arXiv:2505.12006v4 [cs.AI] 11 Nov 2025

SOCIA-∇: Textual Gradient Meets Multi-Agent Orchestration for Automated Simulator Generation

Yuncheng Hua, **Sion Weatherhead**, **Mehdi Jafari**, **Hao Xue**, **Flora D. Salim**†
School of Computer Science Engineering, University of New South Wales, Australia
{devin.hua, s.weatherhead, mahdi.jafari, hao.xue1, flora.salim}@unsw.edu.au

ABSTRACT

In this paper, we present SOCIA-∇, an end-to-end, agentic framework that treats simulator construction as *instance optimization over code* within a textual computation graph. Specialized LLM-driven agents are embedded as graph nodes, and a workflow manager executes a loss-driven loop: code synthesis → execution → evaluation → code repair. The optimizer performs Textual-Gradient Descent (TGD), while human-in-the-loop interaction is reserved for task-spec confirmation, minimizing expert effort and keeping the code itself as the trainable object. Across three CPS tasks—User Modeling, Mask Adoption, and Personal Mobility—SOCIA-∇ attains state-of-the-art overall accuracy. By unifying multi-agent orchestration with a loss-aligned optimization view, SOCIA-∇ converts brittle prompt pipelines into reproducible, constraint-aware simulator code generation that scales across domains and simulation granularities. We will release the code soon.

KEYWORDS

Textual Gradients, Multi-agent Orchestration, Simulator Code Generation, End-to-end Agentic Framework, Data Calibration

ACM Reference Format:
Yuncheng Hua, **Sion Weatherhead**, **Mehdi Jafari**, **Hao Xue**, **Flora D. Salim**[2], School of Computer Science Engineering, University of New South Wales, Australia, {devin.hua, s.weatherhead, mahdi.jafari, hao.xue1, flora.salim}@unsw.edu.au, . 2026. SOCIA-∇: Textual Gradient Meets Multi-Agent Orchestration for Automated Simulator Generation. In *Proc. of the 25th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2026), Paphos, Cyprus, May 25 – 29, 2026*, IFAAMAS, 11 pages.

1 INTRODUCTION

Building simulators is a research hotspot because they offer a low-cost, reproducible in-silico sandbox for controlled intervention and "what-if" analysis, while enabling researchers to probe emergent outcomes under shifts and unseen regimes [4, 9, 13]. By design, simulators span two paradigms: *policy-oriented aggregate (system-dynamics / compartmental) models*, which summarize populations through equations over aggregated state variables, and *agent-based / microsimulation models*, which follow heterogeneous decision units and their interactions to capture fine-grained dynamics [22, 38, 42, 64]. These approaches arise across cyber, physical, and social (CPS) domains [29, 40, 47, 55, 56], making controlled, repeatable cross-domain experimentation feasible.

Regardless of form, simulators share a universal medium: **executable code**. In practice, a simulator is only useful when its rules, interfaces, and data flows are embodied in code that can be run, evaluated, and calibrated [18, 30, 32]. This observation motivates a pragmatic goal: if we could automate the construction of high-fidelity, extrapolatable simulator code with minimal expert effort, we would substantially lower the cost of building CPS simulators for computer scientists, sociologists, and domain practitioners.

Existing pathways only partially meet this bar. *Manual, expert-built simulators* [25, 28, 49, 62, 74, 75, 80] are accurate but expensive to produce and maintain. *Description-driven systems*—for example, YuLan-OneSim [66], which converts natural-language scenario descriptions into code using ODD protocol [17] and behavior graphs—reduce coding burden but do not, by themselves, provide an inner loop for *data calibration* of the generated programs. At the other end, recent *automatic frameworks* such as G-Sim [20] pair LLM-proposed designs with simulation-based inference, showing promising alignment to observed data; however, these efforts have largely focused on aggregate models and may not readily extend to highly heterogeneous, micro-level simulators.

In parallel, *multi-agent code generation* has advanced rapidly: CAMEL [33], ChatDev [52], AgentCoder [23], AutoGen [19], and AI Scientist-v2 [72] stage specialized LLM agents to propose, test, and refine software or to conduct scientific experiments. These frameworks demonstrate compelling task decomposition and collaboration patterns, but they are typically *prompt-sensitive, hand-tuned, and not designed for simulator construction or data calibration*—their agent protocols and success criteria are not embedded in a loss-compiled optimization view, and they seldom close the loop from execution metrics back to targeted code repair.

To overcome these challenges, we introduce **SOCIA-∇** (Simulation Orchestration for Computational Intelligence with Agents), an end-to-end multi-agent framework that treats simulator building as *instance optimization over code* within a **textual computation graph**. SOCIA-∇ defines specialized agents for data analysis, code generation, execution, evaluation, and feedback, and *embeds each agent as a node* in a directed textual computation graph that carries both textual artifacts (code, logs, critiques) and numeric signals (metrics). A centralized workflow manager orchestrates *forward execution* (generate code → run simulator → compute validation loss and constraint checks) and *backward repair* (route *textual gradients*—loss-aligned, natural-language critiques—toward the responsible upstream code components). The optimizer uses *momentum* (history-aware aggregation of critiques) and *projected gradient descent (PGD)–style projection* (constraint-aware repair to preserve compilability and schema conformance) to stabilize long-horizon

†Corresponding author.

Proc. of the 25th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2026), C. Amato, L. Dennis, V. Mascardi, J. Thangarajah (eds.), May 25 – 29, 2026, Paphos, Cyprus. © 2026 International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org). This work is licenced under the Creative Commons Attribution 4.0 International (CC-BY 4.0) licence.

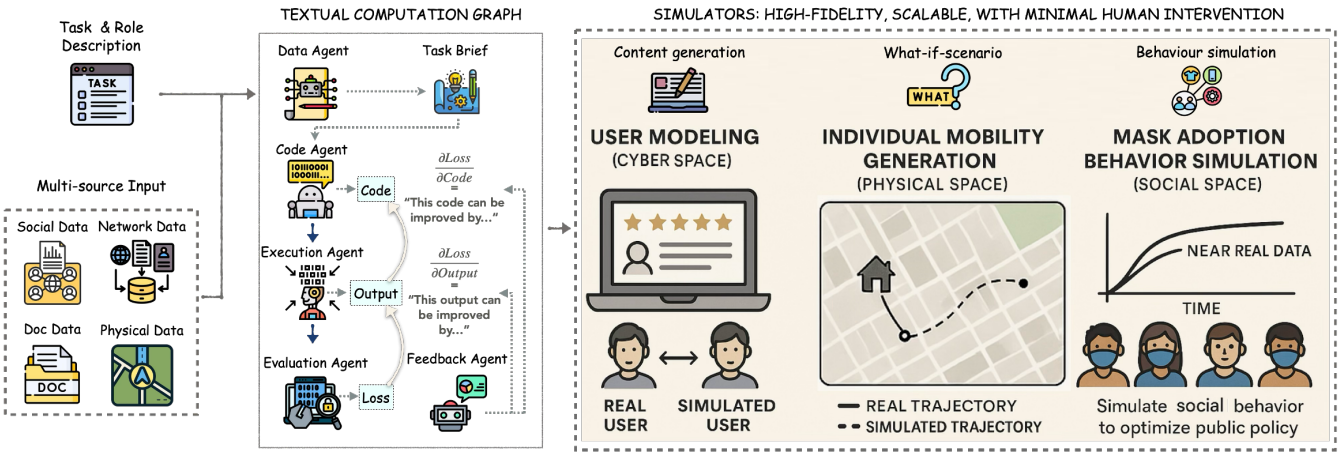

Figure 1: Overview of SOCIA-∇. A multi-agent textual computation graph takes task brief and input data, synthesizes simulator code, executes it, and evaluates loss; the loss is converted into textual gradients that drive code optimization. Right: three representative applications illustrating alignment of simulated outputs with real observables.

edits. Human-in-the-loop (HITL) is employed for task-spec confirmation, but the system aims to minimize expert effort by making the *code itself the trainable object* and iterating until convergence.

In summary, this paper makes three main contributions: (1) We present SOCIA-∇, a reliable, end-to-end simulator-code construction framework that requires minimal human supervision, coordinating heterogeneous LLM agents to synthesize, test, and refine executable simulators. (2) We propose a *textual computation graph* that embeds multi-agent reasoning and treats *simulator code* as the *optimization variable*, enabling *loss-driven, constraint-aware textual-gradient updates* with momentum and projection. (3) We demonstrate empirically that SOCIA-∇ constructs *accurate, extrapolatable simulators* across CPS domains and across both aggregate and agent-based modeling granularities.

