Selecting a Simulation Runtime for Opinion Dynamics: Evaluating Elixir for Agent-Based Models

Yurii Lytvynenko[1,*,†], Grygoriy Zholtkevych[1,2,†]

[1]V.N. Karazin Kharkiv National University, 4 Svobody sq., Kharkiv, 61022, Ukraine
[2]Ivan Franko National University of Lviv, 1 Universyteska Str., Lviv, 79007, Ukraine

Abstract

We evaluate Elixir as a simulation runtime for opinion dynamics by implementing the same extended pairwise dialogue model in three engines with RNG parity: an Elixir *actor* engine, an Elixir *task* engine, and a Python *multiprocessing* baseline, ensuring identical outputs given the same seed. On a dense all-pairs stress test (N=300, T=100, ∼4.5M dialogues), the *actor* engine was fastest; under a sparse random-matching topology (k=8; 120k dialogues) the *task* engine outperformed *actor* by ≈18%. Both Elixir engines consistently exceeded Python. Ten repeated trials produced tight 95% confidence intervals, confirming stability and efficient multicore utilization on BEAM. We conclude that Elixir is a robust choice for high-concurrency agent-based simulations, with topology guiding engine selection (actor for dense, task for sparse). This runtime decision provides the methodological foundation for our subsequent modeling work.

Keywords

Opinion Dynamics, Agent-Based Simulation, Simulation Runtime Evaluation, Elixir Programming Language, BEAM Virtual Machine, Computational Social Science, Methodological Foundations

1. Introduction

Simulation of opinion dynamics in networked communities is a computationally demanding task. Modern models, such as the extended pairwise dialogue framework [1], represent social interactions as repeated dialogues where agents retain, adopt, or reject opinions based on attributes like resistance and persuasiveness. These models are mathematically rigorous and offer valuable insights into polarization, consensus, and the emergence of alternative viewpoints. However, their computational cost increases rapidly with network size and interaction density.

Selecting an appropriate programming language and runtime environment is therefore critical. The requirements are precise: the simulation must support a large number of lightweight, independent computations, manage concurrency efficiently, and scale across modern multicore processors without significant programmer overhead. While Python is traditionally used in computational social science due to its extensive ecosystem of scientific libraries, its concurrency limitations pose non-trivial challenges for such workloads.

Other languages have also been employed for computational modeling. Julia [2] offers high numerical performance and a growing ecosystem for scientific computing. However, its concurrency model is still evolving and less mature for large-scale distributed workloads. C and C++ deliver raw performance but require significant manual management of parallelism, memory, and fault tolerance, which increases development complexity and reduces flexibility. In contrast, Elixir [3], built on the Erlang virtual machine (BEAM) [4], provides a concurrency-first design. Its lightweight process model, supervision trees, and fault-tolerant architecture were created for large-scale distributed systems and map naturally onto multi-agent simulations where thousands of dialogues coincide.

ProfIT AI'25: 5th International Workshop of IT-professionals on Artificial Intelligence, October 15–17, 2025, Liverpool, UK
*Corresponding author.
†These authors contributed equally.
inbox@yury-lytvynenko.com (Y. Lytvynenko); g.zholtkevych@karazin.ua,grygoriy.zholtkevych@lnu.edu.ua (G. Zholtkevych)
0009-0004-0732-0602 (Y. Lytvynenko); 0000-0002-7515-2143 (G. Zholtkevych)

In this paper, we argue that Elixir offers unique advantages as the core simulation engine for opinion dynamics models. By evaluating the extended dialogue model in the challenging setting of $N = 300$ agents under all pairs mode, we demonstrate that Elixir provides a scalable and robust foundation for simulation backends in computational social science. Importantly, this evaluation is not an end in itself but a methodological step: it was undertaken to identify the most suitable runtime for subsequent research on the dynamics of community opinions, where the extended dialogue model will be developed and applied at scale.

2. Background

2.1. Opinion Dynamics Model

The study of opinion dynamics seeks to understand how individual preferences evolve through repeated interactions within a community. In the extended dialogue model [1], communication is represented as a series of **pairwise dialogues**. Each dialogue is treated as an atomic operation, where two agents exchange views and update their opinions probabilistically.

