Software Engineering Agent Economics: A Blockchain Software Development Kit for Economic Network Simulations

Mohamed Ahmed Fouad Aly Mohamed Ibrahim

Universidade Federal de Uberlândia
Faculdade de Computação
Programa de Pós-Graduação em Ciência da Computação

Uberlândia
2025

Mohamed Ahmed Fouad Aly Mohamed Ibrahim

Software Engineering Agent Economics: A Blockchain Software Development Kit for Economic Network Simulations

Dissertação de mestrado apresentada ao Programa de Pós-graduação da Faculdade de Computação da Universidade Federal de Uberlândia como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Área de concentração: Computer Science

Advisor: Prof. Dr. Marcelo de Almeida Maia

Uberlândia
2025

Ficha Catalográfica Online do Sistema de Bibliotecas da UFU
com dados informados pelo(a) próprio(a) autor(a).

I14
2025

Ibrahim, Mohamed Ahmed Fouad Aly Mohamed, 1986-
Software Engineering Agent Economics: A Blockchain Software Development Kit for Economic Network Simulations [recurso eletrônico] / Mohamed Ahmed Fouad Aly Mohamed Ibrahim. - 2025.

Orientador: Marcelo Maia.
Dissertação (Mestrado) - Universidade Federal de Uberlândia, Pós-graduação em Ciência da Computação.
Modo de acesso: Internet.
DOI http://doi.org/10.14393/ufu.di.2025.712
Inclui bibliografia.
Inclui ilustrações.

1. Computação. I. Maia, Marcelo,1969-, (Orient.). II. Universidade Federal de Uberlândia. Pós-graduação em Ciência da Computação. III. Título.

CDU: 681.3

Bibliotecários responsáveis pela estrutura de acordo com o AACR2:
Gizele Cristine Nunes do Couto - CRB6/2091
Nelson Marcos Ferreira - CRB6/3074

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
Coordenação do Programa de Pós-Graduação em Computação
Av. João Naves de Ávila, 2121, Bloco 1A, Sala 243 - Bairro Santa Mônica, Uberlândia-MG, CEP 38400-902
Telefone: (34) 3239-4470 - www.ppgco.facom.ufu.br - cpgfacom@ufu.br

ATA DE DEFESA - PÓS-GRADUAÇÃO

Programa de Pós-Graduação em:	Ciência da Computação				
Defesa de:	Dissertação, 50/2025, PPGCO				
Data:	15 de Dezembro de 2025	Hora de início:	14:00	Hora de encerramento:	16:10
Matrícula do Discente:	12222CCP017				
Nome do Discente:	Mohamed Ahmed Fouad Aly Mohamed Ibrahim				
Título do Trabalho:	Software Engineering Agent Economics: A Blockchain Software Development Kit for Economic Network Simulations				
Área de concentração:	Ciência da Computação				
Linha de pesquisa:	Engenharia de Software				
Projeto de Pesquisa de vinculação:	Parece funcionar, mas... Qualidade do Teste de Software na era dos Modelos de Linguagem.				

Reuniu-se por videoconferência, a Banca Examinadora, designada pelo Colegiado do Programa de Pós-graduação em Ciência da Computação, assim composta: Professores Doutores: Fabiano Azevedo Dorça - FACOM/UFU, Diego Elias Damasceno Costa - CSSE/Concordia University e Marcelo de Almeida Maia - FACOM/UFU, orientador do(a) candidato(a).

Os examinadores participaram desde as seguintes localidades: Diego Elias Damasceno Costa - Montreal/Canadá e os outros membros da banca da cidade de Uberlândia/MG. O aluno Mohamed Ahmed Fouad Aly Mohamed Ibrahim participou de Brasília/DF.

Iniciando os trabalhos o presidente da mesa, Prof. Dr. Marcelo de Almeida Maia, apresentou a Comissão Examinadora e o(á) candidato(a), agradeceu a presença do público, e concedeu ao(á) Discente a palavra para a exposição do seu trabalho

A seguir o senhor presidente concedeu a palavra, pela ordem sucessivamente, aos examinadores, que passaram a arguir o(á) candidato(a). Ultimada a arguição, que se desenvolveu dentro dos termos regimentais, a Banca, em sessão secreta, atribuiu o resultado final, considerando o candidato(a):

Aprovado

Esta defesa faz parte dos requisitos necessários à obtenção do título de Mestre.

O competente diploma será expedido após cumprimento dos demais requisitos, conforme as normas do Programa, a legislação pertinente e a regulamentação interna da UFU.

Nada mais havendo a tratar foram encerrados os trabalhos. Foi lavrada a presente ata que após lida e achada conforme foi assinada pela Banca Examinadora.

Referência: Processo nº 23117.087259/2025-10 SEI nº 6923477

To the pursuit of peer-to-peer SWE-Agent economics.
To the opportunities ahead.

Acknowledgments

I thank the anonymous reviewers of an earlier, more primitive version of this work, who encouraged me to continue developing it and collectively described it as *“an innovative idea,” “a novel setting for studying economic interactions between SWE-Agents,”* and *“a thought-provoking and timely concept that could significantly impact the field of software engineering.”* I am especially grateful to the second reviewer, whose critical feedback on the design and long-term viability of the approach helped sharpen the direction of this dissertation. I am also deeply indebted to Prof. Marcelo Maia for giving me the opportunity to pursue that challenge and for his guidance and unwavering support throughout this journey.

Epigraph

"The real genius of AI isn't in thinking like us, but in showing us new ways to think."
— Yann LeCun

List of Acronyms

IPW	Intelligence Per Watt
ABM	Agent-Based Modeling
ACE	Agent-Based Computational Economics
ACI	Agentic Capability Interface
AI	Artificial Intelligence
API	Application Programming Interface
BDI	Belief–Desire–Intention
BT	Behavior Tree
CI/CD	Continuous Integration and Continuous Delivery
EGTA	Empirical Game-Theoretic Analysis
EVM	Ethereum Virtual Machine
FPS	First-Pass Success
FSM	Finite-State Machine
HHI	Herfindahl–Hirschman Index
IBC	Inter-Blockchain Communication
IC	Incentive Compatibility
IDE	Integrated Development Environment
IR	Individual Rationality
ISE	Intelligent Software Engineering
JSON	JavaScript Object Notation
LLM	Large Language Model
MEV	Maximal Extractable Value

ReAct	Reason+Act (interleaved planning and tool use)
SDK	Software Development Kit
SDGE	Synthetic Data Generation Engine
SEE	Software Engineering Economics
SWE-Agent	Software-Engineering Agent
SWEChain	Software-Engineering Chain (application-specific blockchain)
SWEChain-SDK	SDK for SWE-Agent Economic Simulations on SWEChain
VVUQ	Verification, Validation, and Uncertainty Quantification

Abstract

AI agents' strategic interactions in economic settings—a key line of inquiry in AI-agent economics—have attracted increasing attention amid recent advances in generative AI. We investigate blockchain-based SWE-Agent outsourcing markets, which link intelligent software engineering and software-engineering economics through programmable, transparent, and auditable mechanisms that centralized platforms rarely expose. We evaluate whether a blockchain-native SDK for economic network simulations can support controlled, paired baseline–intervention experiments under fixed conditions in decentralized SWE-Agent outsourcing markets.

We present *SWEChain-SDK*, a configurable, extensible blockchain-native SDK that provides a local sandbox chain for SWE-Agent software outsourcing auctions and event-complete logging of bids, allocations, payments, and artifacts, together with configuration metadata sufficient to support re-runnable experiments under comparable conditions. Using *SWEChain-SDK*, we conduct paired simulation experiments that vary one policy dimension at a time—(A) comparative advantage via agent specialization, (B) competitive bidding via bidders' price-signal utilization, and (C) principal-weighted selection via quality-signal utilization—while holding rules, datasets, random seeds, time base, and the agent pool fixed. We report the observed paired differences in task completion, outsourcing cost, and revenue concentration, making trade-offs explicit. The results show that *SWEChain-SDK* supports controlled baseline–intervention experiments on decentralized SWE-Agent outsourcing markets.