2 RELATED WORK

Multi-agent systems (MAS) in Software Engineering. MAS are widely used for code and software generation [19, 23, 26, 33]. Frameworks like MetaGPT [21] and ChatDev [53] formalize workflows via structured roles, while CodeR [10], RGD [27], UnitTest [3], and AI Scientist-v2 [72] target specific phases (issue resolution, testing, or experimentation). Unlike these often predefined and human-supervised pipelines, SOCIA-∇ autonomously synthesizes, executes, and refines *executable simulators* from task descriptions and data.

Simulator construction. Prior work spans: (1) **Domain-crafted simulators** which are authored by expert, creating an engineering bottleneck [25, 28, 32, 49, 62, 74, 75, 80]; (2) **LLM-based multi-agent systems** that compose simulators from natural-language scenarios but lack a closed, loss-aligned inner loop to edit code [11, 15, 19, 54, 59, 66, 78]; (3) **Automated, data-calibrated pipelines** (e.g., G-Sim) that align models to observations yet emphasize aggregate calibration over localized code repair [20]. SOCIA-∇ embeds specialized agents as nodes in a textual computation graph, enabling instance-level, constraint-aware code optimization.

Textual gradients & iterative self-improvement. DSPy [31] formalizes LM pipelines as text-transformation graphs and optimizes them algorithmically; TextGrad [79] improves upstream components in compound systems by *backpropagating textual feedback*; and Self-Refine [39] and Reflexion [60] demonstrate that iterative critiques and episodic memory outperform one-shot prompting across tasks. Unlike these approaches, SOCIA-∇ delivers *loss-aligned, component-targeted* updates together with a *projector* that enforces hard constraints—turning reflection into *reproducible optimization*.

3 SOCIA-∇: APPROACH

As shown in Figure 1, the SOCIA-∇ framework adopts a multi-agent architecture tailored to constructing high-fidelity simulators across heterogeneous CPS domains. Given a task description and data as inputs, SOCIA-∇ outputs *runnable, high-quality simulator code*; once executed by the user, this code launches a high-fidelity simulator. In this chapter, §3.1 elaborates the notions of **computation graphs** and **textual gradients**; §3.2 describes the roles and capabilities of different **agents** in SOCIA-∇; and §3.3 explains how these agent functions are embedded into the computation graph and how **textual-gradient computation and backpropagation** are used to optimize the simulator code.

3.1 Computation Graphs and Textual Gradients

In mainstream ML, a computation graph is a directed acyclic graph (DAG) whose nodes are operations and whose edges carry intermediate values [1, 8, 24, 48]; the forward pass evaluates the graph, and the backward pass applies the chain rule to accumulate gradients for parameter updates [5].

Nowadays, many researchers have begun *building compound AI systems with LLMs*, abstracting complex AI pipelines as *textual computation graphs* and using *differentiation and gradients as a metaphor for LLM-produced textual feedback* [51, 60, 79]. This leverages LLMs' strengths in *reasoning, self-checking, and self-refinement* [7, 31, 34, 39, 60, 77] to optimize AI systems.

In SOCIA-∇, we lift this idea from numeric operators to an *agentic pipeline* that builds a high-fidelity simulator. Each agent that performs a specific function or behavior is a **node** in the DAG; **edges** pass text or numeric artifacts (code, logs, metrics). Many nodes are *non-differentiable* (e.g., LLM calls, compilers, simulators). Following TextGrad [79], we endow this computation graph with **textual gradients**—natural-language critiques that describe how to modify upstream textual variables (here, the simulator code) to improve the downstream objective (high-fidelity simulation). The framework explicitly mirrors the *autograd* metaphor but with *text as the gradient carrier*, and provides a *PyTorch-like paradigm* [48] to define losses and update variables.

As a textual-gradient paradigm with a prompt-LLM backbone, we must ask: **why textual gradients over plain prompt engineering?** Prompt engineering is model/version-dependent, phrasing-sensitive, and hard to evaluate systematically [57], yielding brittleness, poor portability, and dependence on measurement protocols [50]. *Textual gradients* recast the problem as *principled optimization*—a *loss-driven, backprop-style loop* over the whole system that produces targeted, evidence-grounded updates and improves reproducibility and comparability [31]. Rather than global prompt fiddling, textual gradients enable *local, loss-aligned edits*: downstream signals (metric diffs, failing cases, constraint violations) are turned into *machine-readable, executable feedback* and routed to the responsible node (e.g., a faulty function), after which an optimizer LLM proposes *small, verifiable* patches that preserve working components. Moreover, *iterative, feedback-driven refinement* consistently outperforms one-shot prompting [39, 60], reinforcing the superiority of *loss-aligned local edits* over *global prompt tweaks. In sum*, textual gradients transform brittle, template-tuned systems into *reliable, loss-aligned optimization pipelines* with *targeted, verifiable updates* that scale across agents and tasks—precisely what plain prompt engineering struggles to offer.

3.2 Orchestrated Agents

Workflow Manager. Serving as the hub, the Workflow Manager controls the multi-agent workflow. Following the computation-graph design, it activates different agents, loads the task-specific prompts that instruct each agent, executes the graph's forward/backward computations in order, and terminates the iteration according to the convergence status of code optimization.

Code Generation Agent (CGA). CGA supports multiple prompts that each is aligned with a distinct function. It ships with three working prompts:

- *Initial code synthesis.* Before gradient computation begins, CGA reads the task brief and the predefined role description (see §3.3) and, using the `code-generation` prompt, produces the initial simulator code.
- *Code optimization.* During textual-gradient–based optimization, CGA loads the `code-optimization` prompt and upgrades the code according to the back-propagated textual gradients (i.e., critiques on how to improve the code).
- *Constraint repair.* CGA loads the `code-fix` prompt to handle *hard constraints*—coding conventions, interface contracts, and I/O requirements for the simulator (see §3.3).

Additionally, CGA implements a **self-loop** mechanism: after generating/optimizing code, it performs automatic quality checks (e.g., compilation, syntax/static analyses for potential runtime errors, empty-function checks), aggregates all detected issues, and applies improvements. This *self-check* → *fix* cycle runs up to three times per iteration to enhance robustness.

Simulation Execution Agent (SEA). The SEA compiles and executes the simulator code, generates simulation outputs (e.g., agent trajectories or actions), and records any compilation or runtime errors encountered during execution.

Result Evaluation Agent (REA). Given SEA's predictions and error logs, the REA computes fidelity losses using task-specific evaluation metrics and performs code diagnostics based on the compilation and runtime errors reported by SEA. The REA incorporates *constraint satisfaction* into the overall loss computation.

Feedback Generation Agent (FGA). FGA provides two dedicated working prompts. When *REA*'s computed losses and diagnostic reports are available, the FGA loads the `output-criticism` prompt to analyze the comparison between the simulator's outputs and the ground truth for the current iteration—identifying where the output falls short, whether predictions exhibit distortion, and under what conditions such distortion arises. In parallel, the FGA maintains a **historic log** that records past errors and their corresponding fixes. Conditioned on the current simulation code and the output criticisms, the FGA loads the `gradient-generation` prompt and performs both prediction attribution analysis (highlighting failure modes or undesirable behaviors) and code-correction analysis, proposing *executable, code-improving patches* to remedy detected issues. It also summarizes recent errors and fixes from the history window and integrates them with the current round's code-edit proposals before delivering them to the CGA.

Data Analysis Agent (DAA). DAA first consolidates the task description, the task dataset, and task inputs (e.g., agent-related data such as profiles, historical trajectories, and community context). Using a `data-schema-analysis` prompt, it infers the **data schema**: attribute analyses, linkage and interaction relationships across data sources, and the connections between these data and simulator construction (e.g., which data build agent profiles vs. which define the environment). Next, DAA loads a `task-brief-analysis` prompt and, guided by a chain-of-thought (**CoT**) procedure [68], iteratively reasons through and answers the following to produce the simulator's **task brief**: (1) clarify target phenomena/outcomes and modeling stance; (2) specify time step, spatial resolution, and population size; (3) define agent unit(s) and roles; list static attributes and dynamic states of agents; (4) describe agent interactions; (5) determine whether/how information diffuses; (6) identify external signals/interventions and their access paths to agents/modules; (7) define actions and the policy mapping observations/signals to actions; (8) define task-relevant metrics.

Importantly, *after* DAA drafts the task brief, we introduce a **human-in-the-loop (HITL)** stage [44, 45, 61, 69]: domain experts review and provide feedback to confirm the simulator design. DAA incorporates human feedback, produces a revised task brief, and repeats expert confirmation until approval. In practice, leaving all design choices solely to an LLM can lead to *hallucinations* [36, 70];

due to CoT, early hallucinations may *propagate* through the reasoning process, causing substantial *semantic drift* between the LLM's final design and user intent [71, 76]. Incorporating expert feedback corrects such drift and significantly mitigates these issues [6, 63]. The goal is to combine the LLM's efficiency with human judgment to improve system reliability and safety.