Unlike classical binary models, where agents either retain or adopt an interlocutor's opinion, the extended framework introduces a third possibility: selecting an alternative state. As a result, each dialogue may lead to one of three outcomes:

1. An agent retains their current opinion.
2. An agent adopts the interlocutor's opinion.
3. An agent rejects both options and transitions to a generalized alternative.

The likelihood of these outcomes is shaped by two independent attributes: **resistance** (the tendency to keep one's own opinion) and **persuasiveness** (the ability to influence others). By decoupling these factors, the model captures more nuanced dynamics than deterministic consensus frameworks. For instance, when both agents exhibit high resistance and high persuasiveness, they may fail to persuade each other and instead converge on an alternative, reflecting real-world behaviors such as withdrawal from mainstream positions.

By explicitly modeling **resistance** and **persuasiveness** as independent attributes, the framework allows for richer dynamics than deterministic consensus models. For example, when both agents are highly resistant and highly persuasive, they may converge not on each other's opinions but on an alternative, reflecting real-world behaviors such as rejection of mainstream narratives.

2.2. Markovian Transition Structure

The dialogue process is formalized as a Markov chain. Each pair of interacting agents constitutes a state, defined by their current opinions. Transition probabilities between states are governed by the agents' resistance (ρ) and persuasiveness (π) values. This leads to a stochastic transition matrix T, where each row sums to one and encodes the likelihood of moving from one dialogue outcome to another.

The Markovian structure ensures that opinion evolution is both mathematically tractable and computationally efficient to implement, while still capturing non-trivial behaviors such as deadlocks, polarization, or convergence to alternatives.

2.3. Computational Requirements

Although conceptually simple, the extended dialogue model, where opinion updates occur through pairwise dialogues, becomes computationally demanding at scale. Suppose a network has N agents and, in each iteration, every agent engages in k dialogues with distinct peers (assuming dialogues are undirected):

$$D(N, k) = \frac{N\,k}{2}$$

Over T iterations, the total number of dialogue computations is $T \cdot D$. Each dialogue triggers a probabilistic state update governed by the model's transition rules, so runtime scales with both N and k, and T.

To illustrate, consider a sizeable community with $N = 100{,}000$ agents where each agent interacts with $k = 20$ distinct peers per iteration (e.g., a dense social platform cohort or an enterprise communication snapshot). Then:

$$D(100{,}000, 20) = \frac{100{,}000 \times 20}{2} = 1{,}000{,}000$$

dialogues **per iteration**. Across $T = 1{,}000$ iterations, that yields

$$T \cdot D = 1{,}000 \times 1{,}000{,}000 = 10^9$$

dialogues — each an independent, lightweight computation whose outcomes must be aggregated into consistent agent states for the next iteration. This profile highlights three practical demands:

- **High volume of independent interactions** (embarrassingly parallel within an iteration);
- **Natural concurrency** (amenable to multi-core and distributed execution);
- **State management pressure** (efficient aggregation and deterministic progression across iterations).

This workload profile requires a runtime capable of handling massive concurrency with minimal overhead while maintaining robustness over extended execution times.

3. Elixir as a Simulation Runtime

Elixir, a functional programming language built on the Erlang virtual machine (BEAM) [3], was designed originally for distributed, fault-tolerant systems. Its lightweight process model, supervision trees, and transparent scalability across multicore and distributed environments directly address the requirements identified in Section 2. These features make Elixir a natural fit for implementing large-scale opinion dynamics simulations in computational social science.

3.1. Concurrency Model

Elixir implements the actor model through lightweight processes that communicate via message passing. These processes are inexpensive to create and schedule, enabling thousands to run simultaneously on a single machine. In the context of opinion dynamics, this allows each agent or dialogue to be represented as an isolated process, mirroring the independence of interactions in the model. The result is a natural mapping between simulation logic and execution.

3.2. Fault Tolerance

Long-running simulations can be disrupted by runtime errors, leading to incomplete or inconsistent results. Elixir addresses this with supervision trees, which monitor groups of processes and automatically restart them in case of failure. This design provides resilience without requiring explicit error-handling logic for every component, ensuring that simulations can run reliably over millions of dialogue steps.

