AgentEmulator: A Blockchain-Empowered Testbed for Trustworthy AI Agent Infrastructure

Jian Zheng , Jianbo Xiong, Feihong Hu, Huawei Huang*

***Abstract*—Research on trustworthy AI agent infrastructure requires experimental tools that connect specified agent behavior to blockchain execution and observable account effects. This paper presents AGENTEMULATOR, a trace-driven experimentation platform built on BlockEmulator-X. In version 1.0, researchers describe agent lifecycle actions and payment intents in JSONL and configure the underlying blockchain. The `agentSupervisor` module validates lifecycle constraints, deterministically derives identities, compiles transactions with action-to-hash mappings, and manages backend execution. The current release follows the User-specified Original Sequence policy, with no additional transaction scheduling algorithm. It records identity registration and revocation calls and executes direct payments, then produces transaction records, reconstructed per-agent ledgers, four account-balance views, and an HTML gallery. We describe the system architecture and formalize the distinctions among logical trace order, backend execution order, and visualization order. A demonstration of experimentation with 100 agents and a trace containing 10,000 transactions on top of a sharding blockchain illustrates the workflow and its accounting views. Conditional account-balance invariants and an evaluation methodology specify how to assess reproducibility, payment completion, and execution overhead. The demonstration does not establish performance or scalability. We encourage other researchers to extend the experimental foundation by designing and integrating their own user-defined mechanisms, identity-related smart contracts, and adaptive interactions of AI agents.**

***Index Terms*—AI agent infrastructure, agent emulation, blockchain, trace-driven experimentation.**

I. INTRODUCTION

AI agents increasingly interact with digital services and other agents [1]. Infrastructure for these interactions must support participation, value exchange, and activity records. Studying such infrastructure calls for a configurable testbed that brings agent behavior and blockchain execution into the same experimental workflow.

Recent research develops agent collaboration and evaluation along several complementary directions. Internet of Agents (IoA) [2] addresses heterogeneous agent integration and dynamic collaboration. AgentSquare [3] searches over modular agent designs, while MultiAgentBench [4] evaluates collaboration and competition through interactive scenarios. These systems motivate experiments that pair agent interactions with blockchain records and payments.

At the blockchain level, BlockEmulator [5] supports experimentation with sharding mechanisms. Researchers exploring agent infrastructure also need a convenient way to describe agent actions, prepare transaction workloads, configure the blockchain environment, and inspect experiment outputs.

Three practical requirements follow. First, a workload interface should describe agent participation and payments without requiring users to manually construct blockchain transactions. Second, experiment setup should coordinate the agent configuration, blockchain parameters, and backend processes. Third, the resulting data should support both individual-agent inspection and an overview of activity across the experiment.

We present AGENTEMULATOR[1], a blockchain-empowered testbed built on BlockEmulator-X[2]. Developed by HuangLab at the School of Software Engineering, Sun Yat-sen University, the platform supports research toward trustworthy AI agent infrastructure. In AGENTEMULATOR v1.0, researchers only need to supply a JSONL trace file containing the agent's `join`, `pay`, and `leave` actions, optionally mixed with ordinary account transfer transactions. The `agentSupervisor` module prepares the transaction dataset, launches the configured blockchain, and collects experiment outputs. Plotting scripts generate per-agent balance views and an HTML gallery.

Fig. 1 summarizes the workflow from trace preparation to result presentation. The highlighted *user-defined agent mechanisms or algorithms* module identifies a direction for research extension. AGENTEMULATOR v1.0 supplies a trace-driven baseline using the *User-specified Original Sequence* policy. Agent identity, auditing, settlement, incentives, and governance define the broader research agenda.

This paper makes the following **contributions**.

- We present a blockchain-empowered testbed that integrates agent behavior, transaction construction, configurable blockchain execution, and experiment output generation. Blockchains provide a reliable transaction platform for high-frequency micropayments between agents.
- We describe the modular implementation centered on `agentSupervisor`, including lifecycle handling, two-layer configuration, backend orchestration, and support for research-specific extensions.
- We demonstrate the result-presentation workflow through four balance views from a 100-agent experiment and outline practical evaluation settings for the testbed.

Section II introduces the background and related experimental tools. Section III presents the preliminaries of agent actions. Section IV describes the implementation, and Section V demonstrates the experimental methodology.

Version: Oct.10, 2026. AI was used to refine writing. All authors are with the School of Software Engineering, Sun Yat-sen University, Zhuhai Campus, Guangdong 519082, China (Huawei Huang is the corresponding author. E-mail: huanghw28@mail.sysu.edu.cn).

[1]AgentEmulator: https://github.com/HuangLab-SYSU/agent-emulator

[2]BlockEmulator-X: https://github.com/HuangLab-SYSU/block-emulator-x

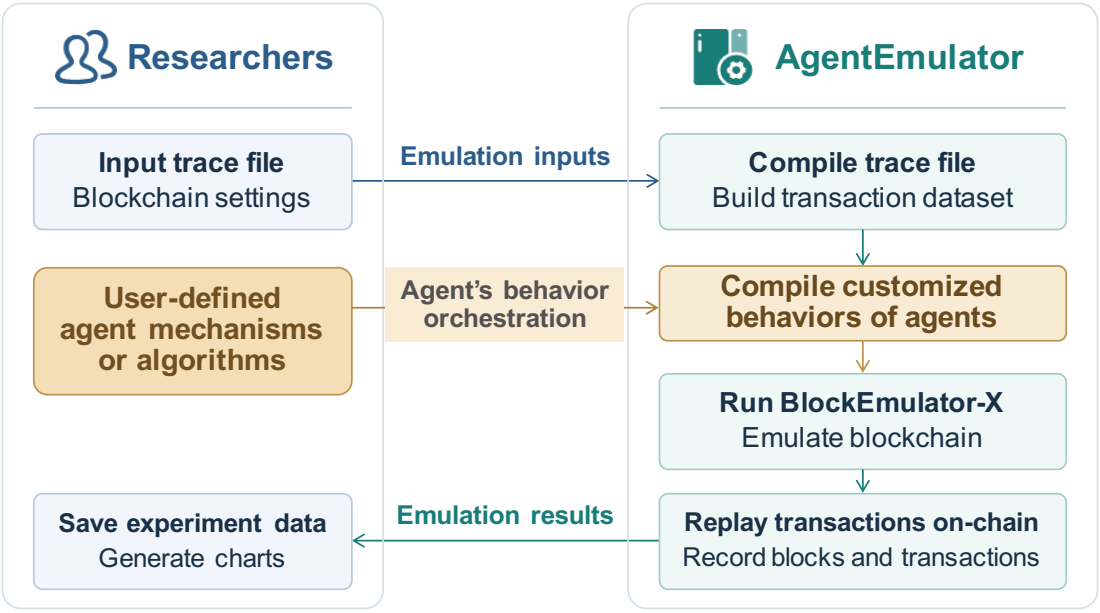


Fig. 1. General workflow of AGENTEMULATOR. Researchers must provide traces and blockchain settings and inspect saved records and plots. The highlighted *User-defined agent mechanisms or algorithms* module expresses the broader research scope. AGENTEMULATOR v1.0 uses the *User-specified Original Sequence* policy and includes no additional agent behavior orchestration or scheduling algorithm.