We contribute (i) a conceptual and terminological framing of SWE-Agent Economics as a domain linking intelligent software engineering and software-engineering economics; (ii) *SWEChain-SDK*, a blockchain-native SDK for controlled studies of decentralized, auction-based SWE-Agent outsourcing markets under comparable conditions; and (iii) a public, evaluation-ready release of the SDK—including simulation code, configuration manifests, and event logs—for further empirical studies of decentralized SWE-Agent outsourcing markets.

Keywords: Intelligent Software Engineering; Software Engineering Economics; Software Engineering Agent Economics; Blockchain.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

This dissertation investigates how autonomous Software-Engineering Agents (SWE-Agents) can be studied rigorously when they interact as economic decision-makers in decentralized outsourcing markets implemented on blockchain-style infrastructure. It follows the emerging line of *AI-agent economics*, which analyzes autonomous Artificial Intelligence (AI) agents' objectives, information, and strategic behavior in economic settings, and instantiates this agenda in software engineering by focusing on blockchain-based SWE-Agent outsourcing markets. Rather than evaluating SWE-Agents as isolated predictors, the dissertation treats them as participants in an operational environment where explicit rules for task allocation, pricing, information disclosure, and winner selection shape outcomes such as task completion, outsourcing cost, and the concentration of revenue across agents. To make such studies feasible and credible, the dissertation designs, develops, and empirically evaluates *SDK for SWE-Agent Economic Simulations on SWEChain (SWEChain-SDK)*, a configurable, extensible blockchain-native Software Development Kit (SDK) that provides a local sandbox chain for SWE-Agent outsourcing auctions and supports controlled economic network simulations in which policies can be varied while other experimental conditions remain fixed and runs can be re-executed under comparable conditions.

We evaluate whether such a blockchain-native SDK for economic network simulations can support controlled, paired baseline–intervention experiments under fixed conditions in decentralized SWE-Agent outsourcing markets. Concretely, we use *SWEChain-SDK* to run paired simulation experiments that vary one policy dimension at a time—(A) comparative advantage via agent specialization, (B) competitive bidding via bidders' *price-signal utilization*, and (C) principal-weighted selection via *quality-signal utilization*—and we report the observed paired differences in task completion, outsourcing cost, and revenue concentration, making trade-offs explicit.

This chapter explains the motivation for the work, states the problem addressed, lays out the objectives and research questions, sketches the methodological approach, clarifies scope and limitations, highlights the expected significance and contributions, and outlines the structure of the dissertation.

1.1 Context and Motivation

In current practice, most capable SWE-Agents are not deployed as standalone systems but as services and assistants embedded in centralized platforms. Code and collaboration are concentrated on GitHub, which now hosts both the dominant social coding infrastructure and AI coding assistants such as GitHub Copilot that run inside mainstream Integrated Development Environments (IDEs) like Visual Studio Code and Visual Studio (GitHub, 2023; Microsoft, 2023). The underlying models are typically provided as cloud Application Programming Interfaces (APIs) by a small number of vendors, including OpenAI's GPT-4 (OPENAI, 2023), Anthropic's Claude 3 family (Anthropic, 2024), and code-specialized models such as DeepSeek-Coder (GUO et al., 2024). AI-first editors such as Cursor integrate these models directly into the development environment and connect to centralized repositories over Git or GitHub, further tying agent behavior to the policies and infrastructure of a few platform providers (Anysphere, 2024; PENG et al., 2023). As a result, both the model capabilities and the operational context of many SWE-Agents are effectively mediated by centralized code-hosting and model-serving platforms.

Recent advances in generative Large Language Models (LLMs) have renewed interest in *AI-agent economics*, which studies autonomous AI agents as economic decision-makers and analyzes how their objectives, information, and strategic behavior shape outcomes in markets and related economic environments. Within software engineering, this connects directly to SWE-Agents: agents that plan, implement, and maintain software artifacts, increasingly with LLMs embedded as components for perception, code synthesis, refactoring, and explanation. At the same time, recent work on the efficiency of local AI inference makes the underlying infrastructure constraints more concrete. Saad-Falcon et al. (SAAD-FALCON et al., 2025) introduce Intelligence Per Watt (IPW) as task accuracy per unit of power and conduct a large-scale study across more than twenty local language models, eight hardware accelerators, and one million single-turn chat and reasoning queries. They show that, as of 2025, small local models with at most 20B active parameters can correctly answer 88.7% of such queries, that IPW has improved 5.3× from 2023 to 2025 through combined model and hardware advances, and that hybrid local–cloud routing can cut energy, compute, and monetary cost by roughly 60–80% while maintaining output quality. Taken together, these developments suggest that economically meaningful LLMs workloads are increasingly serviceable by programmable local and hybrid infrastructures, and that the economic behavior of SWE-Agents within these infrastructures is an object of study in its own right.

Autonomous SWE-Agents couple LLMs with developer tools to plan tasks, write and modify code, run builds and tests, and verify results. Unlike single-shot code completion systems, these agents operate as multi-step workflows: they decompose problems into sub-tasks, call tools such as compilers and test runners, examine intermediate outputs, and iterate until a stopping condition—often an acceptance test or reviewer criterion—is satisfied. In many frameworks, agents maintain working memory across steps and can collaborate via explicit roles, for example dividing responsibilities between a planner, an implementer, and a reviewer. As these agents become more capable and more common, they no longer just emit code in response to prompts; they make sequential decisions, adjust to feedback, and compete for limited tasks.

When many such agents operate concurrently on a stream of software tasks, their interactions resemble an economic system. Each agent consumes priced resources such as model inference tokens, CPU and GPU time, and network bandwidth; each competes for tasks that vary by domain, complexity, and risk; and each settles outcomes against verifiable criteria such as test suites, static analysis checks, or human approval. In this setting, rules governing task allocation and payment become first-order determinants of behavior. A canonical rule used in procurement is the reverse auction, where a task owner solicits bids and the lowest acceptable bid wins; a pay-as-bid auction pays the winner exactly the submitted bid rather than a uniform market price, which is straightforward to implement and audit but can encourage strategic bidding below true cost. Practical deployments also decide which bids are admissible by enforcing thresholds such as required tags or prior success rates, set maximum acceptable prices via reserves, choose deterministic tie-breaking when offers are equivalent, and decide which information is visible during the process, for example whether the current best admissible bid is revealed. The combination of these choices determines who works on what, at what price, how quickly the process converges, and how rewards are distributed across agents.

Studying these effects credibly requires an environment where the rules are programmable, the events are observable, and the runs are reproducible. By programmable, we mean that a researcher can implement and swap allocation and pricing rules, change timing parameters, and adjust what information is revealed without rewriting large parts of the system. By observable, we mean that every important state change—task creation, bid submission, updates to the standing best admissible bid, winner selection, and settlement—is recorded with stable identifiers and timestamps so that the entire process can be reconstructed and audited. By reproducible, we mean that experiments can be re-run with shared random seeds, stable ordering rules, and event-complete logs, so that independent groups can obtain comparable results from the same experimental setup.

In this dissertation, we use the term *comparable conditions* to describe paired runs that share the same controlled elements of the experimental setup—including auction rules and parameters, task datasets and their ordering, the SWE-Agent pool and configurations, and the random seeds that govern task assignment and agent decisions—while differing only in the specific policy under study. Comparable conditions do not require that all low-level sources of variation (such as nondeterminism inside LLMs) be eliminated; instead, they require that the experimental design and configuration be aligned closely enough that observed differences in outcomes can be meaningfully attributed to the policy change rather than to background drift or uncontrolled changes in the environment.