3.3. Distribution

The BEAM runtime also supports transparent distribution, allowing processes to span multiple CPU cores or even multiple machines with minimal configuration changes. Since dialogues are independent, distributing them across cores or nodes is straightforward, enabling simulations to scale horizontally as network size or iteration count increases.

3.4. Comparison to Python

Python remains the default language for computational modeling due to its extensive scientific libraries and user-friendly syntax. However, its runtime is poorly suited for workloads dominated by massive concurrency:

- The Global Interpreter Lock (GIL) prevents true parallel execution of threads on multiple cores. [5]
- Achieving parallelism typically requires additional frameworks such as *multiprocessing* [6], *Dask* [7], or *Ray* [8], which add overhead and complexity.
- Fault tolerance and process supervision must be implemented manually, increasing the risk of brittle long-running simulations.

Elixir integrates concurrency, fault tolerance, and distribution at both the language and runtime level. These properties make it particularly well-suited for large-scale opinion dynamics simulations, where each iteration involves thousands or even millions of independent dialogues across a network. By reducing implementation complexity and minimizing runtime overhead, Elixir offers a compelling alternative to Python for executing the simulation core, particularly in scenarios that require sustained high concurrency and reliable long-running performance.

It is worth noting that recent developments in Python (PEP 703) [9] have introduced a no-GIL build, which allows for true parallelism across threads. While promising, this feature is not yet the default in production environments, lacks broad library support, and was therefore excluded from our evaluation. We restrict our comparison to the widely used mainstream Python runtime, which remains the de facto standard in computational social science.

4. Experimental Comparison of Elixir and Python

Based on the requirements and architectural considerations outlined in the previous sections, we did an empirical comparison of Elixir and Python. The goal of these experiments was not to fine-tune implementations, but rather to observe how the two runtimes handle workloads characterized by large numbers of lightweight, independent interactions. Both languages were used to execute the same extended dialogue model under identical simulation parameters, enabling a direct assessment of concurrency handling, scalability, and robustness.

4.1. Simulation Parameters

- **Interaction pattern**: *all pairs* — every agent interacts with every other agent in each iteration. Although this pattern may not be realistic, it is employed as a worst-case stress test to maximize computational load while keeping the model simple. By adopting the all-pairs setup, we avoid additional complexity related to dynamically selecting subsets of peers and ensure that differences in runtime performance can be attributed directly to the efficiency of the execution environment.
- **Total dialogues per iteration**: $D(N) = \binom{N}{2} = \frac{N(N-1)}{2}$ For $N = 300$, this results in $44{,}850$ dialogues per iteration.
- **Iterations (T)**: 100.
- **Attributes**: Each agent is initialized with random values of resistance ($\rho \in [0, 1]$) and persuasiveness ($\pi \in [0, 1]$) drawn from a uniform distribution.
- **Outcome aggregation**: Dialogue outcomes are accumulated per agent and averaged at the end of each iteration to update preferences.

4.2. Experimental Environment

All experiments were conducted on a dedicated Amazon EC2 instance of type **c8g.2xlarge** [10], equipped with **8 vCPUs** and **16 GiB RAM**, based on the **AWS Graviton3 (ARM64) architecture**. The instance

was located in the **us-east-1 region**. CPU frequency scaling was left at the AWS defaults (no explicit pinning to performance mode).

The operating system was **Ubuntu 24.04 LTS** with kernel version **6.14.0-1011-aws**.

The software stack included:

- **Elixir 1.14.0** (compiled with Erlang/OTP 24), running on **Erlang/OTP 25 [erts-13.2.2.5]** with JIT enabled.
- **Python 3.12.3**, installed via the system package manager (apt).
- No external Python packages were required beyond the standard library.

Both implementations were installed from Ubuntu's package repositories (apt) to ensure a consistent and reproducible setup.

The benchmarks were run on a **dedicated instance** without competing workloads.

4.3. Implementations

To run the experiments, we developed three independent implementations of the extended dialogue model: two in Elixir and one in Python. Although they differ in architecture and concurrency approach, they all adhere to the same specification to guarantee deterministic parity of results. Shared design choices include:

- **All-pairs interaction** at each iteration, with agent indices ordered ($i < j$) to enforce consistency in dialogue roles.
- **Random number generation (RNG) parity** through a shared 64-bit Linear Congruential Generator (LCG) [11], seeded identically and consumed in the same order across engines.
- **Aggregation** by summing contributions per agent and dividing by $N - 1$.
- **Rounding** to three decimals at fixed stages, ensuring identical numeric results across languages.