II. BACKGROUND AND RELATED EXPERIMENTAL TOOLS

A. AI Agent

An AI agent interacts with an environment to pursue a task by interpreting observations, selecting actions, and using execution feedback. Language-model-based agents can combine reasoning, memory, and external tools, while multiple agents can communicate and divide work according to their roles. In AGENTEMULATOR, an agent is represented as a participant named in a behavior trace, whose externally specified actions determine when it joins, pays another participant, and leaves. This abstraction accommodates hand-written scenarios, recorded application behavior, and future agent adapters without requiring a particular reasoning model. The current workflow receives the complete action sequence before execution, allowing researchers to study infrastructure effects while holding the supplied behavior constant. The `did-simple` plugin derives deterministic experimental identifiers and maintains lifecycle state during workload preparation. These identifiers associate actions with participants but do not by themselves establish authenticated real-world identities or a deployed identity registry.

Tool Use and Multi-Agent Collaboration. Agent frameworks provide different methods for generating actions and coordinating tasks. Toolformer [6] learns when and how to call external APIs. ToolLLM [7] supports tool use through training data, model training, API retrieval, and evaluation. Reflexion [8] uses textual feedback and episodic memory to improve later decisions without changing model weights. Collaboration frameworks focus on interactions among agents. CAMEL [9] studies cooperation through role-playing and instruction following. MetaGPT [10] coordinates specialized agents through standard operating procedures. ChatDev [11] uses dialogue to coordinate software design, coding, and testing. IoA [2] connects heterogeneous agents through an integration protocol, dynamic team formation, and conversation management. Other approaches search for effective agent structures and workflows. AgentSquare [3] searches modular designs that combine planning, reasoning, tool use, and memory. AFlow [12] improves code-based workflows through Monte Carlo Tree Search and execution feedback. These frameworks can generate behaviors for infrastructure experiments. Their application events still need to be mapped to lifecycle actions and payment intents. AGENTEMULATOR currently follows the supplied action sequence under the User-specified Original Sequence policy. Agent adapters and behavior generation based on execution feedback remain directions for future development.

Agent Evaluation and Trustworthiness. Agent benchmarks evaluate different aspects of task execution. AgentBench [13] tests reasoning and decision-making across eight interactive environments. WebArena [14] evaluates task completion on realistic websites. OSWorld [15] covers computer tasks involving multiple applications and file operations. SWE-bench [16] tests whether repository changes resolve real software issues. MultiAgentBench [4] evaluates collaboration and competition through task outcomes and interaction progress. These benchmarks assess application outcomes that transaction records alone cannot capture. A completed payment confirms an agent payment, but it does not establish task quality or effective collaboration. Agent decisions also require security evaluation. AgentDojo [17] examines prompt-injection attacks and defenses when agents encounter untrusted tool outputs. A blockchain record alone cannot show whether an action was authorized or influenced by an attack. AGENTEMULATOR provides lifecycle checks, transaction records, and balance views for analysis. Task quality and agent security require additional evaluation with suitable task environments and agent runtimes.

B. Blockchain

Consensus. A blockchain maintains transaction records through distributed execution and agreement. Consensus protocols govern how transactions are disseminated, ordered, and committed. Their guarantees depend on the faults assumed by the system model. Narwhal and Tusk [18] separate transaction dissemination from ordering. Their design combines a mempool based on a directed acyclic graph (DAG) with asynchronous consensus. Bullshark [19] derives transaction order from the DAG structure and reduces latency during synchronous operation. Mysticeti [20] uses uncertified DAGs to reduce consensus latency and introduces a fast commit path for asset transfers. These protocols use different mechanisms to commit transactions. An agent trace specifies logical timestamps and submission order, but final commitment depends on backend execution. Latency evaluation therefore requires measured final-commit times.

Sharding. A sharded blockchain divides transaction processing among committees. Each shard handles part of the workload, while transfer transactions between shards require cross-committee coordination. Kronos [21] studies generic cross-shard processing that preserves atomicity under malicious clients. Its design also reduces overhead within and across shards. Logos [22] addresses robust cross-shard processing, including resistance to invalid-transaction flooding. Its design consists of broadcast, transmission, and agreement stages. These studies motivate separate analysis of intra-shard and cross-shard payments. Experiments can also examine valid and invalid traffic under normal and adversarial conditions. The resulting security and performance properties depend on the implemented protocols. An agent workload interface alone does not provide these guarantees.

Experimental Infrastructure. BlockEmulator [5] provides a modular platform for implementing and testing blockchain sharding protocols. It is the closest experimental reference for this work. AGENTEMULATOR builds on the BlockEmulator-X workflow. It adds workload preparation with agent lifecycle checks, backend orchestration, and results organized by participant. The configured backend determines consensus behavior. Each experiment must specify the backend revision, shard configuration, and enabled mechanisms. Agent frameworks can supply application events, which a trace represents as participant actions. AGENTEMULATOR converts these actions into transactions and collects account records from backend execution. Comparisons across systems require explicit descriptions of workloads, deployments, and execution mechanisms. Differences in these conditions affect how users interpret the results.

C. Experimental Requirements

The preceding background motivates three requirements for agent infrastructure experiments.