Similarly, we use the term *event-complete logging* to refer to logs that capture every economically relevant state change in the marketplace at the level of abstraction used in this dissertation. Event-complete logs contain, for each trial, all task-creation events, all bids and their attributes, all updates to the standing best admissible bid, all winner selections, all settlement events, and pointers to the resulting artifacts. This does not mean that every low-level protocol message or internal implementation detail of the chain is persisted in the artifact bundle; rather, it means that the logs are sufficiently complete to reconstruct the sequence of market interactions, recompute the outcome measures reported in later chapters, and support additional analyses under the same experimental configuration.

A large share of contemporary software-engineering activity, however, runs through a small set of centralized platforms. GitHub is now the dominant social coding site and a primary data source for mining software repositories, with work such as Kalliamvakou et al. and Cosentino et al. documenting both its central role and systematic biases in the data it exposes (KALLIAMVAKOU et al., 2014; COSENTINO; IZQUIERDO; CABOT, 2017). Online labor platforms such as Upwork are canonical examples of digital labor markets for software and related work, and have been extensively studied by the International Labour Organization (ILO) and others as intermediaries that control access to tasks, pay, and reputation (BERG et al., 2018; GRAY; SURI, 2019; AZEVEDO; MENDONÇA, 2023). Proprietary LLM services accessed via APIs (for example, GPT-4–style foundation models) are increasingly embedded in development workflows, but are typically offered as "limited-access" black boxes whose architectures, training data, and deployment details are only partially disclosed (BOMMASANI et al., 2021; BOMMASANI et al., 2024). In parallel, peer-to-peer code collaboration systems such as Radicle, open-access LLMs like BLOOM, and blockchain-based work platforms such as Gitcoin illustrate alternative designs that emphasize open artifacts, protocol-level coordination, and distributed control rather than reliance on a single hosting provider (Radicle Team, 2021; SCAO et al., 2022; Gitcoin, 2022).

Centralized services shape what information is visible about code, work, and model behavior, and to whom. On GitHub, for example, researchers have shown that public repository and activity data are rich but systematically biased: forks, mirrors, inactive projects, and non-standard workflows can all confound naive interpretations, highlighting that the platform operator has a more complete view of development than external analysts relying on the public interface (KALLIAMVAKOU et al., 2014; COSENTINO; IZQUIERDO; CABOT, 2017). The ILO's study of digital labor platforms similarly finds that workers on freelancing and microtask platforms often lack transparency about how tasks are matched, evaluated, and paid, with key signals and algorithms controlled by the platform (BERG et al., 2018). Ethnographic work such as Gray and Suri's *Ghost*

Work reinforces this picture of informational asymmetry in platform-mediated labor (GRAY; SURI, 2019), while systematic reviews of algorithmic management on digital labor platforms highlight the role of non-transparent, data-driven mechanisms in allocating work and evaluating performance (AZEVEDO; MENDONÇA, 2023). In the LLM domain, surveys of foundation models and the Foundation Model Transparency Index describe leading proprietary models as highly opaque, with limited disclosure about training data and deployment practices (BOMMASANI et al., 2021; BOMMASANI et al., 2024), and Casper et al. argue that black-box access alone is insufficient for rigorous audits (CASPER et al., 2024). More broadly, recent work on data curation in machine learning, such as Dohmatob et al.'s theory of when smaller, curated datasets can outperform using "all available data", underscores that selection and filtering decisions critically shape model capabilities (DOHMATOB; PEZESHKI; ASKARI-HEMMAT, 2025). Open and decentralized alternatives do not automatically remove these asymmetries, but they change the technical baseline: Radicle's peer-to-peer "sovereign forge" gives each participant a full local copy of repository history, and open-access models like BLOOM release architectures and weights, making it technically possible to pin the artifacts that define an agent's capabilities and the curated data it is trained on (Radicle Team, 2021; SCAO et al., 2022).

GitHub, digital labor platforms, and proprietary LLM APIs also function as critical infrastructure, so their failures or changes can have wide impact. GitHub's centrality to modern DevOps pipelines is reflected both in industry practice and in its use as a primary data source for mining software repositories; as Kalliamvakou et al. emphasize, this central role makes its outages and policy changes consequential for many projects at once (KALLIAMVAKOU et al., 2014; COSENTINO; IZQUIERDO; CABOT, 2017). The ILO documents how digital labor platforms centralize task posting, matching, and payment, such that interruptions or policy shifts directly affect the incomes and working conditions of workers (BERG et al., 2018). In the LLM space, work on foundation models and transparency stresses that many prominent models are only available as managed cloud services, leaving users with little control over capacity, latency, or upgrade schedules (BOMMASANI et al., 2021; BOMMASANI et al., 2024). Proposals such as Starcloud's orbital data centers, which advocate moving large-scale AI training infrastructure off-planet to relieve terrestrial energy and siting constraints, underline how tightly current AI workloads are coupled to a small number of centralized infrastructure providers (FEILDEN; OLTEAN; JOHNSTON, 2024). By contrast, Radicle's local-first, peer-to-peer architecture replicates repositories across nodes rather than a single hosting site (Radicle Team, 2021), open-weight models such as BLOOM can be deployed on infrastructure controlled by the experimenter (SCAO et al., 2022), and platforms like Gitcoin implement task posting and payment as transparent smart contracts on public blockchains (Gitcoin, 2022). These designs are not yet as mature or widely adopted as their centralized counterparts, but they demonstrate that code, compute, and work allocation can be implemented as explicit protocols with fully visible logs—properties that are attractive for controlled SWE-Agent experiments.

Another core issue is that centralized platforms concentrate governance power over participants through terms of service, access control, and policy decisions. The ILO's report and related work on digital labor emphasize that platform operators set and change rules on pay, rating, and task access, with workers typically having limited influence or visibility into these processes (BERG et al., 2018; GRAY; SURI, 2019; AZEVEDO; MENDONÇA, 2023). In the foundation-model ecosystem, Bommasani et al. and the Foundation Model Transparency Index highlight that developers of proprietary models make key choices about release, safety constraints, and disclosure unilaterally, with downstream users having little say and limited information about governance practices (BOMMASANI et al., 2021; BOMMASANI et al., 2024). Casper et al. explicitly frame the level of access granted for audits—black-box versus white- or "outside-the-box"—as a central governance decision (CASPER et al., 2024). Decentralized projects explore different governance arrangements: Radicle explicitly states that there is "no single entity controlling the network", relying on cryptographic identities and project-level control (Radicle Team, 2021), while Gitcoin's grants and bounty mechanisms are governed through community processes and on-chain voting (Gitcoin, 2022). These alternatives are not complete solutions, but they show that governance rules can be made explicit, versioned, and auditable.

Taken together, these knowledge, infrastructure, and governance constraints make it important to study alternatives to real-world centralized platforms as experimental settings for *SWE-Agent Economics*. When data access paths, infrastructure conditions, and platform policies can change without being fully logged, it becomes harder to attribute observed differences in outcomes to a specific policy intervention rather than to background drift. One such alternative is to use blockchain-based mechanisms as an experimental testbed for research. Blockchains can encode market rules as on-chain code, record bids and allocations as on-chain events, and expose an auditable history of interactions. For experimentation and simulation in this setting, researchers benefit from a local, configurable, sandboxed environment that behaves like a blockchain-based market but can be instantiated, configured, and reset as needed, and that produces event-complete logs suitable for analysis and scripted re-runs under comparable conditions. This motivates the development of a blockchain-native SDK tailored to economic network simulations in decentralized SWE-Agent outsourcing markets.