It was explicitly validated that all three implementations return identical results given the same parameters and seed, confirming that observed differences in performance are due solely to runtime characteristics rather than algorithmic divergence. Validation was performed by cross-checking outputs across engines, including per-agent preference vectors, average preferences, and vote distributions, ensuring complete agreement within rounding precision.

4.3.1. Elixir Task Engine

Uses `Task.async_stream` to parallelize batches of pairwise dialogues. Each iteration generates all pairs, divides them into chunks, processes them concurrently, and then reduces the results to update agents' states.

4.3.2. Elixir Actor Engine

Implements an actor-based design, with one process per agent and a central Coordinator. Each iteration begins with a snapshot of agent states stored in ETS.Agent processes compute contributions against their peers and send results back to the Coordinator, which merges them and updates agent states.

4.3.3. Python Multiprocessing Engine

Mirrors the Elixir Task Engine design, utilizing the multiprocessing module. Pairs are generated with $i < j$, partitioned into chunks, and distributed to worker processes. Results are collected, reduced per agent, and averaged.

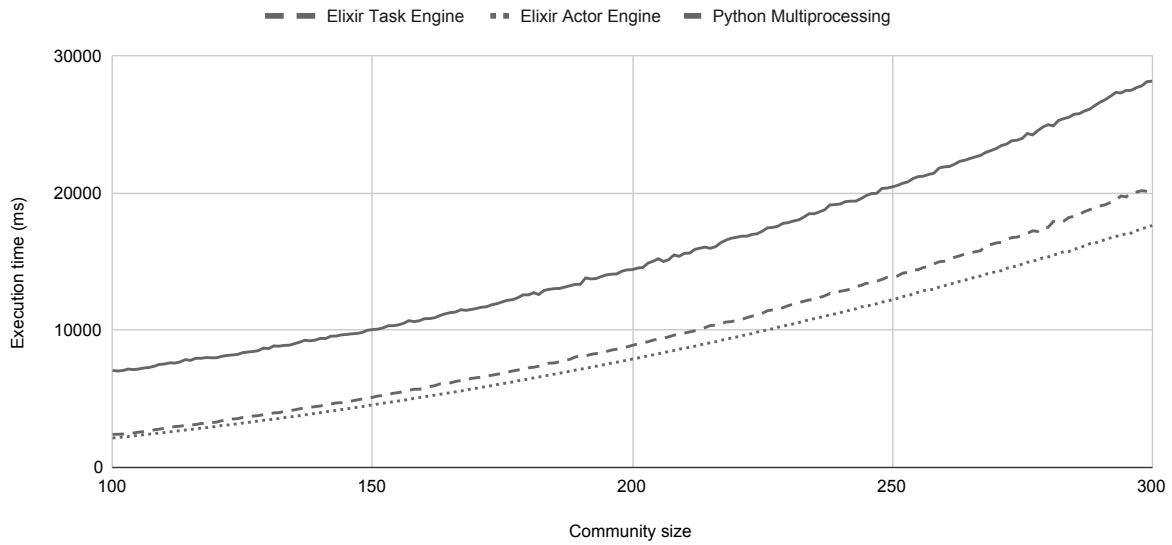

Figure 1: Execution time (ms) for 100 iterations of the all-pairs interaction model as a function of community size N

4.4. Execution Time and Community Size

To assess scalability, we measured execution time as the simulated community size increased, using the *all pairs* interaction pattern for 100 iterations.

We report results starting from $N = 100$. Below this threshold, the number of dialogues per iteration is too small to produce meaningful performance comparisons, as runtimes are dominated by fixed overhead. At $N = 100$, however, each iteration already requires 4,950 dialogues, yielding nearly half a million dialogues across the experiment, which is a substantial computational load.

