**. [CrossRef]
36. Wilson, G.; Aruliah, D.A.; Brown, C.T.; Chue Hong, N.P.; Davis, M.; Guy, R.T.; Haddock, S.H.D.; Huff, K.D.; Mitchell, I.M.; Plumbley, M.D.; et al. Best Practices for Scientific Computing. *PLoS Biol.* **2014**, *12*, e1001745. [CrossRef] [PubMed]
37. Wilson, G.; Bryan, J.; Cranston, K.; Kitzes, J.; Nederbragt, L.; Teal, T.K. Good Enough Practices in Scientific Computing. *PLoS Comput. Biol.* **2017**, *13*, e1005510. [CrossRef] [PubMed]
38. Gmys, J.; Carneiro, T.; Melab, N.; Talbi, E.G.; Tuyttens, D. A comparative study of high-productivity high-performance programming languages for parallel metaheuristics. *Swarm Evol. Comput.* **2020**, *57*, 100720. [CrossRef]
39. Elowitz, M.B.; Leibler, S. A synthetic oscillatory network of transcriptional regulators. *Nature* **2000**, *403*, 335–338. [CrossRef] [PubMed]
40. Elowitz, M.B.; Levine, A.J.; Siggia, E.D.; Swain, P.S. Stochastic Gene Expression in a Single Cell. *Science* **2002**, *297*, 1183–1186. [CrossRef] [PubMed]
41. Gutenkunst, R.N.; Waterfall, J.J.; Casey, F.P.; Brown, K.S.; Myers, C.R.; Sethna, J.P. Universally sloppy parameter sensitivities in systems biology models. *PLoS Comput. Biol.* **2007**, *3*, e189. [CrossRef] [PubMed]
42. Kondo, S.; Miura, T. Reaction-diffusion model as a framework for understanding biological pattern formation. *Science* **2010**, *329*, 1616–1620. [CrossRef] [PubMed]
43. Hanahan, D.; Weinberg, R.A. Hallmarks of cancer: The next generation. *Cell* **2011**, *144*, 646–674. [CrossRef] [PubMed]

44. Hucka, M.; Finney, A.; Sauro, H.M.; Bolouri, H.; Doyle, J.C.; Kitano, H.; Arkin, A.P.; Bornstein, B.J.; Bray, D.; Cornish-Bowden, A.; et al. The Systems Biology Markup Language (SBML): A Medium for Representation and Exchange of Biochemical Network Models. *Bioinformatics* **2003**, *19*, 524–531. [CrossRef] [PubMed]
45. Lloyd, C.M.; Halstead, M.D.B.; Nielsen, P.F. CellML: Its Future, Present and Past. *Prog. Biophys. Mol. Biol.* **2004**, *85*, 433–450. [CrossRef] [PubMed]
46. Smith, L.P.; Bergmann, F.T.; Chandran, D.; Sauro, H.M. Antimony: A Modular Model Definition Language. *Bioinformatics* **2009**, *25*, 2452–2454. [CrossRef] [PubMed]
47. Gillespie, D.T. Exact stochastic simulation of coupled chemical reactions. *J. Phys. Chem.* **1977**, *81*, 2340–2361. [CrossRef]
48. Pearson, J.E. Complex patterns in a simple system. *Science* **1993**, *261*, 189–192. [CrossRef] [PubMed]
49. Hastings, W.K. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika* **1970**, *57*, 97–109. [CrossRef]
50. Hoefler, T.; Belli, R. Scientific Benchmarking of Parallel Computing Systems: Twelve Ways to Tell the Masses When Reporting Performance Results. In *Proceedings of the SC '15*; ACM: New York, NY, USA, 2015; pp. 1–12. [CrossRef]
51. Weber, L.M.; Saelens, W.; Cannoodt, R.; Soneson, C.; Hapfelmeier, A.; Gardner, P.P.; Boulesteix, A.L.; Saeys, Y.; Robinson, M.D. Essential guidelines for computational method benchmarking. *Genome Biol.* **2019**, *20*, 125. [CrossRef] [PubMed]
52. Mangul, S.; Martin, L.S.; Hill, B.L.; Lam, A.K.M.; Distler, M.G.; Zelikovsky, A.; Eskin, E.; Flint, J. Systematic benchmarking of omics computational tools. *Nat. Commun.* **2019**, *10*, 1393. [CrossRef] [PubMed]
53. Microsoft Corporation. F# Language Specification. 2024. Available online: https://fsharp.github.io/fslang-spec/ (accessed on 19 July 2026).