1.2 Problem Statement

This dissertation addresses a practical gap in how we study SWE-Agents that interact through decentralized outsourcing markets under explicit economic rules. The underlying aim is to understand how economic mechanisms such as allocation, pricing, information disclosure, and winner selection shape outcomes such as task completion, outsourcing cost, and the concentration of revenue across agents. To obtain credible evidence about these effects, researchers need tooling that supports controlled experiments in which mechanisms can be varied under fixed experimental conditions and in which the resulting interactions are recorded in sufficient detail to be analyzed and audited.

In practice, this calls for research tooling that can run paired baseline–intervention experiments where only a single policy element is changed, and that records agent interactions and market outcomes in a form suitable for analysis and scripted re-runs, with shared random seeds and configuration metadata so that runs can be re-executed under comparable conditions. To the best of our knowledge, no open, blockchain-native, local-first SDK currently offers this combination of capabilities for SWE-Agent experimentation in decentralized, auction-based outsourcing markets at the intersection of intelligent software engineering and software-engineering economics.

The problem this dissertation tackles is therefore to design, implement, and empirically evaluate such tooling, *SWEChain-SDK*, so that controlled, paired baseline–intervention experiments under fixed conditions become practically achievable for the study of SWE-Agent economic mechanisms through economic network simulations.

1.3 Objectives

The overall objective of this dissertation is to investigate whether and how a blockchain-native SDK for economic network simulations can support controlled, paired baseline–intervention experiments under fixed conditions in decentralized SWE-Agent outsourcing markets, in line with the broader agenda of SWE-Agent Economics. More concretely, the dissertation seeks to shorten the path from a well-posed policy question—for example, whether revealing the current best admissible bid affects price formation and convergence—to an executable experiment with auditable outcomes that can be re-run under comparable conditions.

Concretely, the dissertation aims to:

(i) provide a conceptual and terminological framing of *SWE-Agent Economics* as the study of SWE-Agents in economic settings, with a focus on auction-based outsourcing markets on blockchain-style infrastructures;

(ii) design, implement and evaluate *SWEChain-SDK*, a configurable, extensible blockchain-native SDK that provides a local sandbox chain for SWE-Agent outsourcing auctions and event-complete logging of bids, allocations, payments, and artifacts to support economic network simulations;

(iii) develop an experimental setup for paired baseline–intervention simulations in which a single policy variable—such as comparative advantage via agent specialization, competitive bidding via bidders' price-signal utilization, or principal-weighted selection via quality-signal utilization—is varied while rules, datasets, random seeds, time base, and the agent pool are held constant;

(iv) use this setup to conduct and analyze simulation experiments that assess how these policy dimensions affect task completion, outsourcing cost, and the concentration of revenue across agents, reporting paired differences to make trade-offs explicit; and

(v) package and release *SWEChain-SDK*, together with simulation code and artifacts, in an evaluation-ready form that supports scripted re-runs and extension of the experiments reported in this dissertation under comparable conditions.

1.4 Approach and Methodology

The inquiry is organized around a single, primary research question about the SDK as an enabler of credible, controlled experimentation:

> *Can a blockchain-native SDK for economic network simulations support controlled, paired baseline–intervention experiments under fixed conditions in decentralized SWE-Agent outsourcing markets?*

In this work, a positive answer to this question means that the SDK lets researchers configure policy rules, run paired trials under fixed conditions, and obtain event-complete logs. Here, we use *event-complete* to mean logs that are intended to capture the state-changing events relevant to the experiment in enough detail that the reported metrics can be recomputed from the logs alone. The overall design aims to support independent re-runs and modest policy variations under comparable conditions. To make this question concrete and empirically tractable, the dissertation studies three policy dimensions in decentralized SWE-Agent outsourcing markets, each realized as a paired baseline–intervention simulation experiment:

❏ **Comparative advantage via agent specialization (Experiment A):** introducing lightweight agent-side specialization—via private tags and small cost multipliers—to test how specialization affects routing, task completion, and revenue concentration;

❏ **Competitive bidding via bidders' price-signal utilization (Experiment B):** varying the information available to bidders and the prompts that reference current price signals—for example, whether the current best admissible bid is revealed—to study how price-signal utilization affects task completion, realized outsourcing cost, the concentration of revenue across agents, and observed price levels and dispersion; and

- ❑ **Principal-weighted selection via quality-signal utilization (Experiment C):** replacing price-only selection with a transparent price–quality scoring rule to assess how quality-signal utilization affects realized software quality, total outsourcing cost, and the concentration of revenue.

We employ a simulation-based experimental methodology in empirical software engineering. We develop *SWEChain-SDK*, a blockchain-native SDK for economic network simulations, and use this SDK as an experimental platform to run agent-based simulations of decentralized SWE-Agent outsourcing markets under controlled baseline–intervention conditions.

At the system level, *SWEChain-SDK* is implemented as an application-specific chain that encodes procurement auctions with pay-as-bid pricing. SWE-Agents interact with this chain by submitting bids, receiving allocations, and settling tasks. The chain and associated runtimes emit event-complete logs for key state changes, including task creation, bid submission, updates to the standing best admissible bid, winner selection, and settlement events. At the experimental level, the empirical part of the dissertation relies on controlled, paired baseline–intervention simulations built on *SWEChain-SDK*. Rather than attempting to optimize a single mechanism, we use the SDK to run network-vs-network comparisons in which baseline and intervention settings differ only in one localized, versioned policy module. We use the event-complete logs produced by *SWEChain-SDK* to compute outcome measures that capture both efficiency and distribution. These include measures of participation and coverage, price levels and dispersion, convergence behavior, realized software quality, and the concentration of revenue across agents. All derived tables used to produce figures are computed from the logged events, and the artifacts needed for replication under comparable conditions are made available alongside the code.

1.5 Scope, Assumptions, Delimitations, and Limitations

The empirical evaluation in this dissertation is scoped to small, controlled simulation experiments whose primary purpose is to evaluate *SWEChain-SDK* as an experimental platform. The focus is to examine whether a blockchain-native SDK can support controlled, paired baseline–intervention experiments under fixed conditions, not to provide definitive evidence about real-world software-engineering outsourcing markets. As assumptions, the simulations consider autonomous SWE-Agents in a single-market setting with pay-as-bid procurement auctions and deterministic ordering. Runs are short, executed locally, and use fixed task sets, a finite agent pool, shared random seeds, and explicit timing and ordering rules so that baseline and intervention differ only in the policy under test. As delimitations, only a small set of policy rules is varied—specialization, price-signal visibility, and winner selection—and outcomes are summarized mainly in terms of task completion, outsourcing cost, and revenue concentration.

These choices impose clear limitations. Workloads are synthetic and short, agents do not learn or adapt within a trial, and quality is measured through simple, observable proxies. As a result, the experiments do not capture the scale, diversity, or strategic behavior of production settings, and effects that depend on real-world variability in organizations, data, or infrastructure are intentionally out of scope. The empirical results should therefore be read primarily as an evaluation of what *SWEChain-SDK* can support as an experimental SDK, rather than as a comprehensive assessment of decentralized SWE-Agent markets.

1.6 Significance and Contributions

This dissertation is situated at the intersection of intelligent software engineering and software-engineering economics, under the working label of *SWE-Agent Economics*. Its significance lies in treating decentralized, auction-based SWE-Agent outsourcing markets as a setting in which AI agents act as economic decision-makers and in providing research tooling that makes controlled economic network simulations of such markets practically achievable under comparable conditions.