Execution time grows in line with the quadratic complexity of the all-pairs configuration ($\binom{N}{2}$), but the rate and pattern of growth differ across implementations:

- **Elixir Actor Engine** consistently delivers the best results across all tested community sizes. Despite the coordination overhead of its GenServer design, it handles scaling more efficiently, leading to lower execution times throughout.
- **Elixir Task Engine** performs well but is outpaced by the Actor engine even at small sizes, suggesting that its task scheduling overhead accumulates less favorably.
- **Python multiprocessing** lags significantly behind both Elixir implementations. The gap widens with larger N, reflecting the steep cost of inter-process communication and serialization.

4.4.1. Throughput Analysis

In addition to raw execution time, it is instructive to examine throughput, measured as the number of dialogues processed per second. Throughput was computed as:

$$\text{Throughput} = \frac{\text{Total dialogues}}{\text{Execution time (s)}}$$

where the total dialogues equal the number of iterations (T = 100) multiplied by $\binom{N}{2}$, and execution time was measured in milliseconds and converted to seconds.

Across all tested community sizes, **Elixir Actor Engine** consistently achieves the highest throughput, surpassing both Elixir Task Engine and Python. While Elixir Task Engine performs well, it is outpaced by the Actor engine, indicating that the GenServer-based design not only scales better but also processes

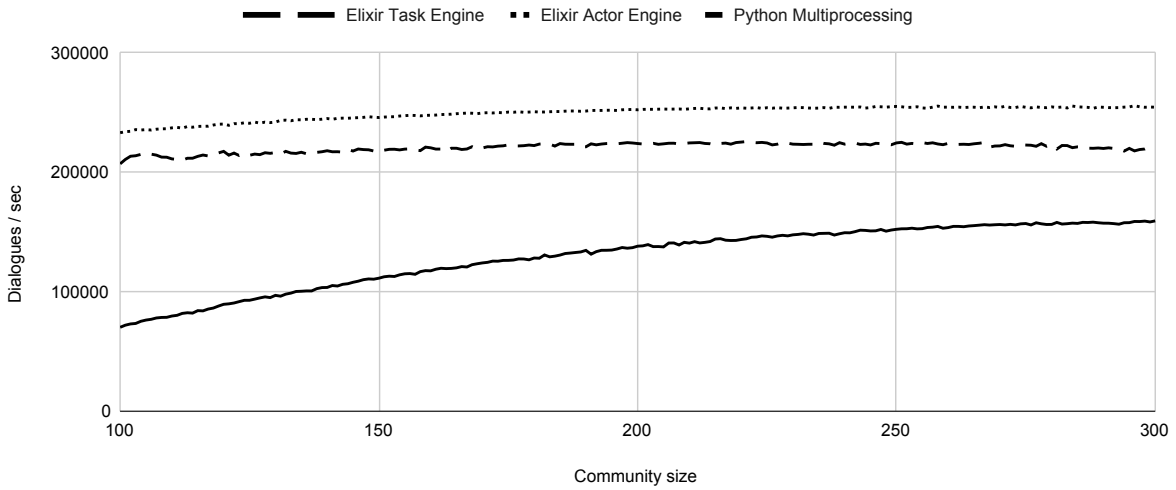

Figure 2: Throughput (dialogues/sec) for 100 iterations under all-pairs interaction, as a function of community size N

dialogues faster in absolute terms. Python multiprocessing lags substantially behind, reflecting the cost of inter-process communication and serialization overhead.

It should be noted that these measurements are based on **single runs per configuration**. While absolute values may vary across repeated executions, the observed ordering of performance and relative scaling trends provide clear evidence of Elixir's advantage as a simulation runtime.

4.5. Repeated Trials at Fixed Parameters

Having established how execution time scales with community size, we next examine performance under fixed parameters by conducting repeated trials. This setting provides a complementary view, focusing on the stability and relative efficiency of the three implementations when the workload is held constant.

4.5.1. Benchmarking Methodology

To ensure reliable and reproducible performance measurements, we developed a systematic benchmarking infrastructure. The methodology consists of three components:

Resource Monitoring. Each simulation run is monitored using the Python `psutil` library [12], which tracks the entire process tree (parent and all child processes). Three metrics are collected:

- **Wall-clock time** (ms): total elapsed time from process start to completion;
- **Peak memory usage** (KB): maximum resident set size (RSS) across the process tree;
- **Average CPU utilization** (%): mean CPU usage measured as system-wide CPU percentage scaled by core count, representing the effective parallelization (e.g., 800% indicates full utilization of 8 cores).