- *First, explicit input semantics and lifecycle validation* are needed to define participation, payment intent, and departure and to detect invalid transitions before execution. When traces originate from an agent framework, their construction should document how application events map to lifecycle actions and payment records. This requirement establishes the behavior interface described in Section III.
- *Second, repeatable workload preparation and execution configuration* require fixed trace bytes, seeds, initial state, and effective configurations. Backend revisions, enabled protocols, and deployment conditions must also be recorded, since deterministic workload construction does not imply an identical distributed execution schedule. Automated preparation and cluster launch should make these settings easier to apply consistently. This requirement guides the implementation and configuration mechanisms in Section IV.
- *Third, result inspection must distinguish intended actions from observed effects.* Transaction records support completion checks, while per-agent histories and aggregate balance views support different forms of account inspection. Researchers should interpret these observations separately from application-level task success and retain the records needed to explain incomplete or failed actions. Organized outputs and repeatable inspection provide the basis for the demonstration and evaluation setup in Section V.

III. PRELIMINARIES OF AGENT ACTIONS

A. Preparing a Behavior Trace

An experiment starts with a JSONL behavior trace, where each line represents either an agent action or an ordinary blockchain transfer transaction. Fig. 2 illustrates the fields of these two record types and how they map to blockchain transactions, using an agent payment as an example.

For an agent action, the `agent_id` field identifies the participant, `action` selects the operation, and `ts` specifies its logical order. A payment action additionally specifies `target`, the recipient's agent identifier, and `amount`, a positive integer transfer amount. Optional fields include `request_id` to label a payment and `params_hash` to identify a parameter version. Researchers supply agent identifiers rather than DIDs. The testbed derives experimental identities from the configured seed and identifier.

An ordinary blockchain transfer transaction specifies `sender`, `recipient`, and `value`. The first two fields are blockchain account addresses, and `value` is the transfer amount. These records provide background traffic alongside agent actions. The `agentSupervisor` orders agent actions by `ts` and resolves ties by file order.

B. Supported Agent Actions

Table I summarizes three agent actions. A `join` marks an agent active, `pay` transfers value between active agents, and `leave` marks an agent inactive. A repeated `join` while the agent is active does not generate another registration transaction. Rejoining after departure generates a new registration call. The system rejects invalid departures and payments involving inactive participants during input processing.

In AGENTEMULATOR v1.0, *registration* and *revocation* calls are recorded without a deployed DID contract updating

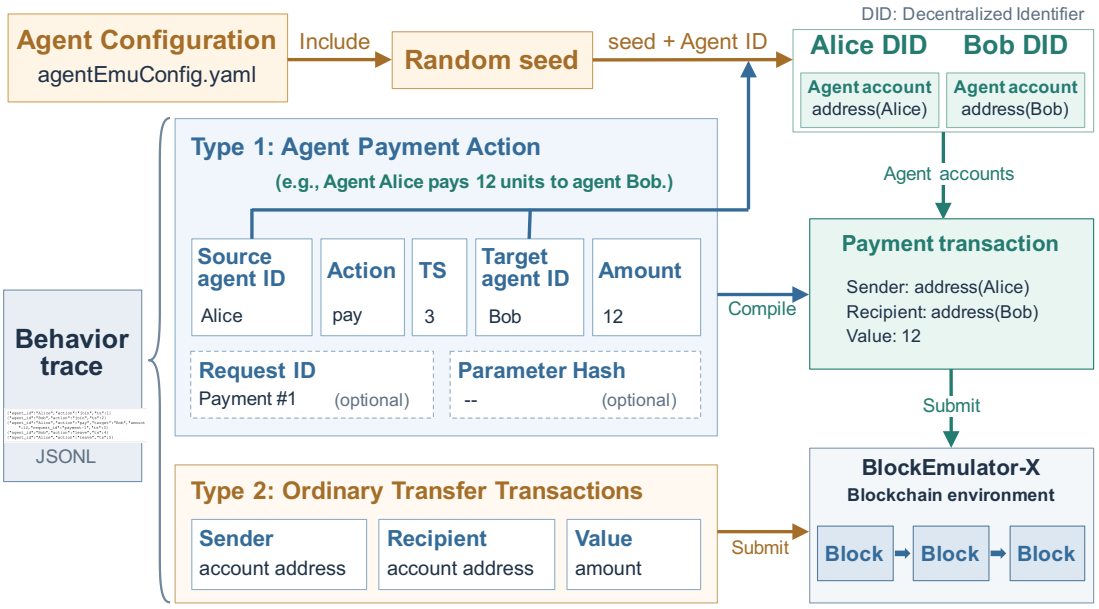


Fig. 2. Illustration of the fields in a behavior trace. An agent action is compiled into a payment transaction that the blockchain can process.

TABLE I
AGENT ACTIONS SUPPORTED BY THE V1.0 TESTBED.

Action	Input behavior	Compiled transaction
`join`	Register or reactivate an agent	DID registration call; no duplicate while already active
`pay`	Pay a positive amount to another active agent	Ordinary balance transfer
`leave`	Deregister an active agent	DID revocation call
Raw transfer	Transfer between ordinary account addresses	Ordinary transfer mixed into the workload

identity state. The agent registry maintains the active status used during trace processing. Payments execute actual balance transfers through the blockchain backend.

Listing 1 gives a small input example of the trace file: Alice and Bob join, Alice pays Bob 12 units, and Bob leaves. This example illustrates how to describe an experiment without manually specifying transaction nonces, data fields, or derived identities.

```
{"agent_id":"Alice","action":"join","ts":1}
{"agent_id":"Bob","action":"join","ts":2}
{"agent_id":"Alice","action":"pay","target":"Bob","amount
    ":12,"request_id":"payment-1","ts":3}
{"agent_id":"Bob","action":"leave","ts":4}
{"agent_id":"Alice","action":"leave","ts":5}
```

Listing 1. Example trace of agent actions, including `join`, `pay`, and `leave`.

C. Orchestration of Agent Actions

The current release uses the *User-specified Original Sequence* policy for orchestrating agent actions. After applying the trace's timestamp and file-order rules, the agent layer introduces no additional transaction reordering, priority rule, or agent-weight algorithm. The trace timestamp determines logical action order rather than transaction arrival or block production time.

Using the same trace, seed, configuration, implementation, and initial registry supports repeatable transaction preparation. Blockchain packing and timing still depend on execution conditions. Researchers can use this default policy as a baseline when implementing research-specific mechanisms.