With these definitions of the central manager and agents in SOCIA-∇, the agents are instantiated as nodes in the computation graph described in §3.3.

3.3 Textual-Gradient Optimization

Following *TextGrad* [79], we abstract our AI system as a **computation graph**, where functionally distinct agents are represented as graph nodes. Optimizing the simulator code becomes an *instance optimization problem*: we treat the code itself as an **optimization variable**, compute "*textual gradients*" for that variable via a backpropagation-like mechanism, and then apply a gradient-descent-style procedure to iteratively improve the *code variable*.

(1) Problem Definition. We study *instance optimization* of a simulator whose implementation is itself the trainable object.

- **Task brief (natural language):** D (generated by the **DAA** after executing the `task-brief-analysis` prompt).
- **Data:** task inputs I and ground-truth observables Y.
- **Variable:** $x \in \mathcal{X}$, which is a *runnable simulator program* (code string).
- **Role description (text):** r, specifies the intended interface/IO/semantics of x.
- **Hard constraints (feasible set):** $C = \{\, x : \; c_j(x) \le 0, \; j = 1, \ldots, J \,\}$. Here, *constraints* constitute an important part of our design, serving to restrict the generated simulator code so that it adheres to domain-specific requirements or predefined specifications. They are design rules for the simulator code, such as: *no syntax errors; compilable; read files according to policy; non-empty main flow; outputs match the specified schema*, etc.
- **Inequality-constrained Optimization Goal:**

$$\min_{x \in \mathcal{X}} L(x) \;\; \text{s.t.} \;\; x \in C$$

  Our objective is to ensure that, under the constraints C, the generated code x achieves minimum fidelity-loss.
- We use the notation $LLM(z)$ to denote giving z as a prompt to a language model and collecting its response.

(2) Textual Computation Graph. We model the system as a DAG $\mathcal{G} = (V, E)$, whose nodes are *black-box operators*:

- *LLM nodes* produce or edit text (e.g., code).
- *Function nodes* execute code or compute metrics.

Each node v computes $z_v = f_v(\{z_u : u \in \mathrm{Pa}(v)\})$, where f_v is a function that performs a specific capability, and $\mathrm{Pa}(v)$ denotes the parents of node v. In our setting, all nodes correspond to **agents** in SOCIA-∇ with different abilities. Although f_v may be *non-differentiable*, we enable *backward signal flow* via *textual gradients*—natural-language critiques that explain how to change an upstream text variable to reduce downstream loss.

(3) Forward Pass and Loss with Constraints. We first perform *code initialization* by synthesizing the code variable: $x_0 = f_{\mathrm{LLM}}(D, r)$, where the *input* is the task description D (natural-language brief) and the role description r (the required interface/constraints for the simulator code). The *output* is the initial simulator program x_0 as text (simulator code). The *job* of this mapping is to turn the brief into an executable baseline implementation that already adheres to the required API, I/O policy, determinism, and output schema (with **CGA** executing `code-generation` prompt).

At *iteration* t, we use **SEA** to perform the *forward execution*:

$$o_t = f_{\mathrm{Sim}}(x_t, I).$$

Here, the *input* is the current simulator code x_t and the evaluation inputs I (validation data). The *output* is the simulator outputs o_t (e.g., trajectories, curves, logs, intermediate traces). The *job* of this mapping is to compile/run x_t on I, producing the observables needed for scoring; it also yields run metadata (errors, timing, determinism checks) that is later used as evidence for feedback.

After the forward computation, we employ **REA** to compare the simulator's predictions against the ground-truth observables to compute the *loss*:

$$L_t = \ell(o_t, Y) + \lambda \sum_j \max\big(0, c_j(x_t)\big).$$

The *input* is the simulator outputs o_t and the ground-truth observables Y. The *output* is a scalar loss L_t (lower is better). The *job* of this mapping is to measure how closely the simulator matches reality, e.g., via MSE/MAE over curves, distributional distances (JS), or task-specific metrics.

In parallel, we use **REA** to enforce *constraint satisfaction*: throughout, we require $x_t \in C$ (no syntax errors; compilable; policy-compliant I/O; non-empty main flow; output schema conformity; etc.). Violations are both *forbidden* for the final solution and *used as evidence* in the backward step.

(4) Textual Gradients (Backward as Text). In our graph, x_t has one numeric successor ("*simulate & score*"). Therefore, by the chain rule, we have:

$$\frac{\partial L_t}{\partial x_t} = \frac{\partial L_t}{\partial o_t} \cdot \frac{\partial o_t}{\partial x_t} = \nabla_{\mathrm{LLM}}(x_t, o_t, \tfrac{\partial L_t}{\partial o_t}).$$

Here, we use ∇_{LLM} to compose natural-language feedback such as "*This prediction can be improved by …*" where the feedback *describes how to modify the variable* to improve the downstream objective, *analogous to gradients* in standard Stochastic Gradient Descent (SGD) [35].

We construct evidence $\frac{\partial L_t}{\partial o_t}$ to describe the criticisms on the *prediction* o_t (completed by the **FGA** using the `output-criticism` prompt.): *diffs* between o_t and Y revealed by evaluation metrics, and *constraint violations* $\{c_j(x_t) > 0\}$ with failing tests (e.g., compile errors, IO errors, abnormal outputs discovered during simulator

execution). Then, we have:

$$g_t \triangleq \frac{\partial L_t}{\partial x_t} = \nabla_{\text{LLM}}\Big(x_t, o_t, \tfrac{\partial L_t}{\partial o_t}\Big) \tag{1}$$

$$\begin{aligned} \triangleq\ & \text{LLM}\Big(\text{We run simulator } x_t \text{ and obtain output } o_t: \\ & \{o_t \mid x_t\}. \\ & \text{Below are the critiques on } o_t\text{: } \big\{\tfrac{\partial L_t}{\partial o_t}\big\}. \\ & \text{Explain how to improve } x_t.\Big) \end{aligned} \tag{2}$$

The resulting *textual gradient* is *a set of localized, actionable code-edit patches.*

(5) Textual Momentum. In standard SGD, momentum (Polyak Heavy-Ball) [58] uses a linear combination of past gradients and the most recent one to define a new gradient for updating a variable. Analogously, to *stabilize long-horizon editing*, we maintain a momentum buffer m_t as an *exponentially decayed synopsis* of past critiques:

$$m_t = \text{Decay}_\beta(m_{t-1}) \ \oplus\ g_t,$$

where $\oplus$ denotes *ordered merge*, and Decay_β down-weights stale items (keep last K iterations).

Concretely, we *keep track of past gradients* by maintaining a *historic log* $\{M\}$ that records the optimization history across iterations. After each code optimization round, we append the current gradient to the tail of $\{M\}$. When forming the current **Textual Momentum**, we use $\text{Decay}_\beta(m_{t-1})$ to retrieve the *latest* K (in our implementation $K = 3$) historical gradient records from $\{M\}$, and we *extract salient edits/themes* from them to understand: (1) *what mistakes* prior code versions made (to avoid repeating them); (2) *how we previously fixed issues* (to inform the current round). We then merge $\text{Decay}_\beta(m_{t-1})$ with the *current* detailed code-edit feedback g_t into m_t, which acts as the *refined gradient*.

Our Momentum buffer—which summarizes prior critiques together with the current feedback—plays the same memory/reflection role as in Reflexion [60] and the iterative self-feedback role in Self-Refine [39]. By conditioning the next update on history rather than only the latest signal, it operationalizes exactly the mechanism these methods credit for outperforming one-shot prompting. In our design, **momentum** enriches the optimizer's context, enabling progressively smaller, better-aligned edits over time and reducing the chance of repeating past mistakes. Here, the **FGA**, loaded with the `gradient-generation` prompt, is responsible for generating the *textual gradients* and *momentum-based* summarization.

(6) Projected Gradient Descent (PGD). In classical PGD [2, 12], each gradient step is followed by a projection Π_C (mapping z to a point in the feasible set C); analogously, `SOCIA-∇` implements a textual analogue of **PGD**.

Specifically, we employ **Textual Momentum** to perform *textual gradient descent–based optimization* and *variable updates*, we have:

$$\begin{aligned} x_{t+1}^{\text{unproj}} &= \text{TGD.step}(x_t, m_t; r) \\ &\triangleq \text{LLM}(\text{We have critiques on } \{x_t\}\text{: } \{m_t\}. \\ &\qquad \text{Incorporate them to optimize code.}) \end{aligned} \tag{3}$$

Here, `TGD.step` receives: (i) the current x_t; (ii) the aggregated critiques m_t; (iii) the role description r, and returns the *corrected code* x_{t+1}^{unproj}. Here we use **CGA** to execute `code-optimization` prompt to edit code.