Metrics are sampled at 100ms intervals throughout execution. Memory is tracked per-process and summed across all descendants to capture the full resource footprint of concurrent implementations.

Statistical Analysis. Raw benchmark data from multiple trials are aggregated using a bootstrap method [13] to compute robust confidence intervals. For each metric and configuration, we calculate:

- **Median**: the central tendency, robust to outliers;
- **95% confidence interval**: derived from 10,000 bootstrap samples with replacement, providing upper and lower bounds on the median estimate.

This approach avoids distributional assumptions and provides reliable interval estimates even with small sample sizes (n=10 trials).

Execution Control. Benchmarks are orchestrated by shell scripts that execute each configuration multiple times with identical parameters (agents, iterations, seed, chunk size). Results are recorded in CSV format with one row per trial, enabling reproducibility and subsequent statistical analysis. All benchmarks were run on an isolated AWS EC2 instance (c8g.2xlarge, ARM64) without competing workloads to minimize measurement noise.

4.5.2. All-Pairs Topology Results

The configuration for the all-pairs topology was:

- **Agents:** 300
- **Iterations:** 100
- **Interaction pattern:** all pairs
- **Random seed:** identical across engines and fixed across runs

The choice of 300 agents was deliberate. At this scale, each iteration requires $\binom{300}{2} = 44{,}850$ dialogues, and with 100 iterations, the run produces roughly 4.5 million dialogue computations. This ensures that runtimes are dominated by dialogue processing rather than initialization or warm-up overhead, which plays only a minor role at this load. At the same time, the workload remains tractable on the evaluation hardware (8 vCPUs, 16 GiB RAM) without exhausting memory, and lies within the range commonly used in opinion dynamics studies, where hundreds of agents are sufficient to exhibit realistic aggregate behaviors without entering the territory of very large-scale distributed simulations.

Each implementation was executed **ten times** under these parameters. Table 1 reports the median runtime and 95% confidence intervals for each engine.

Table 1
Performance metrics across ten repeated trials with 300 agents and 100 iterations. Values are reported as median with 95% confidence intervals.

Metric	Elixir Task	Elixir Actor	Python Multi
Walltime (ms)	21,520	18,895	29,521
95% CI	[21,450, 21,567]	[18,828, 18,941]	[29,429, 29,614]
Memory (KB)	290,948	225,986	184,104
95% CI	[289,206, 298,916]	[217,496, 229,036]	[183,934, 184,456]
CPU (%)	629.3	731.9	593.0
95% CI	[627.4, 630.6]	[730.2, 733.7]	[592.2, 594.6]
Throughput (dialogues/s)	$\approx 208k$	$\approx 237k$	$\approx 152k$

The repeated trials confirm several points:

- **Elixir Actor Engine** is consistently the most efficient, completing runs in ~18.9 seconds on average, with the highest CPU utilization (731.9%) indicating effective use of all 8 cores.
- **Elixir Task Engine** averages ~21.5 seconds, about 14% slower than Actor but still faster than Python, with lower CPU utilization (629.3%) suggesting less efficient parallelization.
- **Python multiprocessing** requires ~29.5 seconds, 56% more than Elixir Actor Engine, with CPU utilization (593.0%).
- **Stability:** All implementations show very small confidence intervals (CI widths under 1% of median for walltime), indicating that runtimes are stable and reproducible.
- **Memory usage:** Elixir implementations consume more memory than Python, with Task Engine using the most (291 MB) due to concurrent task overhead, while Actor Engine (226 MB) is more efficient. Python's lower memory footprint (184 MB) reflects sequential processing within workers.

These results reinforce the conclusion that Elixir offers clear performance advantages for highly concurrent workloads. Moreover, the stability across trials demonstrates that the observed differences are not due to random noise but reflect genuine runtime characteristics.

4.5.3. Random Matching Topology

To assess how interaction topology affects performance, we repeated the experiment using a **random matching** pattern instead of all-pairs. Under random matching, each agent is paired with $k = 8$ randomly selected distinct peers per iteration, reducing the number of dialogues from $\binom{N}{2}$ to $\frac{N \cdot k}{2}$.