In this context, the dissertation makes three main contributions:

(i) it develops a conceptual and terminological framing of *SWE-Agent Economics* as a domain linking intelligent software engineering and software-engineering economics, with decentralized SWE-Agent outsourcing markets as the central case;

(ii) it presents *SWEChain-SDK*, a configurable, extensible blockchain-native SDK for controlled studies of decentralized, auction-based SWE-Agent outsourcing markets under comparable conditions, providing a local sandbox chain and event-complete logging of bids, allocations, payments, and artifacts to support controlled, paired baseline–intervention experiments under fixed conditions; and

(iii) it releases a public, evaluation-ready bundle of *SWEChain-SDK*—including simulation code and artifacts such as configurations, datasets, manifests, logs, and figure-generation scripts—to support scripted re-runs and extension of the experiments reported in the dissertation under comparable conditions.

Together, these contributions provide an environment and artifact set for studying SWE-Agent outsourcing markets in a way that is transparent, auditable, and repeatable under clearly defined experimental setups.

1.7 Dissertation Structure

The remainder of the dissertation is organized to move from conceptual framing, through system design and methodology, to experimental evaluation and implications.

Chapter 2 introduces the necessary background and terminology. It reviews intelligent software engineering and SWE-Agents, software-engineering economics and outsourcing markets, AI-agent economics, and blockchain-based mechanisms, and uses these strands to motivate SWE-Agent economics and the architectural requirements for a blockchain-native SDK for economic network simulations.

Chapter 3 presents the architecture of *SWEChain-SDK*. It states the design goals, explains how they are instantiated in a local, blockchain-native sandbox for SWE-Agent outsourcing markets, and describes the main components, including the execution environment, auction and settlement logic, logging layer, and agent interfaces.

Chapter 4 reports the simulation experiments conducted with *SWEChain-SDK*. It sets out the experimental design, SWE-Agent models, task datasets, policy configurations, and metrics, and then presents and analyzes paired baseline–intervention experiments that vary agent specialization (comparative advantage), bidders' price-signal utilization, and principal-weighted quality-signal utilization in winner-selection rules under fixed conditions.

Chapter 5 situates the dissertation in related work on intelligent software engineering, software-engineering economics, agent-based and AI-agent economics, mechanism design, evaluation frameworks, and blockchain-based research infrastructures, and contrasts existing approaches with the capabilities provided by *SWEChain-SDK*.

Finally, Chapter 6 summarizes the main findings, reflects on limitations and threats to validity, and outlines directions for future work on SWE-Agent economics and on extending *SWEChain-SDK*, including richer quality models, adaptive agents, and multi-market settings.

Chapter 2
Background

This chapter develops the conceptual foundation for the dissertation. We move from intelligent software engineering and modern Software-Engineering Agents (SWE-Agents) to an economic view of software work, then to agent-based simulations as a way to study economic mechanisms, and finally to blockchain-based decentralized networks as programmable environments in which to implement and observe those mechanisms. We keep the focus on constructs that we later reuse in the architecture and simulation chapters and align the background with the abstract: we are interested in blockchain-based SWE-Agent outsourcing markets; we use a blockchain-native Software Development Kit (SDK) to run controlled, paired baseline–intervention experiments under fixed conditions; and we vary one policy dimension at a time—comparative advantage via agent specialization, competitive bidding via bidders' price-signal utilization, and principal-weighted selection via quality-signal utilization—while measuring task completion, outsourcing cost, and revenue concentration.

2.1 Intelligent Software Engineering

Intelligent Software Engineering (ISE) extends the classic software-engineering arc—from NATO's push for modularity, contracts, and verification (NAUR; RANDELL, 1969; PARNAS, 1972; HOARE, 1969; DIJKSTRA, 1976) to Continuous Integration and Continuous Delivery (CI/CD)'s culture of repeatable delivery (HUMBLE; FARLEY, 2010)—by adding data-driven perception, search, and synthesis across the lifecycle (XIE, 2018; HOU et al., 2024; ZHANG et al., 2023). The basic stance is familiar: interfaces and contracts should be explicit, claims should be verifiable, and control over change should be reproducible. Modern work adds learned components into this picture, but does not abandon it.

A central construct in this dissertation is the *SWE-Agent*: an autonomous software-engineering agent with an explicit controller and a concrete tool interface. Historically, controller designs evolved from reactive Finite-State Machines (FSMs) and subsumption architectures (BROOKS, 1991) through Behavior Trees (BTs) (COLLEDANCHISE; ÖGREN, 2018) and Belief–Desire–Intention (Belief–Desire–Intention (BDI)) frameworks (RAO; GEORGEFF, 1995; WOOLDRIDGE, 2009) to hybrids that combine reactive safety with deliberative goal management. These architectures share the property that control flow is explicit and testable: each state and transition can be examined, every action is whitelisted, and behavior is repeatable under fixed conditions. Transformer-based Large Language Models (LLMs) (VASWANI et al., 2017), including code-tuned variants (CHEN et al., 2021; ROZIÈRE; GEHRING; AL., 2023; LI; WERRA; AL., 2023), now act as replaceable subroutines inside such controllers for perception, planning, and code synthesis. Orchestration patterns such as Reason+Act (interleaved planning and tool use) (ReAct)-style interleaving of reasoning and tool calls (YAO et al., 2023) and behavior-tree or FSM executives with guarded transitions provide a way to embed stochastic model calls without giving up determinism at the controller level.

In this dissertation we use SWE-Agent in a specific and reproducible sense. An agent is an explicit controller with a documented Agentic Capability Interface (ACI) for repository and test interaction (for example, list/open/search/edit/run), within which LLM calls act as stochastic subroutines in semantic states such as plan, implement, and repair. Deterministic guards based on compiler and test outcomes, schema validation, and simple invariants gate transitions and termination, and prompts, seeds, decoding parameters, and environment hashes are recorded. In the SWEChain simulations, these controllers are implemented as finite-state machines with fixed, hand-designed policies and prompts rather than as adaptive or learning systems; we cite richer architectures such as BDI and reinforcement learning only as conceptual predecessors. This explicit, non-learning controller structure supports comparability and provenance and allows causal attribution to specific policy choices—such as prompts, agent specialization, bidders' price-signal utilization, and principal-weighted quality-signal utilization—rather than to incidental randomness or uncontrolled adaptation.

Reproducibility follows familiar engineering instincts: make control flow explicit, keep acceptance objective, and record enough context that another team can retrace the steps. We realize this by sequencing a small set of semantic states (plan, implement, repair) under an FSM whose edges are guarded by facts the toolchain can attest: compiles succeed, tests pass, and schemas validate (HAREL, 1987; COLLEDANCHISE; ÖGREN, 2018). LLMs are confined to these states and may propose plans or edits but cannot bypass gates. We fix random seeds and decoding parameters for LLM calls; pin compilers, interpreters, and runners in containerized environments (MERKEL, 2014; KLEPPMANN, 2017); impose deterministic tie-breaks when scores collide; and serialize non-commutative effects using defined clocks and ordering (LAMPORT, 1978; SIGELMAN et al., 2010). With the substrate held still, we can run paired baseline–intervention experiments under fixed conditions where exactly one policy dimension varies at a time (specialization, price-signal utilization, or quality-signal utilization), so that observed differences can be attributed to that policy rather than to infrastructure drift. This pattern supplies the "agent" side of the SWE-Agent Economics framing used later.

2.2 Software Engineering Economics

Software engineering economics (Software Engineering Economics (SEE)) provides constructs for treating software work as production under scarcity and for describing mechanisms that allocate work and payments. Economics studies choice under scarcity: ends exceed means, so agents allocate limited resources and accept opportunity costs (ROBBINS, 1932; SAMUELSON; NORDHAUS, 2010; VARIAN, 2014). Read through this lens, the NATO reports did more than name a "software crisis"—they recast software development as a production process with explicit decision variables and constraints (NAUR; RANDELL, 1969). Modularity and explicit interfaces make delivered functionality and integration work countable; formal specifications and verification convert quality into a contractible objective; configuration control and versioning turn effort, rework, and schedule slippage into observables. Trade-offs among cost, schedule, scope, and quality can therefore be modeled as objectives and constraints, estimated as priors, and updated from data (BOEHM, 1981; BOEHM et al., 2000).