Afterwards, we *project* x_{t+1}^{unproj} back to the feasible set using a **textual projector** Π_C that repairs violations via the *smallest necessary edits*:

$$x_{t+1} = \Pi_C(x_{t+1}^{\text{unproj}}), \qquad \Pi_C : \mathcal{T} \to \{\, x : \ c_j(x) \le 0 \ \forall j \,\}.$$

Practically, Π_C is designed as a *constraint repairer* that sequentially performs *static checks*, *interface tests*, *compilation checks*, etc.; if a violation is detected, it conducts *minimal-edit repairs* (an inner loop of **CGA**'s *code-fix* ↔ *self-check*; up to *three* attempts per iteration). We write the constraint rules into `<CONSTRAINTS>` in the **CGA**'s `code-fix` prompt so that they become the target of the projector.

(7) One-Line Summary (All Pieces Together).

Forward:	$o_t = f_{\text{Sim}}(x_t, I)$,
	$L_t = \ell(o_t, Y) + \lambda \sum_j \max\big(0, c_j(x_t)\big)$.
Backward:	$g_t = \nabla_{\text{LLM}}(x_t, o_t, \frac{\partial L_t}{\partial o_t})$
Momentum:	$m_t = \text{Decay}_\beta(m_{t-1}) \oplus g_t$
Update:	$x_{t+1}^{\text{unproj}} = \text{TGD.step}(x_t, m_t; r)$,
	$x_{t+1} = \Pi_C\Big(x_{t+1}^{\text{unproj}}\Big)$.

We iteratively optimize x, and *stop by early-stopping on validation* when L_t *plateaus* and *all constraints* hold.

4 SOCIA-∇: EXPERIMENTAL SETUP

4.1 Benchmarks

We evaluate `SOCIA-∇` on three real-world–inspired simulation tasks.

(1) User Modeling. (*cyber* space, *agent-based* model), drawn from the AgentSociety Challenge [73], uses an LLM as a tool to predict ratings for new products from users' historical star ratings on an e-commerce platform.

(2) Mask Adoption Behavior Simulation. (*social* space, *aggregate* model), models the temporal dynamics of mask-wearing decisions during a pandemic in a socially embedded population. Decisions depend on heterogeneous social ties, risk perception, and governmental interventions, with a public-health campaign on Day 10 triggering diffusion. Inspired by the data paradigms in BESSIE [43] and pandemic-era mask decision simulators [41], we generate manually perturbed, LLM-based synthetic data for 1,000 agents embedded in a multi-relational graph; *note*: from Day 10 onward, an external governmental intervention affects community decisions, and the simulator must model this intervention process.

(3) Personal Mobility Generation. (*physical* space, *agent-based* model), adopts the LLMob dataset [65] containing real-world spatiotemporal trajectories from residents in Japan. The objective is to predict each individual's next-day mobility trajectory under three settings: (a) normal → normal, (b) pandemic → pandemic, and (c) normal → pandemic using only normal-period history. *Note*: **Personal Mobility** includes both in-domain (ID) fitting (normal→normal and pandemic→pandemic; training and prediction data follow the same distribution) and out-of-domain (OOD)

extrapolation (normal→pandemic), requiring the simulator to generalize to new conditions to answer "**what-if**" questions.

4.2 Baseline Methods.

We compare SOCIA-∇ against several baselines.**AI Scientist-v2** [72] is an automated pipeline that ingests experimental data and constructs/executes ML-based models; however, due to the complexity of simulator construction, AI Scientist-v2 still requires a certain amount of *manual* code adjustments to complete code generation.

We also include **YuLan-OneSim** [66], which structures scenarios via the ODD protocol [17], generates code with a Behavior Graph, and improves it via an iterative *verify–repair–regenerate* loop; *note*: YuLan-OneSim provides scenario-aware data understanding (metrics, analysis, and real/Sim alignment) for its simulations, but it is not a general-purpose data-understanding or statistical calibration toolkit. Therefore, for fairness, we equip it with SOCIA-∇ 's data-understanding module in our implementation.

The **G-SIM** family [20] is another key baseline: it uses LLMs to generate simulator code and integrates gradient-free calibration, including evolutionary strategies (**G-SIM-ES**) and simulation-based inference (**G-SIM-SBI**), to produce calibrated, uncertainty-aware simulators; G-SIM-SBI offers stronger OOD extrapolation.

Also, we compare with **Reflexion** [60], which prompts an LLM to self-reflect on code snippets, writes verbal reflections from trial feedback and stores them in episodic memory to guide subsequent trials, and—conditioned on these reflections and observed errors—produces updated code. In our implementation, all other mechanisms mirror SOCIA-∇, except that Reflexion's self-check/self-update is used to refine simulator code.

4.3 Evaluation metrics and Implementation.

Evaluation metrics. We report **means with 95% confidence intervals**. Stochastic LLM- and simulator-based runs are repeated with five seeds [14], while SBI draws 1,000 posterior samples to propagate parameter uncertainty [16]. Task-specific metrics (following **CoT**-structured prompts, see §3.2) are: **MAE** for **User Modeling** [67]; **RMSE** for **Mask Adoption**, emphasizing large deviations; and **DARD**/**STVD** for **Personal Mobility Generation**, measuring activity–time and spatiotemporal footprint alignment. All are error distances, where **lower** ↓ indicates **better** ↑ performance.

Implementation. All SOCIA-∇ agents—and the reproduced baselines—are built on GPT-5 [46]. We re-implemented baselines either via code reproduction or via dataset-level reproduction to ensure evaluation consistency.

5 SOCIA-∇: EVALUATION

5.1 Simulator Overview

Our SOCIA-∇ derives task-specific simulator code through data-driven self-checking and self-optimization. By inspecting the generated code, we briefly summarize the construction of each simulator.

Mask Adoption. SOCIA-∇ implements the Mask Adoption Behavior Simulator as a networked system that rolls forward 40 daily steps and couples data-driven calibration with progressively richer decision mechanics. Individuals maintain states (mask status, risk perception, memory) and typed ties across family / workplace–school / community in a social network with learned edge weights. Each day, the simulator performs a two-phase update to compute a latent adoption score that aggregates (i) multi-layer social influence with temporal decay and echo-chamber reinforcement, (ii) environmental risk signals, (iii) habit formation / memory from past choices, and (iv) exogenous intervention inputs; this score is mapped through a sigmoid to yield a probabilistic adopt/keep decision. From Day 10, a policy module injects information via high-centrality nodes and tracks intervention fatigue over time.

Personal Mobility Generation. SOCIA-∇ builds the Personal Mobility Simulator as a city-scale, agent-based pipeline that consumes an individual's historical spatiotemporal traces and outputs a feasible next-day itinerary with timestamps and locations. Concretely, the system defines `Resident`, `Location`, and `Simulation Environment` primitives; ingests mobility logs; and derives resident profiles (home/work anchors, daily routines, activity preferences) via pattern extraction over step-length/interval distributions and time-of-day frequencies. An LLM-based schedule generator then plans a day as a sequence of activities (e.g., work, dining, study, shopping) and selects candidate POIs using preference- and context-aware scoring subject to spatiotemporal constraints (opening hours, travel-time budgets, inter-POI distance). The simulator supports all three experimental regimes by switching conditioning data and priors: (i) normal → normal, (ii) abnormal → abnormal, and (iii) normal → abnormal (distribution shift).

User Modeling. SOCIA-∇ implements the User Modeling Simulator as an agentic, modular architecture composed of three functional agents—Planning, Reasoning, and Memory—that respectively decompose high-level goals, infer user behavior, and persist interaction history. The Planning Agent orchestrates subtasks; the Reasoning Agent generates ratings (and review text) via LLM reasoning; and the Memory Agent conditions decisions on historical preferences and prior outputs. The system replaces ad-hoc randomness with preference-aware scoring distributions, incorporates platform awareness to control tone, style, and rating policies, and introduces recommendation logic that aligns users with products based on behavioral signals and attribute profiles.

5.2 Main Findings (Table 1)

(1)* SOCIA-∇ *dominates. On all three simulations, SOCIA-∇ attains the lowest error in 7/8 metrics: User Modeling (MAE), Personal Mobility (ID: DARD/STVD for N→N and A→A), and Mobility OOD (N→A: DARD/STVD). Only Mask Adoption RMSE is not the best (where G-SIM-SBI edges out SOCIA-∇ by 0.02). This pattern highlights the advantage of our loss-driven, constraint-aware, instance optimization loop over heterogeneous agent pipelines.