With 300 agents, $k = 8$ peers, and 100 iterations, random matching produces 120,000 total dialogues — a 97.3% reduction compared to the 4.5 million dialogues in the all-pairs configuration. This sparser interaction pattern more closely resembles realistic social network dynamics where agents engage in limited interactions per time step.

All other experimental parameters remained identical:

- **Agents:** 300
- **Iterations:** 100
- **Random seed:** identical across engines and fixed across runs
- **Trials:** 10 repeated runs per engine

Table 2 presents the performance metrics for random matching topology.

Table 2
Performance metrics across ten repeated trials with 300 agents, 100 iterations, and random matching topology. Values are reported as median with 95% confidence intervals.

Metric	Elixir Task	Elixir Actor	Python Multi
Walltime (ms)	2,278	2,693	6,900
95% CI	[2,276, 2,327]	[2,691, 2,694]	[6,887, 6,946]
Memory (KB)	149,316	135,402	155,318
95% CI	[145,388, 155,492]	[134,004, 138,166]	[155,280, 155,424]
CPU (%)	357.5	432.0	660.5
95% CI	[355.0, 360.8]	[428.6, 435.7]	[655.2, 661.2]
Throughput (dialogues/s)	$\approx 52.7k$	$\approx 44.6k$	$\approx 17.4k$

The results reveal several noteworthy patterns:

- **Performance reversal:** Unlike the all-pairs configuration, **Elixir Task Engine** (2.3s) now outperforms **Elixir Actor Engine** (2.7s) by approximately 18%. This suggests that the Actor engine's coordination overhead becomes proportionally more expensive when dialogue volume is low, while Task's simpler scheduling model is more efficient for sparse workloads.
- **Python still lags: Python multiprocessing** requires 6.9 seconds, still 3× slower than Elixir Task and 2.6× slower than Elixir Actor, though the absolute gap narrows due to reduced computational load.
- **CPU utilization varies by topology:** Both Elixir implementations show lower CPU usage (357% and 432%) compared to all-pairs, reflecting the reduced parallelism available when only 1,200 dialogues occur per iteration (vs. 44,850 in all-pairs). Notably, Python's CPU usage is substantially higher in this sparse topology (660%) than in all-pairs (593%), suggesting different parallelization behavior at varying workload scales.
- **Memory efficiency:** Memory consumption decreased across all engines, with Elixir Actor using the least (135 MB), followed by Task (149 MB) and Python (155 MB). The reduced dialogue volume eliminates the memory pressure observed in all-pairs.

- **Stability varies:** The Actor engine exhibits exceptional precision (CI width of only 3ms), while Task and Python show slightly wider but still tight intervals (51ms and 59ms respectively), all demonstrating consistent performance across trials.

These findings highlight that **topology matters** for runtime selection. While the Actor engine excels at dense all-pairs interactions where massive concurrency is available, the Task engine becomes more efficient for sparse topologies where coordination overhead dominates. Python remains consistently slower regardless of topology, though the performance gap narrows as computational intensity decreases.

5. Conclusions

This work examined the suitability of Elixir as a runtime environment for large-scale simulations of opinion dynamics. Using the extended dialogue model under two distinct interaction topologies — dense all-pairs and sparse random matching — we compared three implementations: an actor-based Elixir engine, a task-based Elixir engine, and a Python multiprocessing baseline. The purpose of this evaluation was not to establish a general benchmarking framework, but to support the authors' ongoing research on community opinion dynamics by identifying a runtime environment that balances correctness, reproducibility, and performance.

The experiments demonstrated that Elixir provides clear advantages for highly concurrent workloads, though the optimal implementation varies with topology. For dense all-pairs interactions (4.5 million dialogues), the Actor engine excelled with superior parallelization and throughput. For sparse random matching (120,000 dialogues with $k = 8$ peers per agent), the Task engine proved more efficient, as coordination overhead became proportionally more significant than raw dialogue processing. Python multiprocessing consistently lagged behind both Elixir implementations regardless of topology, showing 56–200% longer runtimes and varying CPU utilization patterns (593% for all-pairs, 660% for random matching vs. 629–732% for Elixir). The results confirm that the concurrency-first design of the BEAM runtime offers tangible benefits for agent-based simulations that require millions of independent, lightweight interactions to be executed and aggregated deterministically.