IV. DESIGN AND IMPLEMENTATION OF AGENTEMULATOR

A. Architecture and Responsibilities

The v1.0 architecture centers on `agentSupervisor`, which combines transaction-dataset construction with experiment orchestration (Fig. 3). It consumes the JSONL trace and two configuration files: `agentEmuConfig.yaml` specifies agent-side experiment parameters, while `config.yaml` supplies the blockchain-side environment inside BlockEmulator-X. The *transaction construction* component inside `agentSupervisor` parses actions, validates behavior and identity state, and generates transaction datasets for blockchains. The *experiment orchestration* component inside `agentSupervisor` reads parameters, configures and launches the blockchain environment, and then manages the experiment running.

BlockEmulator-X provides the blockchain backend for `agentSupervisor`. BlockEmulator-X executes the submitted transactions and supplies execution records. The recording and visualization stage combines compilation artifacts with those results to save experiment data and generate the figure gallery. From the user's perspective, the workflow is to supply a trace and configuration, run the `agentSupervisor` controller, and inspect the saved records and plots. The `agentSupervisor` controller is distinct from the *supervisor* process launched inside the BlockEmulator-X.

Next, we explain 5 functionalities of `agentSupervisor` with reference to Fig. 3.

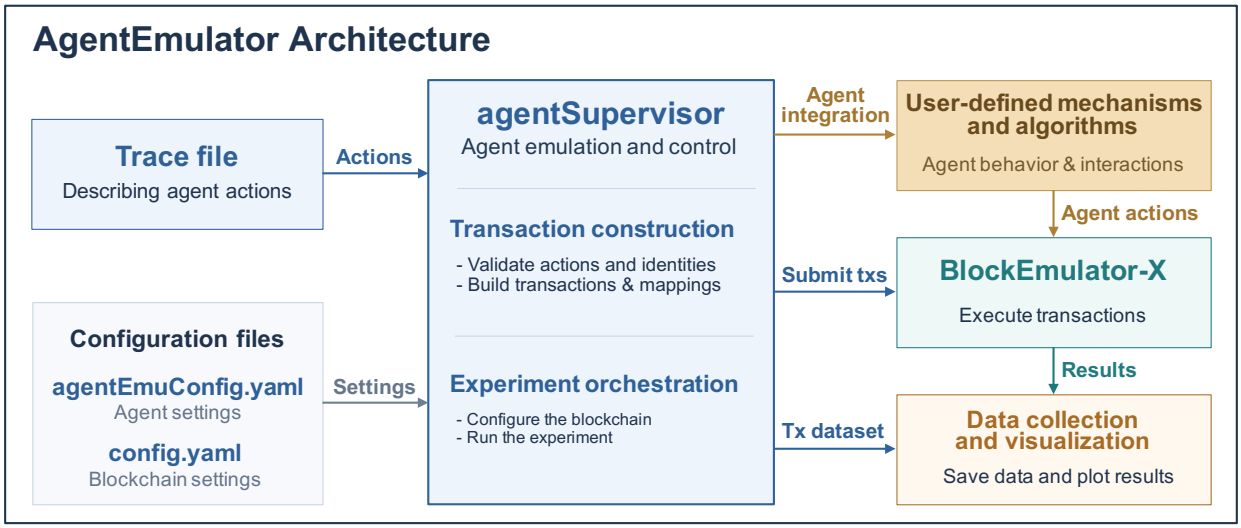


Fig. 3. System architecture of AGENTEMULATOR. The `agentSupervisor` module constructs transactions and mappings, and orchestrates the experiment. BlockEmulator-X executes transactions, and the analysis stage combines the transaction dataset with execution results. The highlighted user-defined mechanisms and algorithms indicate a direction for extension. Their agent-integration and agent-action arrows do not imply a built-in adaptive agent or scheduling implementation in AGENTEMULATOR v1.0.

1) Trace handling: The input uses one JSON object per line. A researcher supplies agent identifiers and intended actions. The compiler supplies the identity and transaction fields. *Agent Lifecycle*[3] checks reject payments involving inactive agents and invalid departures. The input can also include ordinary account-transfer transactions, allowing non-agent traffic to compete with agent transactions for backend processing. This allows experiments to combine agent activity with background blockchain workloads.

2) Transaction compilation: The documented *identity and payment plugins*[4] are `did-simple` and `direct-pay`. Registration and departure actions produce DID registration and revocation call records, respectively. A payment produces an ordinary balance-transfer transaction. The plan stores transaction-level information, including hashes, endpoints, values, nonces, and data. In parallel, the compiler retains the behavioral fields and associated hash lists in the mapping file, and records the action stream in `Agent_Events.csv`[5].

3) User-defined mechanisms and algorithms: The highlighted component in Fig. 3 identifies where research on agent behavior and interactions can connect to the experiment workflow. The user guide[6] suggests transaction reordering, priority rules, and agent weights as possible extensions to the default policy. These examples require an implementation that specifies when it changes actions or transactions and how it preserves lifecycle constraints, nonces, and action mappings. They are not additional built-in v1.0 policies or configuration switches.

For example, a priority-policy experiment could compare two admitted payment workloads under identical chain settings while changing a precisely defined dispatch rule. Such an extension would need to record its policy, preserve the relevant action dependencies, and compare completion and account effects as well as latency. If it changes the action sequence before compilation, lifecycle admission must be rechecked. If it changes the transaction plan afterward, it must still respect transaction dependencies.

4) Backend orchestration: The `agentSupervisor` prepares the backend configuration from the blockchain settings. The configuration specifies storage and log paths, uses `json_source` as the transaction source, and sets `tx_number` to the number of compiled transactions.

The `agentSupervisor` generates the matching port table and launches BlockEmulator-X. The BlockEmulator-X emulates the blockchain, replays the compiled transactions on-chain, and records blocks and transactions. The `agentSupervisor` manages run completion and cleans up processes. Node output appears in the console with node identifiers, and full logs are saved in the run directory.

5) Results and visualization: The experiment directory stores a registry, a summary of each round, and a separate subdirectory for each round. Each round directory contains the transaction plan, action mapping, events, per-agent CSV records, backend configuration, logs, and chain measurements. The default workflow runs one round. The directory structure supports round-based storage, but AGENTEMULATOR v1.0 does not implement adaptive multi-round experiments.

After a successful experiment, the `agentSupervisor` reads the agent records from the highest-numbered round. `agentSupervisor` generates plots with `figs/python_code/plot_agent_balance.py` and creates a static HTML gallery with `figs/python_code/build_fig_html.py`. The gallery shows the data directory, agent count, generation time, and figure count. Its language switch changes the interface text between English and Chinese. Labels within the PNG figures remain unchanged.