(2)* SOCIA-∇ *vs. G-SIM: wins broadly; close second on Mask. Relative to **G-SIM-ES/SBI**, SOCIA-∇ excels on **User Modeling** and **Mobility** because these tasks benefit from *LLM-based reasoning over data and structure* and from *textual-gradient* updates that *repair specific code components* (add/modify logic, fix functions, adjust data handling, then calibrate parameters as needed). By design, **G-SIM** does *not* invoke LLMs; it constructs simulators by assembling

Table 1: Evaluation results (±denotes 95% CIs) on three simulation tasks, and lower values indicate better performance. The best and second-best results are highlighted in bold and underlined. Mob. N→N, Mob. A→A, and Mob. N→A denote Personal Mobility (training on normal period data and predicting normal period trajectory), abnormal-to-abnormal (pandemic) prediction, and abnormal prediction using only normal-period history, respectively.

Methods ↓	User.	Mask.	Mob. N→N (ID)		Mob. A→A (ID)		Mob. N→A (OOD)	
	MAE ↓	RMSE ↓	DARD ↓	STVD ↓	DARD ↓	STVD ↓	DARD ↓	STVD ↓
AI Scientist-v2	0.89±0.10	0.45±0.07	0.77±0.06	0.82±0.06	0.78±0.06	0.83±0.06	0.80±0.07	0.85±0.06
Yulan-OneSim	0.72±0.08	0.42±0.04	0.53±0.06	0.59±0.07	0.58±0.04	0.61±0.05	0.69±0.07	0.70±0.04
G-SIM-ES	0.60±0.07	0.30±0.08	0.45±0.02	0.48±0.03	0.52±0.04	0.56±0.04	0.58±0.05	0.64±0.05
G-SIM-SBI	0.69±0.08	**0.20±0.10**	0.54±0.03	0.56±0.03	0.47±0.03	0.53±0.03	0.60±0.05	0.66±0.05
Reflexion	0.57±0.08	0.36 ±0.04	0.42±0.05	0.43±0.07	0.46±0.04	0.50±0.05	0.55±0.04	0.61±0.07
SOCIA-∇	**0.54±0.06**	0.22±0.07	**0.40±0.02**	**0.41±0.03**	**0.43±0.03**	**0.46±0.03**	**0.50±0.05**	**0.59±0.06**

mathematical models and functional modules only. Consequently, on tasks that *only* require a fixed mathematical formulation—such as **Mask Adoption** (essentially *parameter calibration + exogenous intervention modeling*)—G-SIM's *gradient-free* calibration (**ES/SBI**) is naturally strong. That said, even though SOCIA-∇ is *not* a purpose-built parameter-calibration method, it *nearly matches* G-SIM on Mask Adoption (*0.22 vs. 0.20 RMSE*) thanks to *constrained repairs* and *projection (PGD)*, while *clearly outperforming* G-SIM on tasks that *require reasoning and structural edits* (**User Modeling, Mobility**).

(3) SOCIA-∇ vs. Reflexion: loss-aligned, component-level repairs. Both systems iterate with feedback, but **Reflexion** relies on free-form verbal reflections shaped by reward/success signals and episodic memory; it *does not define a graph-wide loss or a backprop rule*, so guidance can be looser and updates typically adjust *future decisions/strategies* rather than issuing *component-level patches tied to a loss*. In contrast, **SOCIA-∇** performs *Textual-Gradient Descent (TGD)* over a *computation graph*, routing critiques grounded in *metric diffs, failures, and constraint violations* directly to the *offending component* (e.g., a specific function), then enforcing feasibility via *projection* and *hard constraints*—yielding *small, verifiable patches* that preserve working parts and producing *consistent gains over Reflexion across all metrics*.

(4) Strong ID fitting and OOD extrapolation. Across all three tasks, SOCIA-∇ delivers the strongest overall performance: it achieves the best **User Modeling** error (MAE) and the best **Mobility** scores in both **ID** (N→N, A→A) and **OOD** (N→A) settings, while ranking a close second on **Mask Adoption** (RMSE)—narrowly behind G-SIM-SBI. This pattern indicates that *data-calibrated textual gradients + Momentum + PGD* not only fit in-distribution behavior but also adapt under distribution shift via targeted edits to dynamics, priors, and policy modules; meanwhile, *projection* keeps every update runnable, deterministic, and schema-compliant.

(5) Why YuLan-OneSim and AI Scientist-v2 trail. **YuLan-OneSim** excels at ODD/behavior-graph blueprinting and iterative verify–repair during generation, *but it does not specify an inner-loop, loss-aligned code–refinement step that uses a test/validation loss to directly edit the simulator code itself*. By contrast, SOCIA-∇ 's explicit goal is a *data-calibrated simulator* whose *code is continuously repaired*—under hard constraints—to *reduce a measurable test/validation loss* via *minimal, testable patches*. In other words, where YuLan-OneSim lacks a *closed-loop, loss-driven code-editing procedure* that optimizes a dataset-defined objective (e.g., validation loss), the SOCIA-∇ framework *makes this central*. Likewise, **AI Scientist-v2** is tailored to *ML*-based experimentation rather than simulator construction, and it *does not provide data calibration* for instance-level code repair. These design gaps align with their weaker results across tasks compared to SOCIA-∇.

Takeaway. SOCIA-∇'s textual computation graph + TGD with Momentum and PGD delivers loss-aligned, constraint-aware, and localized code repairs, translating directly into superior accuracy in-domain and under shift, and outperforming baselines that either lack data-calibrated inner loops (YuLan-OneSim), specialize in parameter-only calibration (G-SIM), or rely on reflection without backprop-through-system variables (Reflexion).

5.3 Qualitative Study

(1) LLM Calling. LLM calling is the core mechanism for constructing *agent-based* models (**Personal Mobility Generation** and **User Modelling**, see §5.1). When building such simulators, one must craft well-structured contextual prompts to drive the LLM in simulating individual behaviors [37]. In practice, the simulator code (generated by SOCIA-∇) must first *construct information elements* for the agent: in user modelling, we aggregate a user's historical ratings, peer ratings on the target item, and cross-site rating distributions; in mobility, we assemble nuanced resident profiles and personalized schedules to enhance behavioral diversity and realism. A central challenge is then how to *package* these elements into prompts—and how to script the prompt instructions—to elicit reliable LLM reasoning about user behavior. In SOCIA-∇'s design, this entire workflow is unified *inside the code*: SOCIA-∇ treats the optimization target as a *code variable* and apply *textual gradients* to update the code itself. Consequently, during each textual-gradient step the LLM's self-check and self-correction not only revise *simulator logic* but also *co-optimize information-element construction* and *prompt formulation*. This integrated approach subsumes prompt optimization into code repair, enabling a single, end-to-end loop that jointly improves data aggregation, prompt elicitation, and executable simulator quality.

Table 2: On the User Modeling (User.), Mask Adoption (Mask.), and Personal Mobility (normal→pandemic) (Mob. N→A) tasks, we conducted an ablation study using SOCIA-∇ as the baseline. Larger positive Δ values indicate greater performance degradation.

Models ↓	User.	Mask.	Mob. N→A
	ΔMAE ↓	ΔRMSE ↓	ΔDARD ↓
SOCIA-∇	-	-	-
w/o mom.	+0.05±0.02	+0.07±0.03	+0.09±0.04
w/o proj.	+0.06±0.02	+0.10±0.03	+0.16±0.05
w/o CoT	+0.08±0.04	+0.12±0.06	+0.20±0.07
w/o HITL	+0.11±0.04	+0.17±0.04	+0.26±0.06
w/o iter	+0.18±0.06	+0.28±0.07	+0.38±0.09

(2) Exogenous Interventions and OOD. In **Mask Adoption** (a *what-if* intervention setting) and **Personal Mobility** (OOD forecasting under shift), SOCIA-∇ performs strongly (Mask: *second-best*, close to G-SIM-SBI; Mobility: best), because the *generated simulator code* explicitly anticipates and absorbs distributional changes. For *Mask Adoption*, *iterative code upgrades* add: (i) *government intervention* handlers and *intervention fatigue*; (ii) *habit formation* and *dynamic decision thresholds*; (iii) refined *social influence* (echo chambers, environmental risk), two-phase propagation, and *dynamic decay*; (iv) *multi-relationship networks* (family/work/community) with weights, *cluster-specific parameters*, and memory effects; (v) a *calibrate_parameters* routine with sigmoid/probabilistic decisions. For *Personal Mobility*, *code evolves* to OOD-aware components: *resident personas* and *daily pattern extraction*; *motivation/history* modeling; *LLM-based profile & schedule synthesis*; *POI selection* and *location-aware scheduling* (distance/opening hours); plus *transport modes* and shortest-path routing. Crucially, SOCIA-∇ applies *textual gradients with Momentum and PGD*: metric diffs and violations back-propagate to offending functions; the optimizer proposes *minimal, verifiable* patches—coupling *intervention-aware* and *shift-aware* code with a *loss-aligned* loop to deliver the observed ID (N→N, A→A) and OOD (N→A) gains.