Classical results quantify these levers. Brooks explained declining and sometimes negative marginal productivity from late staffing due to rising communication and coordination costs (JR., 1975). Putnam's Rayleigh–Norden life-cycle model traced the schedule–effort frontier (PUTNAM; MYERS, 1992), and parametric models such as COCOMO related effort to size via multiplicative drivers and diseconomies of scale (BOEHM, 1981; BOEHM et al., 2000). Prevention and appraisal investments such as inspections and testing rise non-linearly to contain interaction effects as systems scale (BANKER; KAUFFMAN; MOREY, 1994), and technical debt names an explicit intertemporal trade-off between expedient design and future rework (CUNNINGHAM, 1992). Because software is labor-intensive, human heterogeneity matters; disciplined processes and feedback loops (for example, via CI/CD) help make cost, latency, and quality observable and manageable (HUMBLE; FARLEY, 2010).

We view software outsourcing through the lens of mechanism design. A requester has a task with scoped deliverables, a deadline, a budget or reserve, and executable acceptance tests. Potential suppliers (firms, teams, or SWE-Agents) have private costs and limited capacity, and must decide whether to enter and, where applicable, at what price. Mechanisms repair information and coordination frictions by specifying an allocation rule, a payment rule, and an information policy (KRISHNA, 2009a; KLEMPERER, 2004; MYERSON, 1981; LAFFONT; MARTIMORT, 2002). Reverse posted-price mechanisms are simple and suitable for standardized work; reverse sealed-bid first-price auctions provide fast cost screening; scoring or multi-attribute auctions fold non-price attributes such as expected quality and latency into a weighted score that is cleared competitively (CHE, 1993). Information policy and clearing cadence (continuous versus frequent-batch) affect participation, competition, and racing behavior (KLEMPERER, 2004; BUDISH; CRAMTON; SHIM, 2015a). We use the term *software outsourcing auction* for institutions that map scoped tasks, budgets or reserves, and acceptance tests to allocations and payments via explicit rules, under adverse selection and moral hazard (AKERLOF, 1970; HOLMSTRÖM; MILGROM, 1991).

Outcomes of interest include task completion rates, latency distributions, cost per accepted task, price dispersion, and concentration in revenues or market share. Classical diagnostics such as Lorenz curves, Gini indices, and top-k shares summarize how work and reward concentrate or spread (FENTON; BIEMAN, 2014; GINI, 1912; EASLEY; KLEINBERG, 2010). In this dissertation we implement only a narrow family of reverse-procurement and scoring auctions on a single chain and vary a small number of governed parameters and agent-side policies, rather than exploring the full space of auction formats. SEE furnishes the language for these mechanisms and the metrics used to compare them in simulation.

2.3 Agent-Based Simulations

Agent-based modeling (Agent-Based Modeling (ABM)) and agent-based computational economics (Agent-Based Computational Economics (ACE)) link local decision rules and institutional settings to system-level outcomes (RAILSBACK; GRIMM, 2019; EPSTEIN; AXTELL, 1996; TESFATSION, 2002). Systems are represented as populations of heterogeneous agents with local state, information, and decision rules, interacting under frictions and timing. Local rules for search, bidding, implementation, and review, together with institutional levers for allocation, payments, and cadence, jointly generate macro outcomes such as throughput, congestion, coordination cost, concentration, and inequality. This picture matches software task markets, where small changes in auction rules or timing can induce large shifts in who participates, how aggressively they bid, and how work and rewards concentrate.

Foundational examples provide design patterns rather than blueprints. Schelling's segregation model showed how mild local preferences over neighborhood composition can yield macro-level segregation (SCHELLING, 1971a). Axelrod's iterated-prisoner's-dilemma tournaments operationalized strategy evaluation as an open competition in a fixed environment, with heterogeneous strategies interacting repeatedly under noise (AXELROD, 1984). The Santa Fe artificial stock market connected simple trading heuristics to price dynamics and volatility (ARTHUR et al., 1997), while models such as Sugarscape and Echo linked resource and credit dynamics to distributional outcomes (EPSTEIN; AXTELL, 1996; HOLLAND, 1995). RoboCup Simulation and NetLogo illustrated how standardized environments and tooling can turn agent-based models into reusable benchmarks (KITANO; VELOSO et al., 1997; WILENSKY, 1999). We draw on these precedents mainly for three ideas: fix rules and instrumentation, allow heterogeneous strategies, and evaluate outcomes from logged traces.

In software engineering, this general ABM view maps naturally to ecosystems of heterogeneous actors—developers, teams, tools, platforms, and clients—whose local decisions and interactions generate system-level outcomes (ABDEL-HAMID; MADNICK, 1991). We use modeling primitives that mirror software reality. Agents carry state (skills, private costs, backlog, reputation, liquidity), capabilities (search, bid, implement, review, test), and decision rules ranging from simple heuristics to learned policies. Tasks encode requirements, dependencies, and acceptance tests. Resources include developer time, compute, data, and budget. Institutions govern allocation and payments (for example, reverse auctions and reserves), information policy, and cadence (continuous versus frequent-batch clearing). Environments encode arrivals, dependencies, queues, search and matching frictions, and time-zone or handoff costs. Interaction protocols cover search and entry, bidding and awarding, delivery and review, and dispute resolution. These ingredients support experiments that ask how specific policy choices map to performance and distributions, including concentration diagnostics such as Lorenz and Gini curves (FENTON; BIEMAN, 2014; GINI, 1912).

In this dissertation we use generative-agent modeling in a constrained way. We embed generative components such as LLMs inside explicit executives FSMs and gate them with deterministic checks (compilation and tests, schema validation) (COLLEDANCHISE; ÖGREN, 2018; OUYANG et al., 2024). Prompts, tool calls, outputs, seeds, decoding parameters, and environment hashes are journaled so that runs can be rerun under comparable conditions. Agent policies are fixed across trials; we do not allow cross-trial learning or adaptation, and instead focus on how mechanism and configuration changes affect outcomes under a given family of controller and prompt designs. Outcome variables track both levels and distributions: primary observables include cost (developer time, compute, and fees), schedule (lead and cycle time), and quality (defects, rework, and acceptance), while market observables include surplus, price dispersion, and concentration indices (FENTON; BIEMAN, 2014; GINI, 1912). Empirical game-theoretic analysis Empirical Game-Theoretic Analysis (EGTA) suggests ways to summarize induced incentives before comparing mechanism variants under comparable conditions (WELLMAN, 2006); here we borrow its emphasis on fixed environments and reproducible comparisons without implementing a full EGTA pipeline.

Experimental design follows guidance from empirical software engineering (WOHLIN et al., 2012; KITCHENHAM et al., 2009). In the broader simulation literature, verification, validation, and uncertainty quantification (Verification, Validation, and Uncertainty

Quantification (VVUQ)) are treated as necessary checks rather than optional extras: implementations are verified against specifications and invariants, behaviors are validated against stylized facts and traces from version control and issue trackers, and Monte Carlo replication and sensitivity analysis are used to characterize uncertainty (SARGENT, 2013; OBERKAMPF; ROY, 2010; LAW, 2015; FENTON; BIEMAN, 2014). In this dissertation we adopt only a subset of these practices. We design paired baseline–intervention experiments under fixed seeds and toolchains, block by workload mix where appropriate, and pre-specify outcomes, but we do not carry out large-scale Monte Carlo replication or full VVUQ campaigns. Tournament-style evaluations, such as Axelrod's iterated-prisoner's-dilemma experiments, motivate the idea of a fixed environment in which different strategies can be compared under identical conditions; in our case, we run a closed set of hand-designed policies rather than an open competition. These choices keep the simulations small and controlled, and they are sufficient for the goals of this dissertation.