A failed experiment skips plotting. A compiler-only run pro-

[3]https://github.com/HuangLab-SYSU/agent-emulator#trace-basics
[4]https://github.com/HuangLab-SYSU/agent-emulator#configuration
[5]https://github.com/HuangLab-SYSU/agent-emulator#results-directory
[6]https://github.com/HuangLab-SYSU/agent-emulator#agentemulator-v10-user-guide

duces no chain-derived agent CSV records for visualization. The `agentSupervisor` also clears previous experiment output and figures before a new run. Results from independent trials must therefore be archived before the next invocation.

B. Configurations of AgentEmulator and Blockchain

Agent configuration in `agentEmuConfig.yaml` controls the trace, seed, result directory, plugin selection, backend enablement, and timeouts. The blockchain configuration in `config.yaml` controls shard and node counts, block interval, logging, and other native parameters. The default workload contains 100 agents and is described as having 10,000 transactions. In the example deployment, the blockchain has four shards with four consensus nodes per shard, and the backend also includes one supervisor process. These settings define a demonstrated deployment configuration, not a capacity limit.

For the illustrated experiment, the guide specifies `static_relay`: account placement is static and cross-shard transactions use relay messages. Blocks are produced at a configured interval of 2000 ms and packed by transaction count, with a maximum of 5000 transactions per block. Block storage uses BoltDB, world-state storage uses Ethereum-style LevelDB, and the Bloom-filter bitmap length is 4096. These values describe the supplied demonstration.

C. Execution Modes

AGENTEMULATOR v1.0 supports two execution modes.

- *Regular mode* runs the blockchain backend and enables subsequent analysis.
- *Compilation-only mode* uses `chain.enabled=false` and does not start the backend. This mode supports trace inspection and reproducibility checks.

Both modes accept a user-defined configuration file. Behavioral experiment settings remain separate from blockchain settings.

D. Running an Experiment

The launch scripts build the project, clear previous outputs, compile the trace, start BlockEmulator-X, and generate plots after a successful run.

- Windows users run `run_agentemu.bat`;
- macOS/Linux users run `bash run_agentemu.sh`.

A configuration filename can be passed to either script. Existing results should be archived before another run because the scripts clear the experiment and figure directories.

Setting `chain.enabled` to `false` provides a compilation-only preview of the transaction dataset. With blockchain execution enabled, the default workflow runs one round, collects the resulting records, and opens the generated HTML gallery.

E. Agent Records and Numeric Settings

Each agent CSV contains seven fields: `block_height`, `tx_hash`, `sender`, `recipient`, `value`, `balance`, and `block_time_ms`. The `agentSupervisor` builds account histories from the shard block stores, applying transaction values and relay-stage debits or credits. The balance field is reconstructed from blocks. Agent records are ordered by block time, with shard, block height, and position within the block used to break ties.

The initial balance is 10^{36} wei per new account, which exceeds the signed 64-bit integer range. Analysis scripts should read values and balances as strings and use arbitrary-precision integers before subtracting the initial balance. Plots display the resulting balance change instead of the large absolute balance.

F. Visualization Pipeline

The plotting script reads the per-agent CSV files and produces four views: final balance changes across agents, distributions over global transaction order, trajectories aligned by normalized local progress, and individual trajectories in groups of five agents. Normalized progress instead scales each agent's own record index to the interval from zero to one. The grouped plots show individual trajectories using each account's local transaction index.

These views serve different inspection tasks: comparing terminal outcomes, observing distribution changes, comparing trajectory shapes, and examining individual agents. Section V presents their use in the documented experiment.

V. DEMONSTRATION OF EXPERIMENTATION

This section demonstrates the experimentation showcase while using AGENTEMULATOR through 100 AI agents. The experimental results show account-balance changes across agents and individual account histories.

TABLE II
EXPERIMENT SETTINGS.

Parameter	Value
Host	Mac mini; Apple M4 Pro; 12 cores (8 performance and 4 efficiency)
Memory and OS	24 GB unified memory; macOS 15.6
Go runtime	1.25.7
Deployment	One host; 4 shards × 4 consensus nodes; 1 backend supervisor
Communication	`direct`; loopback address `127.0.0.1`
Ports	Nodes: 32217–32547; supervisor: 38800
Messaging model	Cross-shard transactions are handled by `static_relay` inside BlockEmulator-X
Block interval	2000 ms
Block capacity	Up to 5000 transactions per block; count-based packing
Storage	BoltDB for blocks; Ethereum-style LevelDB for world state
Bloom filter	Bitmap length of 4096
Random seed	20260903
Workload	100 agents; 10,000 payment transactions
`tx_source_type`	`json_source`. (When `tx_source_type` is set to `json_source`, agentSupervisor reads JSONL files as agent behavior traces.)
Initial balance	10^{36} `wei` per new address
Execution	One round with a fresh backend

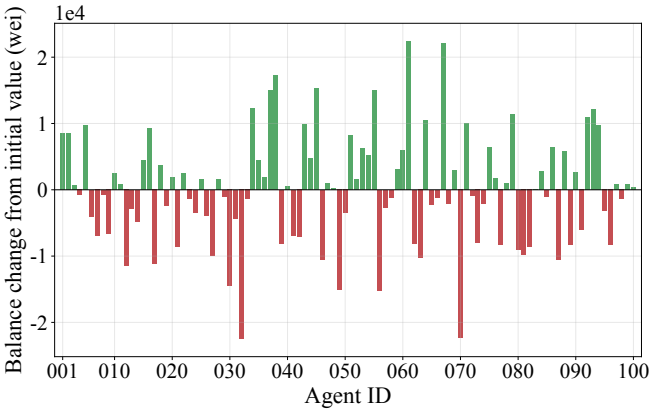
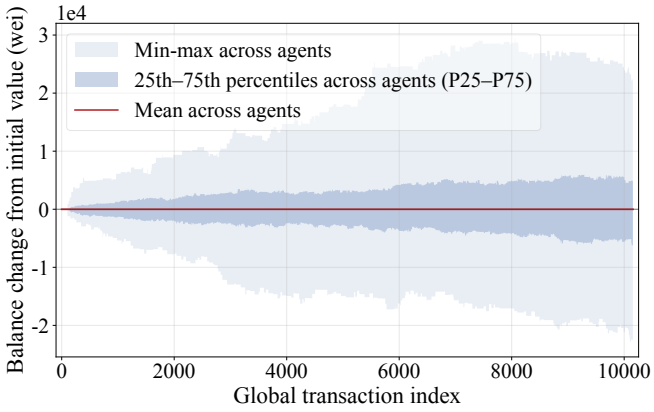
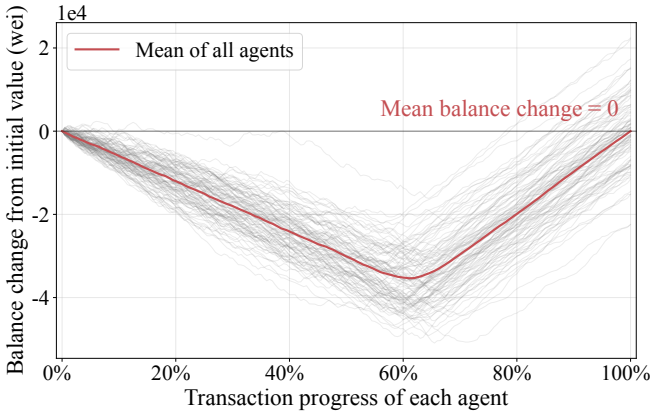