5.4 Ablation Study (Table 2)

We conduct an ablation study to quantify the contribution of each component in **SOCIA-∇**. Unless otherwise noted, we report performance deltas (Δ) relative to the full model on three tasks: **User Modeling** (ΔMAE), **Mask Adoption** (ΔRMSE), and **Personal Mobility (normal→abnormal, OOD)** (ΔDARD). Variants are:

- **w/o mom.**: remove the **Momentum** buffer (history of critiques/patches).
- **w/o proj.**: remove **Projected Gradient Descent (PGD)**, i.e., no constraint-based projection/repair.
- **w/o CoT**: disable **chain-of-thought** guidance when the Data Analysis Agent derives the task brief (use only a basic task-analysis prompt).
- **w/o HITL**: remove **human-in-the-loop** expert confirmation when drafting/refining the task brief.
- **w/o iter**: disable **iterative optimization**, i.e., use only the first version of the code produced by the generation agent.

Results Summary. On **Mask Adoption**, Table 2 shows consistent degradations. Extending this to **User Modeling** and **Mobility (OOD: N→A)** yields a coherent pattern: (i) *OOD extrapolation is more sensitive* to ablations than Mask/User; (ii) **User Modeling** is especially sensitive to CoT/HITL (schema construction and prompt design), while PGD and Momentum remain beneficial; and (iii) removing *iteration* is most damaging across tasks.

Component-wise Analysis.

(1) Momentum (w/o mom). Removing Momentum erases historical critiques, yielding *moderate* Mask degradation (≈ +0.07), *smaller* impact on User (≈ +0.05), and *larger* OOD Mobility drop (≈ +0.09)—confirming Momentum prevents repeated mistakes and stabilizes long-horizon edits, especially under distribution shift.

(2) Constraint Projection (w/o proj). Removing PGD drops the compile/IO/schema/determinism projection, letting brittle edits pass through; this yields *Mask* +0.10, *User* ΔMAE ≈ +0.06, and *OOD* ΔDARD ≈ +0.16. The larger OOD penalty underscores the need for feasibility-preserving repairs under shift.

(3) Chain-of-Thought (w/o CoT). Disabling CoT during task-brief derivation weakens *information-element construction* and prompt scaffolding: *Mask* degrades more (ΔRMSE ≈ +0.12) than *User Modeling*, and OOD Mobility suffers further (ΔDARD ≈ +0.20) as poor specifications propagate downstream.

(4) Human-in-the-Loop (w/o HITL). Removing expert confirmation causes mis-specified objectives and semantic drift, with *pronounced* degradations: Mask (+0.17), User (ΔMAE ≈ +0.11), and especially OOD Mobility (ΔDARD ≈ +0.26). Expert feedback mitigates early hallucinations and anchors the design to domain constraints—crucial for robust extrapolation.

(5) Iteration (w/o iter). Using only the first code draft blocks error-driven refinement, causing the *largest* degradations—Mask (≈ +0.28), User (≈ +0.18), and OOD Mobility (≈ +0.38). Hence, iteration is essential for converging to high-fidelity simulators.

Takeaways. Iteration is indispensable—it delivers the largest gains across tasks. **HITL and CoT are critical for problem specification**, especially for User Modeling and OOD Mobility, where mis-specified briefs/prompt scaffolds propagate widely. **PGD safeguards feasibility and stability**, with outsized benefits under shift. **Momentum** leverages history to avoid regressions.

6 CONCLUSION AND FUTURE WORK

Conclusion. We introduced **SOCIA-∇**, an end-to-end, *multi-agent* framework that embeds heterogeneous agents as *nodes* in a textual computation graph—thereby unifying the multi-agent system under a single, loss-compiled abstraction and enabling seamless agent coordination via graph-based forward/backward flow. Simulator construction is posed as *instance optimization over code* with a loss-driven loop—code synthesis → execution → evaluation → textual-gradient repair—using *TGD* with *Momentum* (history-aware critiques) and *PGD-style projection* (constraint repair), complemented by CoT-guided analysis and HITL verification. Across User Modeling, Mask Adoption, and Personal Mobility, SOCIA-∇ attains state-of-the-art results overall (best on all Mobility ID/OOD metrics; close second on Mask), and ablations confirm the importance of iteration, HITL/CoT, projection, and momentum.

Future Work. We will pursue two directions. **(1) Scaling interaction complexity:** extend SOCIA-∇ from small, mostly linear exchanges to *large, data-induced agent societies* with thousands–millions of agents and richer *parallel/asynchronous* communication; upgrade the textual projector for concurrency safety and interface contracts, and incorporate distributed signals (stability, throughput, fairness) as loss terms. **(2) Domain/space generalization and model pretraining:** deploy SOCIA-∇ across diverse CPS settings to harvest per-iteration *(code, loss)* trajectories; use these as reinforcement-learning supervision to train a *simulator-code LLM* specialized for program synthesis/repair, then adopt it as the coding agent's base model—reducing reliance on proprietary GPT-like systems and improving efficiency and code quality for high-fidelity simulator generation.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. {TensorFlow}: a system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 265–283.

[2] Alekh Agarwal, Sahand Negahban, and Martin J Wainwright. 2010. Fast global convergence rates of gradient methods for high-dimensional statistical recovery. *Advances in Neural Information Processing Systems* 23 (2010).

[3] Nadia Alshahwan, Jubin Chheda, Anastasia Finegenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. 2024. Automated Unit Test Improvement using Large Language Models at Meta. arXiv:2402.09171 [cs.SE] https://arxiv.org/abs/2402.09171

[4] Bernardo Alves Furtado and Vanessa Nadalin. 2023. Policy Comparisons and Causality in an Agent-Based Model. In *Conference of the European Social Simulation Association*. Springer, 95–106.

[5] Shun-ichi Amari. 1993. Backpropagation and stochastic gradient descent method. *Neurocomputing* 5, 4-5 (1993), 185–196.

[6] Maryam Amirizaniani, Jihan Yao, Adrian Lavergne, Elizabeth Snell Okada, Aman Chadha, Tanya Roosta, and Chirag Shah. 2024. Developing a framework for auditing large language models using human-in-the-loop. *arXiv preprint arXiv:2402.09346* (2024).

[7] Yuntao Bai, Andy Jones, Kamal Ndousse, Amanda Askell, Anna Chen, Nova DasSarma, Dawn Drain, Stanislav Fort, Deep Ganguli, Tom Henighan, et al. 2022. Training a helpful and harmless assistant with reinforcement learning from human feedback. *arXiv preprint arXiv:2204.05862* (2022).

[8] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. 2010. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for scientific computing conference (SciPy)*, Vol. 4. Austin, TX, 1–7.

[9] Eric Bonabeau. 2002. Agent-based modeling: Methods and techniques for simulating human systems. *Proceedings of the national academy of sciences* 99, suppl_3 (2002), 7280–7287.

[10] Dong Chen, Shaoxin Lin, Muhan Zeng, Daoguang Zan, Jian-Gang Wang, Anton Cheshkov, Jun Sun, Hao Yu, Guoliang Dong, Artem Aliev, Jie Wang, Xiao Cheng, Guangtai Liang, Yuchi Ma, Pan Bian, Tao Xie, and Qianxiang Wang. 2024. CodeR: Issue Resolving with Multi-Agent and Task Graphs. arXiv:2406.01304 [cs.CL] https://arxiv.org/abs/2406.01304

[11] Guangyao Chen, Siwei Dong, Yu Shu, Ge Zhang, Jaward Sesay, Börje Karlsson, Jie Fu, and Yemin Shi. 2024. AutoAgents: A Framework for Automatic Agent Generation. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI 2024, Jeju, South Korea, August 3-9, 2024*. ijcai.org, 22–30. https://www.ijcai.org/proceedings/2024/3

[12] Yudong Chen and Martin J Wainwright. 2015. Fast low-rank estimation by projected gradient descent: General statistical and algorithmic guarantees. *arXiv preprint arXiv:1509.03025* (2015).

[13] Sarah Cohen-Boulakia, Khalid Belhajjame, Olivier Collin, Jérôme Chopard, Christine Froidevaux, Alban Gaignard, Konrad Hinsen, Pierre Larmande, Yvan Le Bras, Frédéric Lemoine, et al. 2017. Scientific workflows for computational reproducibility in the life sciences: Status, challenges and opportunities. *Future Generation Computer Systems* 75 (2017), 284–298.