2.4 Blockchain Decentralized Networks

We now introduce the environments in which we implement mechanisms and run experiments. We treat blockchains and related decentralized networks as programmable, auditable substrates for economic mechanisms. We use this section to explain why application-specific chains are appropriate for SWE-Agent outsourcing experiments and to define basic terminology for consensus, finality, and execution that we use later when we describe SDK for SWE-Agent Economic Simulations on SWEChain (SWEChain-SDK).

Blockchains address a coordination problem in open, adversarial networks: independent machines that do not trust one another need to construct, verify, and preserve a single canonical sequence of state updates without a central intermediary. The solution ties together an append-only ledger, digitally signed transactions, and a consensus protocol that selects a unique total order of valid transactions. When combined with a deterministic state transition function, these primitives realize a replicated state machine: every honest node processes the same ordered input stream and converges to the same state (NAKAMOTO, 2008; GARAY; KIAYIAS; LEONARDOS, 2015; EYAL; SIRER, 2014; KIAYIAS et al., 2017; BUTERIN; GRIFFITH, 2017). A crypto-economic layer aligns incentives: participants who expend resources to propose and validate blocks are compensated via rewards and fees, while protocol-enforced penalties (for example, slashing in proof-of-stake systems) make deviation costly.

Programmability extends this model from payments to general stateful coordination. A virtual machine with explicit gas metering executes untrusted code safely; contracts can escrow assets, emit events, and settle based on deterministic predicates (BUTERIN, 2014; WOOD, 2014). Application-specific chains specialize this model: they reuse consensus and networking layers but define their own modules and parameters. For our purposes, an application-specific chain is most naturally viewed as code that implements an economic mechanism with explicit rules for admission, ordering, execution, settlement, and governance.

Single-chain designs couple all applications into one execution environment, one mempool, and one fee market. As usage diversifies, this coupling can become a source of confounds: demand spikes in one domain propagate as congestion and higher inclusion costs to unrelated workloads, and the economic value of transaction ordering induces Maximal Extractable Value (MEV) strategies that skew fee and inclusion patterns (DAIAN et al., 2020a; ROUGHGARDEN, 2021). One response is a move to networks of chains with explicit choices over security sharing, inter-chain messaging, and execution packaging. Modularity decouples execution, ordering, and data availability so each can scale or specialize independently; sovereignty with interconnection deploys application-specific chains that control their own consensus parameters, mempool and fee policies, and governance, while interconnecting through verifiable cross-chain messaging (KWON; BUCHMAN, 2016; FOUNDATION, 2020). For this dissertation, we need only a simple point in this design space: a single, sovereign application-specific chain running locally as a deterministic state machine.

Cosmos SDK with CometBFT consensus provides such a substrate: modules define application logic; the base application routes messages and queries; deterministic finality is obtained once a local quorum commits a block (BUCHMAN, 2018; KWON; BUCHMAN, 2016). We adopt this setting to minimize cross-application coupling and exogenous noise: there is no MEV, no competing chains, and no adversarial network. Instead, Software-Engineering Chain (application-specific blockchain) (SWEChain) is run as a single-node appchain whose code, parameters, and genesis are fixed per experiment profile. A blockchain-based application-specific chain can then be treated as code that implements an economic mechanism: the mechanism, in the market-design sense, maps participants' messages to allocations and payments under a specified information and timing policy; on the chain, this mapping is realized as a deterministic state machine with protocol parameters for admission (signature and nonce checks), ordering (consensus and mempool policy), pricing (fee mechanism), and settlement (finality semantics). Under this view, an agent is software that observes on-chain state, forms a strategy within transparent rules, and submits signed transactions to realize objectives.

Historically, autonomous agents on ledgers precede current LLM-based systems. In Ethereum Virtual Machine (EVM)-based decentralized finance, for example, keepers watch collateralization and trigger protocol-specified auctions; liquidators close unsafe loans; and searchers scan public mempools and on-chain liquidity for arbitrage and back-run opportunities (DAIAN et al., 2020a; ANGERIS et al., 2020). In Cosmos appchains, deterministic finality simplifies state-contingent strategies and Inter-Blockchain Communication (IBC) relayers act as first-class agents that carry light-client-verified packets between sovereign chains (FOUNDATION, 2020; KWON; BUCHMAN, 2016; BUCHMAN, 2018). We do not model these details here; they serve only as examples that the same general pattern repeats: mechanisms are encoded as software with explicit rules, and agents operate within those rules. This view motivates our later choice to implement SWE-Agent outsourcing auctions as modules in a Cosmos SDK appchain.

2.5 Discussion

This discussion gives a working definition and scope for what we call *SWE-Agent Economics*. We use the term as a convenient label for a specific intersection of existing areas. From intelligent software engineering, we take SWE-Agents to be autonomous software-engineering agents with explicit controllers (for example, FSMs, behavior trees, or BDI-style executives), documented tool interfaces, and objective acceptance checks on their outputs (NAUR; RANDELL, 1969; PARNAS, 1972; HOARE, 1969; DIJKSTRA, 1976; HUMBLE; FARLEY, 2010; BROOKS, 1991; RAO; GEORGEFF, 1995; WOOLDRIDGE, 2009; COLLEDANCHISE; ÖGREN, 2018; HOU et al., 2024; ZHANG et al., 2023). From software engineering economics, we take the view that software work is carried out under constraints on budget, calendar time, and attention, and that trade-offs among cost, schedule, scope, and quality can be expressed as objectives and feasible frontiers (BOEHM, 1981; BOEHM et al., 2000). Mechanism design and procurement supply language for explicit task-allocation rules: allocation rules map bids to winners, payment rules map bids to transfers, information policies describe what is visible to whom and when, and properties such as Incentive Compatibility (IC) and Individual Rationality (IR) are design targets rather than slogans (CHE, 1993; KRISHNA, 2009b; KLEMPERER, 2004; CRAMTON; SHOHAM; STEINBERG, 2006; MYERSON, 1981; LAFFONT; MARTIMORT, 2002). We use *software outsourcing auction* to mean an institution in which a requester posts a scoped task with a budget and executable acceptance tests, potential suppliers decide whether to participate (and, where applicable, at what price), and an explicit mechanism clears and settles on verifiable delivery.

Agent-based modeling and market modeling provide constructs for linking local decision rules to system-level behavior in such settings. We assume heterogeneous agents with private information and bounded decision procedures, interacting under institutional rules and timing, generate observable outcomes such as throughput, latency distributions, cost per accepted task, price dispersion, and inequality in revenues or market share (RAILSBACK; GRIMM, 2019; EPSTEIN; AXTELL, 1996; SHOHAM; LEYTON-BROWN, 2009; FENTON; BIEMAN, 2014; GINI, 1912). We use *outcome* for metrics computed from event-level traces, including completion rates and simple network summaries on issue–agent graphs. Simulation-based studies, with verification, validation, and uncertainty quantification (VVUQ) treated as standard practice, are a common way to explore how changes in rules affect these outcomes (ABDEL-HAMID; MADNICK, 1991; LAW, 2015; SARGENT, 2013; OBERKAMPF; ROY, 2010; WOHLIN et al., 2012; KITCHENHAM et al., 2009). Blockchain and related decentralized systems then supply concrete environments in which mechanisms are implemented as code: an application-specific chain can be viewed as a deterministic state machine with software-defined transition rules, parameters, and events, and consensus and finality provide a canonical sequence of state changes (BUTERIN, 2014; WOOD, 2014; BUCHMAN, 2018; KWON; BUCHMAN, 2016). We use *environment* to mean a specific configuration of such a system—for example, a Cosmos SDK appchain with CometBFT consensus, particular module code, parameter sets, and fee policy—in which admission checks, ordering, settlement, and governance are fixed and observable, and which emits height-stamped events for messages, state changes, and parameter updates.