Fig. 4. Final balance changes relative to initial balances for all 100 agents, ordered by agent ID. Each bar represents one agent. Green bars indicate balance increases, and red bars indicate balance decreases.



(a) Balance changes vs. the global transaction order.



(b) Balance changes vs. the progress of each agent's related transactions.

Fig. 5. Account-balance changes from different perspectives. Subfigure (a) shows account-balance changes across the global sequence of all transactions. Subfigure (b) presents balance changes following each agent's finalized transactions, where each gray line represents an agent.

A. Experiment Settings

We conducted the experiment on a Mac mini with an Apple M4 Pro processor and 24 GB of unified memory. The processor had eight performance cores and four efficiency cores. The software environment used macOS 15.6 and Go 1.25.7.

The backend blockchain ran on a single host with separate processes. The deployment included 16 consensus nodes across four shards and one BlockEmulator-X supervisor. The processes communicated in `direct` mode through `127.0.0.1`. Consensus nodes used ports 32217–32547, and the supervisor used port 38800. All processes shared the same host resources.

The workload contained 100 agents and 10,000 transactions to emulate micropayments between agents. The random seed is set to 20260903 to generate agent DIDs. AgentEmulator compiled the input trace and supplied the transaction dataset through `json_source`. The experiment used one round with a fresh blockchain backend. Table II summarizes the environment and execution parameters.

B. Agent's Account Balance

The experiment produced 23 figures for 100 agents. These included three overview figures and twenty figures for groups of five agents. Figures 4–6 show four views of the account-balance records. Each view presents balance changes relative to the initial account balances.

1) Final balance changes: Fig. 4 shows each agent's final account balance minus its initial balance. Final balance changes vary substantially across agents, ranging from approximately $-23{,}000$ wei to $+23{,}000$ wei. Some agents' final balances differ only slightly from their initial balances, whereas others end with substantially higher or lower balances. Each bar shows one agent's final balance change. Positive values mean the agent received more than it sent. Negative values mean it sent more than it received. These values do not measure task performance or profit.

2) Balance changes in global transaction order: Fig. 5(a) shows agents' account-balance changes in reconstructed global transaction order. Each unique transfer transaction contributes a debit and a credit to the displayed accounts. The mean balance change therefore remains at zero.

The distribution widens during most of the experiment. The minimum–maximum envelope shows the full range of balance changes. The interquartile band contains the middle half of the account values. These bands describe variation across agents within one run. They do not represent confidence intervals across repeated experiments.

The zero mean and widening distribution describe different aspects of the workload. Agent payments preserve the aggregate balance while changing its distribution across agents. The aggregate mean alone therefore does not capture individual account outcomes.

3) Balance changes over local progress: Fig. 5(b) aligns account histories by each agent's normalized local progress. The mean falls below zero during the intermediate stages and returns to zero at the endpoint.

The intermediate mean differs from the mean in Fig. 5(a) because the two views select account records differently. Equal local progress can correspond to different transactions for different agents. The selected records therefore do not represent a common point in the global replay.

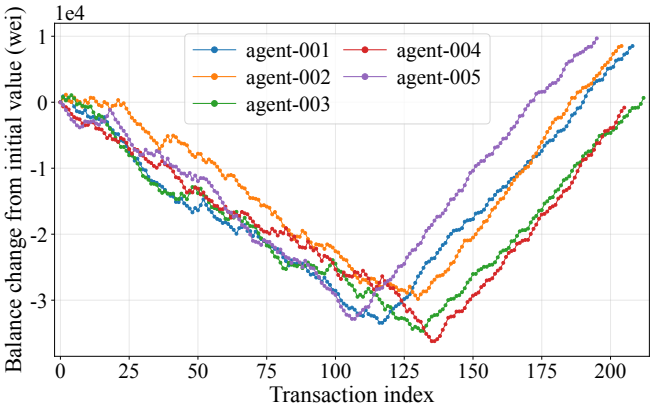


Fig. 6. Balance changes relative to initial balances for agents #001 – #005, ordered by each agent's related transaction index.

This view supports comparison of account trajectories with different numbers of records. The negative intermediate mean does not by itself indicate lost value or relay delay. The horizontal axis represents local progress rather than elapsed time.

4) Individual agent trajectories: Fig. 6 shows the account histories of five agents. Their balance changes follow different paths and reach different final values. Their turning points also occur at different local transaction indices.

The horizontal axis counts records separately for each agent. This view allows researchers to inspect individual histories and compare trajectory shapes. The overview figures place these account-level changes within the distribution across all agents.

C. Evaluation Scope and Further Measurements

The experiment shows how AgentEmulator connects workload execution with account-level analysis. The results cover one workload on a single host. They do not establish performance across independent machines or under larger workloads.

The balance figures use transaction order and local progress as their horizontal axes. These axes do not provide the elapsed time needed to calculate throughput or transaction latency. Quantitative performance evaluation requires separate timing measurements. Resource overhead was not measured in this experiment.

A further study could vary agent count, payment volume, shard count, and background traffic. Candidate settings include 10, 100, and 1000 agents, with 10^3, 10^4, and 10^5 payments. Shard counts could range from 1 to 128. Each comparison would hold the remaining settings fixed.

Workload counts need to distinguish agent actions, ordinary transfer transactions, and compiled transactions. For example, a repeated `join` from an active agent produces no additional registration transaction. Throughput measurements would count completed original transactions over a defined interval. Latency measurements would use transaction creation and final-commit timestamps. Relay stages would remain part of the original transaction.