[14] Cédric Colas, Olivier Sigaud, and Pierre-Yves Oudeyer. 2018. How many random seeds? statistical power analysis in deep reinforcement learning experiments. *arXiv preprint arXiv:1806.08295* (2018).

[15] Jiaxun Cui, Chen Tang, Jarrett Holtz, Janice Nguyen, Alessandro G Allievi, Hang Qiu, and Peter Stone. 2025. Talking Vehicles: Cooperative Driving via Natural Language. https://openreview.net/forum?id=VYlfoA8I6A

[16] Maciej Falkiewicz, Naoya Takeishi, Imahn Shekhzadeh, Antoine Wehenkel, Arnaud Delaunoy, Gilles Louppe, and Alexandros Kalousis. 2023. Calibrating neural simulation-based inference with differentiable coverage probability. *Advances in Neural Information Processing Systems* 36 (2023), 1082–1099.

[17] Volker Grimm, Uta Berger, Donald L DeAngelis, J Gary Polhill, Jarl Giske, and Steven F Railsback. 2010. The ODD protocol: a review and first update. *Ecological modelling* 221, 23 (2010), 2760–2768.

[18] Volker Grimm, Steven F Railsback, Christian E Vincenot, Uta Berger, Cara Gallagher, Donald L DeAngelis, Bruce Edmonds, Jiaqi Ge, Jarl Giske, Juergen Groeneveld, et al. 2020. The ODD protocol for describing agent-based and other simulation models: A second update to improve clarity, replication, and structural realism. *Journal of Artificial Societies and Social Simulation* 23, 2 (2020).

[19] Jeremy Harper. 2024. AutoGenesisAgent: Self-Generating Multi-Agent Systems for Complex Tasks. arXiv:2404.17017 [cs.MA] https://arxiv.org/abs/2404.17017

[20] Samuel Holt, Max Ruiz Luyten, Antonin Berthon, and Mihaela van der Schaar. 2025. G-Sim: Generative Simulations with Large Language Models and Gradient-Free Calibration. *CoRR* abs/2506.09272 (2025). https://doi.org/10.48550/ARXIV.2506.09272 arXiv:2506.09272

[21] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2024. MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net. https://openreview.net/forum?id=VtmBAGCN7o

[22] Susan Howick, Itamar Megiddo, et al. 2024. A framework for conceptualising hybrid system dynamics and agent-based simulation models. *European Journal of Operational Research* 315, 3 (2024), 1153–1166.

[23] Dong Huang, Jie M Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. 2023. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010* (2023).

[24] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*. 675–678.

[25] WANG JIAWEI, Renhe Jiang, Chuang Yang, Zengqing Wu, Ryosuke Shibasaki, Noboru Koshizuka, Chuan Xiao, et al. 2024. Large language models as urban residents: An llm agent framework for personal mobility generation. *Advances in Neural Information Processing Systems* 37 (2024), 124547–124574.

[26] Haolin Jin, Linghan Huang, Haipeng Cai, Jun Yan, Bo Li, and Huaming Chen. 2025. From LLMs to LLM-based Agents for Software Engineering: A Survey of Current, Challenges and Future. arXiv:2408.02479 [cs.SE] https://arxiv.org/abs/2408.02479

[27] Haolin Jin, Zechao Sun, and Huaming Chen. 2024. RGD: Multi-LLM Based Agent Debugger via Refinement and Generation Guidance. arXiv:2410.01242 [cs.SE] https://arxiv.org/abs/2410.01242

[28] Chenlu Ju, Jiaxin Liu, Shobhit Sinha, Hao Xue, and Flora Salim. 2025. TrajLLM: A Modular LLM-Enhanced Agent-Based Framework for Realistic Human Trajectory Simulation. *CoRR* abs/2502.18712 (2025). https://doi.org/10.48550/ARXIV.2502.18712 arXiv:2502.18712

[29] Manpreet Kaur, Flora D. Salim, Yongli Ren, Jeffrey Chan, Martin Tomko, and Mark Sanderson. 2020. Joint Modelling of Cyber Activities and Physical Context to Improve Prediction of Visitor Behaviors. *ACM Trans. Sens. Networks* 16, 3 (2020), 28:1–28:25. https://doi.org/10.1145/3393692

[30] Marc C Kennedy and Anthony O'Hagan. 2001. Bayesian calibration of computer models. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 63, 3 (2001), 425–464.

[31] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Saiful Haq, Ashutosh Sharma, Thomas T Joshi, Hanna Moazam, Heather Miller, et al. [n.d.]. Dspy: Compiling declarative language model calls into state-of-the-art pipelines. In *The Twelfth International Conference on Learning Representations*.

[32] Averill M Law, W David Kelton, and W David Kelton. 2007. *Simulation modeling and analysis*. Vol. 3. Mcgraw-hill New York.

[33] Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. 2023. Camel: Communicative agents for" mind" exploration of large language model society. *Advances in Neural Information Processing Systems* 36 (2023), 51991–52008.

[34] Xuechen Li, Tianyi Zhang, Yann Dubois, Rohan Taori, Ishaan Gulrajani, Carlos Guestrin, Percy Liang, and Tatsunori B Hashimoto. 2023. Alpacaeval: An automatic evaluator of instruction-following models.

[35] Yuanzhi Li and Yingyu Liang. 2018. Learning overparameterized neural networks via stochastic gradient descent on structured data. *Advances in neural information processing systems* 31 (2018).

[36] Fang Liu, Yang Liu, Lin Shi, Houkun Huang, Ruifeng Wang, Zhen Yang, Li Zhang, Zhongqi Li, and Yuchi Ma. 2024. Exploring and evaluating hallucinations in llm-powered code generation. *arXiv preprint arXiv:2404.00971* (2024).

[37] Yuxuan Lu, Jing Huang, Yan Han, Bingsheng Yao, Sisong Bei, Jiri Gesi, Yaochen Xie, Qi He, Dakuo Wang, et al. 2025. Prompting is Not All You Need! Evaluating LLM Agent Simulation Methodologies with Real-World Online Customer Behavior Data. *arXiv preprint arXiv:2503.20749* (2025).

[38] Charles M Macal and Michael J North. 2005. Tutorial on agent-based modeling and simulation. In *Proceedings of the Winter Simulation Conference, 2005*. IEEE, 14–pp.

[39] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2023. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems* 36 (2023), 46534–46594.

[40] Josie McCulloch, Jiaqi Ge, Jonathan A Ward, Alison Heppenstall, J Gareth Polhill, and Nick Malleson. 2022. Calibrating agent-based models using uncertainty quantification methods. *Journal of Artificial Societies and Social Simulation* 25, 2 (2022).

[41] Konstantinos Mitsopoulos, Lawrence Baker, Christian Lebiere, Peter Pirolli, Mark Orr, and Raffaele Vardavas. 2023. Masking behaviors in epidemiological networks with cognitively-plausible reinforcement learning. *arXiv preprint arXiv:2312.03301* (2023).

[42] Lavinia Mitton, Holly Sutherland, and Melvyn Weeks. 2000. *Microsimulation modelling for policy analysis: challenges and innovations*. Cambridge University Press.

[43] Henning S. Mortveit, Stephen C. Adams, Faraz Dadgostari, Samarth Swarup, and Peter A. Beling. 2022. BESSIE: A Behavior and Epidemic Simulator for Use With Synthetic Populations. *CoRR* abs/2203.11414 (2022). https://doi.org/10.48550/ARXIV.2203.11414 arXiv:2203.11414

[44] Eduardo Mosqueira-Rey, Elena Hernández-Pereira, David Alonso-Ríos, José Bobes-Bascarán, and Ángel Fernández-Leal. 2023. Human-in-the-loop machine learning: a state of the art. *Artificial Intelligence Review* 56, 4 (2023), 3005–3054.

[45] Sriraam Natarajan, Saurabh Mathur, Sahil Sidheekh, Wolfgang Stammer, and Kristian Kersting. 2025. Human-in-the-loop or AI-in-the-loop? Automate or Collaborate?. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 39. 28594–28600.

[46] OpenAI. 2025. Introducing gpt-5. (2025).

[47] Shabnam Pasandideh, Pedro Pereira, and Luís Gomes. 2022. Cyber-Physical-Social Systems: Taxonomy, Challenges, and Opportunities. *IEEE Access* 10 (2022), 42404–42419. https://doi.org/10.1109/ACCESS.2022.3167441

[48] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[49] Jinghua Piao, Yuwei Yan, Jun Zhang, Nian Li, Junbo Yan, Xiaochong Lan, Zhihong Lu, Zhiheng Zheng, Jing Yi Wang, Di Zhou, Chen Gao, Fengli Xu, Fang Zhang, Ke Rong, Jun Su, and Yong Li. 2025. AgentSociety: Large-Scale Simulation of LLM-Driven Generative Agents Advances Understanding of Human Behaviors and Society. *CoRR* abs/2502.08691 (2025). https://doi.org/10.48550/ARXIV.2502.08691 arXiv:2502.08691

[50] Felipe Maia Polo, Ronald Xu, Lucas Weber, Mírian Silva, Onkar Bhardwaj, Leshem Choshen, Allysson Flavio Melo de Oliveira, Yuekai Sun, and Mikhail Yurochkin. 2024. Efficient multi-prompt evaluation of LLMs. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*. 22483–22512.