54. Barker, M.; Chue Hong, N.P.; Katz, D.S.; Lamprecht, A.L.; Martinez-Ortiz, C.; Psomopoulos, F.; Harrow, J.; Castro, L.J.; Gruenpeter, M.; Martinez, P.A.; et al. Introducing the FAIR Principles for research software. *Sci. Data* **2022**, *9*, 622. [CrossRef] [PubMed]
55. Fenton, N.E.; Pfleeger, S.L. *Software Metrics: A Rigorous and Practical Approach*, 2nd ed.; PWS Publishing: Boston, MA, USA, 1997.
56. Kitchenham, B.; Mendes, E. Software productivity measurement using multiple size measures. *IEEE Trans. Softw. Eng.* **2004**, *30*, 1023–1035. [CrossRef]
57. McCabe, T.J. A Complexity Measure. *IEEE Trans. Softw. Eng.* **1976**, *SE-2*, 308–320. [CrossRef]
58. Gopstein, D.; Iannacone, J.; Yan, Y.; DeLong, L.; Zhuang, Y.; Yeh, M.K.C.; Cappos, J. Understanding Misunderstandings in Source Code. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*; ACM: New York, NY, USA, 2017; pp. 129–139. [CrossRef]
59. Godoy, W.F.; Valero-Lara, P.; Dettling, T.E.; Trefftz, C.; Jorquera, I.; Sheehy, T.; Miller, R.G.; Gonzalez-Tallada, M.; Vetter, J.S.; Churavy, V. Evaluating performance and portability of high-level programming models: Julia, Python/Numba, and Kokkos on exascale nodes. In *Proceedings of the IEEE IPDPSW 2023*; IEEE: New York, NY, USA, 2023; pp. 373–382. [CrossRef]
60. Nüst, D.; Sochat, V.; Marwick, B.; Eglen, S.J.; Head, T.; Hirst, T.; Evans, B.D. Ten simple rules for writing Dockerfiles for reproducible data science. *PLoS Comput. Biol.* **2020**, *16*, e1008316. [CrossRef] [PubMed]
61. Basili, V.R.; Perricone, B.T. Software Errors and Complexity: An Empirical Investigation. *Commun. ACM* **1984**, *27*, 42–52. [CrossRef]
62. Zaparanuks, D.; Hauswirth, M. The Beauty and the Beast: Separating Design from Algorithm. In *Proceedings of the 25th European Conference on Object-Oriented Programming (ECOOP 2011)*; Lecture Notes in Computer Science; Mezini, M., Ed.; Springer: Berlin/Heidelberg, Germany, 2011; Volume 6813, pp. 27–51. [CrossRef]
63. Briand, L.C.; Morasca, S.; Basili, V.R. Property-Based Software Engineering Measurement. *IEEE Trans. Softw. Eng.* **1996**, *22*, 68–86. [CrossRef]
64. Campbell, G.A. Cognitive Complexity: An Overview and Evaluation. In *Proceedings of the 2018 International Conference on Technical Debt (TechDebt@ICSE)*; ACM: New York, NY, USA, 2018; pp. 57–58. [CrossRef]
65. Kennedy, A.J. Types for Units-of-Measure: Theory and Practice. In *Proceedings of the Central European Functional Programming School (CEFP 2009)*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2010; Volume 6299, pp. 268–305. [CrossRef]
66. Hanenberg, S.; Kleinschmager, S.; Robbes, R.; Tanter, É.; Stefik, A. An Empirical Study on the Impact of Static Typing on Software Maintainability. *Empir. Softw. Eng.* **2014**, *19*, 1335–1382. [CrossRef]
67. Carver, J.C.; Kendall, R.P.; Squires, S.E.; Post, D.E. Software Development Environments for Scientific and Engineering Software: A Series of Case Studies. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*; IEEE: New York, NY, USA, 2007; pp. 550–559. [CrossRef]
68. Arvanitou, E.M.; Ampatzoglou, A.; Chatzigeorgiou, A.; Carver, J.C. Software Engineering Practices for Scientific Software Development: A Systematic Mapping Study. *J. Syst. Softw.* **2021**, *172*, 110848. [CrossRef]
69. Crestana, G.S.; da Silva Batista, U.; Gonçalves Funnicelli, M.I.; Braga, L.G.; Zuvanov, L.; Bizarria, R.; de Sousa, R.M.; Narciso Ferreira, P.H.; Winck, F.V.; Alves Margarido, G.R.; et al. Bridging the Python Training Gap for Bioscientists in Brazil. *bioRxiv* **2024**. [CrossRef]

70. Gibson, M.A.; Bruck, J. Efficient exact stochastic simulation of chemical systems with many species and many channels. *J. Phys. Chem. A* **2000**, *104*, 1876–1889. [CrossRef]
71. Gillespie, D.T. Approximate accelerated stochastic simulation of chemically reacting systems. *J. Chem. Phys.* **2001**, *115*, 1716–1733. [CrossRef]