At the same time, several limitations must be acknowledged.

1. The current implementations were evaluated on a single 8-core instance, and while they scaled well up to a few thousand agents, attempts to simulate **10,000 agents failed due to memory limitations**. This highlights the need for further optimization and, potentially, distributed execution to extend the approach to larger communities.
2. While we evaluated two topologies (all-pairs and random matching with $k = 8$), exploring additional realistic network structures such as scale-free networks, small-world topologies, or dynamic community detection remains an important direction for future work.
3. Emerging Python runtimes with no-GIL support (PEP 703) may alter the performance landscape. A systematic evaluation of these builds is left for future work, as they are not yet mainstream and their ecosystem support remains limited.

5.1. Future Research Directions

Having established Elixir as a suitable choice, the next steps of the work will focus not on further runtime benchmarking, but on advancing the modeling itself: exploring richer agent attributes, network topologies, and empirical calibration. The following directions, therefore, outline the substantive research agenda beyond the technical runtime decision.

- **Overcoming the 10k-agent barrier:** optimize memory usage and explore distributed execution strategies across multiple nodes.
- **Alternative interaction topologies:** extend experiments to include scale-free networks, small-world structures, and community-based topologies with varying density patterns.

- **Distributed scalability:** test Elixir's native distribution capabilities in larger clusters to evaluate horizontal scaling.
- **Dynamic agent attributes:** allow resistance and persuasiveness to evolve with interaction history, making simulations more realistic.
- **Empirical calibration:** integrate data from surveys or social media to validate and tune model parameters.
- **Hybrid workflow:** employ Elixir for running large-scale simulations while using Python's mature ecosystem for post-simulation data analysis and visualization.

Declaration on Generative AI

During the preparation of this work, the authors used ChatGPT and Grammarly to: Grammar and spelling check, Paraphrase and reword. After using this tool/service, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] Y. Lytvynenko, G. Zholtkevych, Simulating Pairwise Communication to Study Opinion Dynamics in Networked Communities, 2025. doi:10.13140/RG.2.2.29044.82566.

[2] JuliaLang Community, The Julia Programming Language, 2025. URL: https://julialang.org/, accessed: 2025-08-20.

[3] Elixir Core Team, The Elixir Programming Language, 2025. URL: https://elixir-lang.org/, accessed: 2025-08-20.

[4] Erlang/OTP Team, The Erlang Programming Language and OTP, 2025. URL: https://www.erlang.org/, accessed: 2025-08-20.

[5] Python Software Foundation, threading — thread-based parallelism, https://docs.python.org/3/library/threading.html, 2025. Accessed: 2025-08-21.

[6] Multiprocessing — Process-based parallelism, 2025. URL: https://docs.python.org/3/library/multiprocessing.html, accessed: 2025-08-20.

[7] Dask | Scale the Python tools you love, 2025. URL: https://www.dask.org/, accessed: 2025-08-20.

[8] Scale Machine Learning & AI Computing | Ray by Anyscale, 2025. URL: https://ray.io, accessed: 2025-08-20.

[9] PEP 703 – Making the Global Interpreter Lock Optional in CPython | peps.python.org, https://peps.python.org/pep-0703/, 2025. Accessed: 2025-08-25.

[10] Cloud Compute Instances – Amazon EC2 Instance Types – AWS, 2025. URL: https://aws.amazon.com/ec2/instance-types/, accessed: 2025-08-25.

[11] W. E. Thomson, A Modified Congruence Method of Generating Pseudo-random Numbers, The Computer Journal 1 (1958) 83. doi:10.1093/comjnl/1.2.83.

[12] G. Rodola, psutil: Cross-platform lib for process and system monitoring in Python, 2025. URL: https://github.com/giampaolo/psutil, accessed: 2025-10-09.

[13] B. Efron, Bootstrap Methods: Another Look at the Jackknife, The Annals of Statistics 7 (1979) 1–26. doi:10.1214/aos/1176344552.

A. Online Resources

The source code for the simulation is available https://github.com/yurylyt/netcomm_elixir_vs_python on GitHub.