[51] Reid Pryzant, Dan Iter, Jerry Li, Yin Lee, Chenguang Zhu, and Michael Zeng. 2023. Automatic Prompt Optimization with "Gradient Descent" and Beam Search. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. 7957–7968.

[52] Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, et al. 2023. Chatdev: Communicative agents for software development. *arXiv preprint arXiv:2307.07924* (2023).

[53] Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, et al. 2024. ChatDev: Communicative Agents for Software Development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 15174–15186.

[54] Sumedh Rasal. 2024. LLM Harmony: Multi-Agent Communication for Problem Solving. https://doi.org/10.48550/arXiv.2401.01312 arXiv:2401.01312 [cs]

[55] Yongli Ren, Martin Tomko, Flora Dilys Salim, Jeffrey Chan, Charles L. A. Clarke, and Mark Sanderson. 2018. A Location-Query-Browse Graph for Contextual Recommendation. *IEEE Trans. Knowl. Data Eng.* 30, 2 (2018), 204–218. https://doi.org/10.1109/TKDE.2017.2766059

[56] Yongli Ren, Martin Tomko, Flora D. Salim, Jeffrey Chan, and Mark Sanderson. 2018. Understanding the predictability of user demographics from cyber-physical-social behaviours in indoor retail spaces. *EPJ Data Sci.* 7, 1 (2018), 1. https://doi.org/10.1140/EPJDS/S13688-017-0128-2

[57] Abel Salinas and Fred Morstatter. 2024. The butterfly effect of altering prompts: How small changes and jailbreaks affect large language model performance. *arXiv preprint arXiv:2401.03729* (2024).

[58] Damien Scieur and Fabian Pedregosa. 2020. Universal average-case optimality of Polyak momentum. In *International conference on machine learning*. PMLR, 8565–8572.

[59] Yu Shang, Yu Li, Keyu Zhao, Likai Ma, Jiahe Liu, Fengli Xu, and Yong Li. 2025. AgentSquare: Automatic LLM Agent Search in Modular Design Space. https://doi.org/10.48550/arXiv.2410.06153 arXiv:2410.06153 [cs]

[60] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems* 36 (2023), 8634–8652.

[61] Chaehan So. 2020. Human-in-the-loop design cycles–a process framework that integrates design sprints, agile processes, and machine learning with humans. In *International Conference on Human-Computer Interaction*. Springer, 136–145.

[62] Jiakai Tang, Heyang Gao, Xuchen Pan, Lei Wang, Haoran Tan, Dawei Gao, Yushuo Chen, Xu Chen, Yankai Lin, Yaliang Li, et al. 2024. GenSim: A General Social Simulation Platform with Large Language Model based Agents. *arXiv preprint arXiv:2410.04360* (2024).

[63] SMTI Tonmoy, SM Zaman, Vinija Jain, Anku Rani, Vipula Rawte, Aman Chadha, and Amitava Das. 2024. A comprehensive survey of hallucination mitigation techniques in large language models. *arXiv preprint arXiv:2401.01313* 6 (2024).

[64] H Van Dyke Parunak, Robert Savit, and Rick L Riolo. 1998. Agent-based modeling vs. equation-based modeling: A case study and users' guide. In *International workshop on multi-agent systems and agent-based simulation*. Springer, 10–25.

[65] Jiawei Wang, Renhe Jiang, Chuang Yang, Zengqing Wu, Makoto Onizuka, Ryosuke Shibasaki, Noboru Koshizuka, and Chuan Xiao. 2024. Large language models as urban residents: An llm agent framework for personal mobility generation. *Advances in Neural Information Processing Systems* 37 (2024), 124547–124574.

[66] Lei Wang, Heyang Gao, Xiaohe Bo, Xu Chen, and Ji-Rong Wen. 2025. Yulan-onesim: Towards the next generation of social simulator with large language models. *arXiv preprint arXiv:2505.07581* (2025).

[67] Yancheng Wang, Ziyan Jiang, Zheng Chen, Fan Yang, Yingxue Zhou, Eunah Cho, Xing Fan, Yanbin Lu, Xiaojiang Huang, and Yingzhen Yang. 2024. RecMind: Large Language Model Powered Agent For Recommendation. In *Findings of the Association for Computational Linguistics: NAACL 2024, Mexico City, Mexico, June 16-21, 2024*, Kevin Duh, Helena Gómez-Adorno, and Steven Bethard (Eds.). Association for Computational Linguistics, 4351–4364. https://doi.org/10.18653/V1/2024.FINDINGS-NAACL.271

[68] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[69] Xingjiao Wu, Luwei Xiao, Yixuan Sun, Junhang Zhang, Tianlong Ma, and Liang He. 2022. A survey of human-in-the-loop for machine learning. *Future Generation Computer Systems* 135 (2022), 364–381.

[70] Ziwei Xu, Sanjay Jain, and Mohan Kankanhalli. 2024. Hallucination is inevitable: An innate limitation of large language models. *arXiv preprint arXiv:2401.11817* (2024).

[71] Tianci Xue, Ziqi Wang, Zhenhailong Wang, Chi Han, Pengfei Yu, and Heng Ji. 2023. Rcot: Detecting and rectifying factual inconsistency in reasoning by reversing chain-of-thought. *arXiv preprint arXiv:2305.11499* (2023).

[72] Yutaro Yamada, Robert Tjarko Lange, Cong Lu, Shengran Hu, Chris Lu, Jakob Foerster, Jeff Clune, and David Ha. 2025. The ai scientist-v2: Workshop-level automated scientific discovery via agentic tree search. *arXiv preprint arXiv:2504.08066* (2025).

[73] Yuwei Yan, Yu Shang, Qingbin Zeng, Yu Li, Keyu Zhao, Zhiheng Zheng, Xuefei Ning, Tianji Wu, Shengen Yan, Yu Wang, Fengli Xu, and Yong Li. 2025. AgentSociety Challenge: Designing LLM Agents for User Modeling and Recommendation on Web Platforms. *CoRR* abs/2502.18754 (2025). https://doi.org/10.48550/ARXIV.2502.18754 arXiv:2502.18754

[74] Yuwei Yan, Qingbin Zeng, Zhiheng Zheng, Jingzhe Yuan, Jie Feng, Jun Zhang, Fengli Xu, and Yong Li. 2024. Opencity: A scalable platform to simulate urban activities with massive llm agents. *arXiv preprint arXiv:2410.21286* (2024).

[75] Ziyi Yang, Zaibin Zhang, Zirui Zheng, Yuxian Jiang, Ziyue Gan, Zhiyu Wang, Zijian Ling, Jinsong Chen, Martz Ma, Bowen Dong, et al. 2024. Oasis: Open agents social interaction simulations on one million agents. *arXiv preprint arXiv:2411.11581* (2024).

[76] Evelyn Yee, Alice Li, Chenyu Tang, Yeon Ho Jung, Ramamohan Paturi, and Leon Bergen. 2024. Faithful and unfaithful error recovery in chain of thought. In *First Conference on Language Modeling*.

[77] Weizhe Yuan, Richard Yuanzhe Pang, Kyunghyun Cho, Xian Li, Sainbayar Sukhbaatar, Jing Xu, and Jason Weston. 2024. Self-rewarding language models. In *Proceedings of the 41st International Conference on Machine Learning*. 57905–57923.

[78] Ling Yue, Nithin Somasekharan, Yadi Cao, and Shaowu Pan. 2025. Foam-Agent: Towards Automated Intelligent CFD Workflows. *arXiv preprint arXiv:2505.04997* (2025).

[79] Mert Yüksekgönül, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. 2024. TextGrad: Automatic "Differentiation" via Text. *CoRR* abs/2406.07496 (2024). https://doi.org/10.48550/ARXIV.2406.07496

arXiv:2406.07496
[80] Xinnong Zhang, Jiayu Lin, Xinyi Mou, Shiyue Yang, Xiawei Liu, Libo Sun, Hanjia Lyu, Yihang Yang, Weihong Qi, Yue Chen, et al. 2025. SocioVerse: A World Model for Social Simulation Powered by LLM Agents and A Pool of 10 Million Real-World Users. *arXiv preprint arXiv:2504.10157* (2025).