A direct BlockEmulator-X run could provide a baseline using the same compiled dataset and backend revision. This comparison would help quantify the additional cost of orchestration and result processing. Compilation, startup, blockchain execution, and plotting would require separate measurements. Repeated trials would also need to report variability and include failed or timed-out runs.

VI. Conclusion

This paper presented AgentEmulator, a blockchain-based testbed for research on trustworthy AI agent infrastructure. AgentEmulator converts agent lifecycle and payment actions into a transaction dataset. It runs the dataset through BlockEmulator-X and produces account records, balance plots, and an HTML gallery.

The experiment demonstration involved 100 agents across four blockchain shards. The balance views of all agents showed differences among individual accounts. The reconstructed global view preserved a zero mean while showing changes in the balance distribution. The local views showed how individual account histories developed during execution.

The current experiment evaluation covers workflow execution and balance analysis on a single host. Future experiments will examine throughput, latency, and resource overhead under different workloads and deployments. Future development will also extend agent mechanisms for identity, auditing, settlement, incentives, and governance.

Appendix A
Experiment Reproducibility

A. Software and Execution

AgentEmulator requires Go 1.25 or later and Python 3.8 or later. The plotting scripts use `matplotlib`, `numpy`, and `pandas`. The source code is available at https://github.com/HuangLab-SYSU/agent-emulator. The backend source code is available at https://github.com/HuangLab-SYSU/block-emulator-x. Reproducible runs require fixed revisions of both repositories.

The experiment script is `run_agentemu.bat` on Windows and `run_agentemu.sh` on macOS and Linux. The macOS and Linux command is `bash run_agentemu.sh`. Both scripts accept a configuration filename as an argument.

B. Experiment Records

A reproducible experiment requires the original trace, random seed, transaction policy, and source revisions. The records also need to include dependency versions, execution commands, and machine specifications. The configuration files should include the agent settings, effective backend configuration, and initial-state policy.

The retained outputs should include the registry, transaction plan, action mapping, account records, chain measurements, and logs. The main output paths for a round are shown here.

```
agent_registry.json
rounds_summary.json
round_001/
  agent_transactions.jsonl
  agent_action_txs.jsonl
  Agent_Events.csv
  agents/<agent-id>.csv
  chain/config.yaml
  chain/ip_table.json
  chain/logs/
  chain/results/
    relay_stats_detail_tx_info.csv
    relay_stats_brief_info.csv
```

The experiment script clears previous results before a new run. Each trial requires a separate archive. File digests can help detect changes to the archived inputs and outputs.

C. Functional and Repeatability Checks

Small traces support checks of registration, repeated registration, departure, re-entry, and payments involving inactive agents. Mixed traces can also test agent actions alongside ordinary transfer transactions. The transaction dataset, registry, and execution logs provide records for checking the outcomes. Listing 1 provides an example input.

Compilation-only runs support comparisons of generated artifacts. These comparisons require the same trace, seed, configuration, source revision, and clean initial registry. Compilation-only outputs support plan inspection but contain no blockchain execution results.

Visualization checks can compare the values in the two final-balance subfigures. Both subfigures should contain the same values in different orders. Each local trajectory should also end at the corresponding account's final balance change.

D. Figure Regeneration

Archived account records provide the inputs for figure regeneration. An explicit data directory identifies the selected run. The original integer-valued records and plotting commands allow the displayed balance changes to be checked.

The following commands regenerate the plots and HTML gallery from one archived run.

```
python3 figs/python_code/plot_agent_balance.py \
  --data-dir <archive>/round_001/agents \
  --fig-dir <archive>/figures
python3 figs/python_code/build_fig_html.py \
  --fig-dir <archive>/figures \
  --data-dir <archive>/round_001/agents
```

The gallery builder uses `-fig-dir` to locate the PNG files and save the HTML page. Its `-data-dir` argument supplies the displayed data source and agent count. This argument does not regenerate the figures or verify their source. Both commands therefore need to reference the same archived run. Windows execution uses the corresponding Python command and path syntax.

REFERENCES

[1] J. Liao, Q. Chen, J. Zheng, X. Tang, F. Hu, and H. Huang, "A survey of blockchain transaction processing and scalability: toward a trust infrastructure for ai agents," *Blockchain*, vol. 1, no. 1, p. 0005, 2026. [Online]. Available: https://doi.org/10.55092/blockchain20260005

[2] W. Chen, Z. You, R. Li, Y. Guan, C. Qian, C. Zhao, C. Yang, R. Xie, Z. Liu, and M. Sun, "Internet of agents: Weaving a web of heterogeneous agents for collaborative intelligence," in *International Conference on Learning Representations (ICLR)*, 2025.

[3] Y. Shang, Y. Li, K. Zhao, L. Ma, J. Liu, F. Xu, and Y. Li, "AgentSquare: Automatic LLM agent search in modular design space," in *International Conference on Learning Representations (ICLR)*, 2025.

[4] K. Zhu, H. Du, Z. Hong, X. Yang, S. Guo, Z. Wang, Z. Wang, C. Qian, X. Tang, H. Ji, and J. You, "MultiAgentBench: Evaluating the collaboration and competition of LLM agents," in *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Jul. 2025, pp. 8580–8622.

[5] H. Huang, G. Ye, Q. Yang, Q. Chen, Z. Yin, X. Luo, J. Lin, J. Zheng, T. Li, and Z. Zheng, "BlockEmulator: An emulator enabling to test blockchain sharding protocols," *IEEE Transactions on Services Computing*, vol. 18, no. 2, pp. 690–703, 2025.

[6] T. Schick, J. Dwivedi-Yu, R. Dessi, R. Raileanu, M. Lomeli, E. Hambro, L. Zettlemoyer, N. Cancedda, and T. Scialom, "Toolformer: Language Models Can Teach Themselves to Use Tools," in *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, Eds., vol. 36. Curran Associates, Inc., 2023, pp. 68 539–68 551.

[7] Y. Qin, S. Liang, Y. Ye, K. Zhu, L. Yan, Y. Lu, Y. Lin, X. Cong, X. Tang, B. Qian, S. Zhao, L. Hong, R. Tian, R. Xie, J. Zhou, M. Gerstein, D. Li, Z. Liu, and M. Sun, "ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs," in *International Conference on Learning Representations*, B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun, Eds., vol. 2024, 2024, pp. 9695–9717.

[8] N. Shinn, F. Cassano, A. Gopinath, K. Narasimhan, and S. Yao, "Reflexion: language agents with verbal reinforcement learning," in *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, Eds., vol. 36. Curran Associates, Inc., 2023, pp. 8634–8652.

[9] G. Li, H. Hammoud, H. Itani, D. Khizbullin, and B. Ghanem, "CAMEL: Communicative Agents for "Mind" Exploration of Large Language Model Society," in *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, Eds., vol. 36. Curran Associates, Inc., 2023, pp. 51 991–52 008.

[10] S. Hong, M. Zhuge, J. Chen, X. Zheng, Y. Cheng, J. Wang, C. Zhang, Z. Wang, S. Yau, Z. Lin, L. Zhou, C. Ran, L. Xiao, C. Wu, and J. Schmidhuber, "MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework," in *International Conference on Learning Representations*, B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun, Eds., vol. 2024, 2024, pp. 23 247–23 275.

[11] C. Qian, W. Liu, H. Liu, N. Chen, Y. Dang, J. Li, C. Yang, W. Chen, Y. Su, X. Cong, J. Xu, D. Li, Z. Liu, and M. Sun, "ChatDev: Communicative agents for software development," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, L.-W. Ku, A. Martins, and V. Srikumar, Eds. Bangkok, Thailand: Association for Computational Linguistics, Aug. 2024, pp. 15 174–15 186.

[12] J. Zhang, J. Xiang, Z. Yu, F. Teng, X. Chen, J. Chen, M. Zhuge, X. Cheng, S. Hong, J. Wang, B. Zheng, B. Liu, Y. Luo, and C. Wu, "AFlow: Automating agentic workflow generation," in *International Conference on Learning Representations (ICLR)*, 2025.

[13] X. Liu, H. Yu, H. Zhang, Y. Xu, X. Lei, H. Lai, Y. Gu, H. Ding, K. Men, K. Yang, S. Zhang, X. Deng, A. Zeng, Z. Du, C. Zhang, S. Shen, T. Zhang, Y. Su, H. Sun, M. Huang, Y. Dong, and J. Tang, "AgentBench: Evaluating LLMs as Agents," in *International Conference on Learning Representations*, B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun, Eds., vol. 2024, 2024, pp. 52 989–53 046.

[14] S. Zhou, F. F. Xu, H. Zhu, X. Zhou, R. Lo, A. Sridhar, X. Cheng, T. Ou, Y. Bisk, D. Fried, U. Alon, and G. Neubig, "WebArena: A Realistic Web Environment for Building Autonomous Agents," in *International Conference on Learning Representations*, B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun, Eds., vol. 2024, 2024, pp. 15 585–15 606.

[15] T. Xie, D. Zhang, J. Chen, X. Li, S. Zhao, R. Cao, T. J. Hua, Z. Cheng, D. Shin, F. Lei, Y. Liu, Y. Xu, S. Zhou, S. Savarese, C. Xiong, V. Zhong, and T. Yu, "OSWorld: Benchmarking Multimodal Agents for Open-Ended Tasks in Real Computer Environments," in *Advances in Neural Information Processing Systems*, A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, Eds., vol. 37. Curran Associates, Inc., 2024, pp. 52 040–52 094.

[16] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, "SWE-bench: Can Language Models Resolve Real-world Github Issues?" in *International Conference on Learning Representations*, B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun, Eds., vol. 2024, 2024, pp. 54 107–54 157.
[17] E. Debenedetti, J. Zhang, M. Balunovic, L. Beurer-Kellner, M. Fischer, and F. Tramèr, "AgentDojo: A Dynamic Environment to Evaluate Prompt Injection Attacks and Defenses for LLM Agents," in *Advances in Neural Information Processing Systems*, A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, Eds., vol. 37. Curran Associates, Inc., 2024, pp. 82 895–82 920.
[18] G. Danezis, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman, "Narwhal and Tusk: A DAG-based mempool and efficient BFT consensus," in *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys)*. Association for Computing Machinery, 2022, pp. 34–50.
[19] A. Spiegelman, N. Giridharan, A. Sonnino, and L. Kokoris-Kogias, "Bullshark: DAG BFT protocols made practical," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. Association for Computing Machinery, 2022, pp. 2705–2718.
[20] K. Babel, A. Chursin, G. Danezis, A. Kichidis, L. Kokoris-Kogias, A. Koshy, A. Sonnino, and M. Tian, "Mysticeti: Reaching the latency limits with uncertified DAGs," in *Network and Distributed System Security (NDSS) Symposium*. Internet Society, 2025.
[21] Y. Liu, A. Liu, Y. Lu, Z. Pan, Y. Li, J. Liu, S. Bian, and M. Conti, "Kronos: A secure and generic sharding blockchain consensus with optimized overhead," in *Network and Distributed System Security (NDSS) Symposium*. Internet Society, 2025.
[22] Y. Liu, B. Zhao, Y. Hu, H. Tan, F. Ran, A. Liu, Z. Pan, Y. Lu, S. Bian, J. Liu, and Z. Guan, "Logos: Robust sharding blockchain with fast processing and optimal cross-shard overhead," in *35th USENIX Security Symposium (USENIX Security 26)*. USENIX Association, Aug. 2026, pp. 1047–1066.



Jian Zheng is a student pursuing his Ph.D. degree at Sun Yat-sen University, China. His research interests primarily focus on blockchain and DeFi protocols.



Jianbo Xiong is a student pursuing his M.Eng. degree at the School of Software Engineering, Sun Yat-sen University. His research interests mainly include blockchain.



Feihong Hu is pursuing a Master's degree at the School of Software Engineering, Sun Yat-sen University. The main research area is blockchain technology.



Huawei Huang (SM'22) is a professor at the School of Software Engineering, Sun Yat-Sen University, China. He received his Ph.D. in Computer Science and Engineering from the University of Aizu (Japan). His research interests include blockchain, distributed systems/protocols, and computer networking. He has served as a guest editor for multiple special issues on blockchain in IEEE Journal on Selected Areas in Communications (JSAC), IEEE OJ-CS, IET Blockchain, etc. He also served as an organizing chair for conferences IEEE ICC, CBDCom, VTC Fall, BlockSys, and IEEE GBC. He is also an associate editor of multiple journals, including IEEE TNSE, ELSP Blockchain, etc.