Within this vocabulary, we use SWE-Agent Economics in this dissertation to refer to the study of autonomous SWE-Agents, engineered in the intelligent software engineering sense, acting as economic decision makers in task markets whose objectives and mechanisms are drawn from software engineering economics and whose behavior is analyzed with agent-based and market-modeling methods. A SWE-Agent is an explicit, testable coding agent with a defined tool interface; a task is a scoped software work item with a budget or reserve and machine-executable acceptance criteria; a mechanism is a rule system that maps agents' messages to allocations and payments under a specified information and timing policy; an environment is a concrete execution setting that implements the mechanism as software with particular ordering, fee, and governance properties; and outcomes are efficiency, quality, market-structure, and distributional metrics derived from event-level traces (BOEHM, 1981; BOEHM et al., 2000; RAILSBACK; GRIMM, 2019; EPSTEIN; AXTELL, 1996; FENTON; BIEMAN, 2014; GINI, 1912; KLEMPERER, 2004). This is a working definition, narrow enough to be grounded in the cited background and broad enough to cover the cases we actually study. It sits alongside, rather than replaces, related areas such as AI-agent economics and ACE, which already examine autonomous agents in economic environments (SHOHAM; LEYTON-BROWN, 2009; RAILSBACK; GRIMM, 2019; EPSTEIN; AXTELL, 1996), and blockchain economics, which already treats blockchains as economic mechanisms with explicit fee markets and governance (BUTERIN, 2014; WOOD, 2014; BUCHMAN, 2018; KWON; BUCHMAN, 2016). The term SWE-Agent Economics highlights the intersection where agents are SWE-Agents in this software-engineering sense, tasks are software tasks with executable tests, and mechanisms and outcomes are described using the language of software engineering economics and market design.

The remainder of the dissertation focuses on a concrete subclass of this intersection. We restrict attention to settings in which tasks are software outsourcing issues, suppliers are SWE-Agents, and mechanisms are auction-based market designs deployed on a decentralized blockchain. Concretely, each task is a software work item with explicit scope, a budget or reserve, and machine-executable acceptance tests; suppliers are autonomous SWE-Agents that read issues, estimate costs, decide whether to participate, implement, and submit evidence of completion under explicit controller logic; and mechanisms are reverse procurement or scoring auctions with parameters for reserves, eligibility filters, scoring weights, penalties, and clearing cadence, implemented as on-chain modules on an application-specific chain (CHE, 1993; KRISHNA, 2009b; KLEMPERER, 2004; CRAMTON; SHOHAM; STEINBERG, 2006; BUDISH; CRAMTON; SHIM, 2015b; MYERSON, 1981; LAFFONT; MARTIMORT, 2002; BUCHMAN, 2018; KWON; BUCHMAN, 2016). SWEChain-SDK is one concrete realization of this subclass: a blockchain-native SDK that instantiates SWE-Agent outsourcing auctions on a local Cosmos SDK appchain, exposes a small set of governed parameters (such as reserve semantics, eligibility filters, scoring weights, and auction duration), and couples the chain to deterministic workload generators and agent runtimes. Runs are driven by manifests; randomized components are seeded; toolchains and binaries are pinned; and on-chain events and off-chain agent actions are logged at a level of detail sufficient to reconstruct metrics and figures or to script additional runs under comparable conditions. In the simulations reported here, agent policies are fixed across trials; we do not allow cross-trial learning or adaptation, and instead focus on how mechanism and configuration changes affect outcomes under a given policy family.

We then use this SDK to conduct paired simulation experiments that vary one policy dimension at a time—(A) comparative advantage via agent specialization, (B) competitive bidding via bidders' price-signal utilization, and (C) principal-weighted selection via quality-signal utilization—while keeping rules, datasets, random seeds, time base, and the agent pool fixed, and we report the observed paired differences in task completion, outsourcing cost, and revenue concentration. The discussion in this section names and narrows the slice of existing work that we build on and provides a reference point for the architecture and experiments developed in the rest of the dissertation.

Chapter 3 SWEChain-SDK Architecture

In the background chapter we introduced SWE-Agent Economics as a working label for the intersection of intelligent software engineering and software-engineering economics, centered on task markets, incentive rules, and information structures. This chapter moves from that general framing to a blockchain-native architectural instantiation: *SWEChain-SDK*, a blockchain-native software development kit designed to support controlled, paired baseline–intervention experiments under fixed conditions in decentralized SWE-Agent outsourcing markets.

This dissertation states three main aims for *SWEChain-SDK*. First, it should provide a local, blockchain-based sandbox chain for SWE-Agent outsourcing auctions with programmable, transparent, and auditable mechanisms that centralized platforms rarely expose. In this work, that role is played by a Cosmos-SDK appchain, *SWEChain*, that implements a software outsourcing issue market via a custom `issuemarket` module. Second, SWEChain-SDK should support controlled, paired experiments that vary a single policy dimension at a time—comparative advantage via agent specialization (Experiment A), competitive bidding via bidders' price-signal utilization (Experiment B), or principal-weighted selection via quality-signal utilization (Experiment C)—while keeping rules, datasets, random seeds, time base, and the agent pool fixed. Third, it should provide event-complete logging of bids, allocations, payments, and artifacts, together with configuration metadata sufficient to support re-runnable experiments under comparable conditions.

At a system level, *SWEChain-SDK* is intended as a compact, local-first toolkit for controlled economic network simulations of decentralized, auction-based software-engineering markets. The guiding principle is to isolate the effect of one policy variable at a time by composing small binaries with clear inputs and outputs, deterministic behavior where feasible, and version-pinned environments. Concretely, the SDK ecosystem provides: (i) a local Cosmos SDK blockchain (*SWEChain*) that encodes and enforces outsourcing auction rules via the `issuemarket` module; (ii) access layers that expose SWEChain functionality to humans (via the `swechaind` command-line interface) and to SWE-Agents (via a Model Context Protocol (MCP) bridge); and (iii) companion simulation, agent, dashboard, and analytics tooling that prepare datasets and manifests, run paired trials, and turn event streams into metrics and figures. Determinism is encouraged by seeding randomized components, pinning toolchains, and keeping module parameters and binaries stable across baseline and intervention.

The public GitHub repository `swechain-sdk`[1] serves as the canonical gateway to this ecosystem. It provides the stable entry point referenced in this dissertation and in the evaluation-ready release mentioned in the abstract. Internally, `swechain-sdk` organizes the SWEChain appchain implementation, MCP servers, agent runners, simulation drivers, dashboards, and analytics utilities as versioned components. The gateway repository does not require there to be a single implementation of each off-chain role; rather, it curates a reference set of MCP servers, agent runners, and simulation drivers used in this dissertation, while allowing alternative implementations to be plugged in if they adhere to the same contracts. This indirection allows the SDK's internal layout and auxiliary repositories to evolve over time while preserving a fixed, citable external reference for the architecture described here.

At the system-context level, the primary user is a researcher who configures experiments and inspects results. Figure 1 depicts this view. The researcher interacts with the SWEChain-SDK system, which is shown as a single yellow box representing the appchain and its surrounding tooling. The system executes on a single local host machine under the researcher's control, and interacts with two external services: a LeetCode-like task corpus that provides programming problems for the synthetic data generation engine (SDGE), and a remote or local LLM service such as Ollama or OllamaCloud, which is used by MCP tools for model calls. The arrows indicate how the researcher configures experiments and inspects results, how SDGE consumes the external corpus as input, how SWEChain-SDK uses LLM inference via MCP, and how all of these activities are hosted on the local machine.

[1] <https://github.com/lascam-UFU/swechain-sdk>

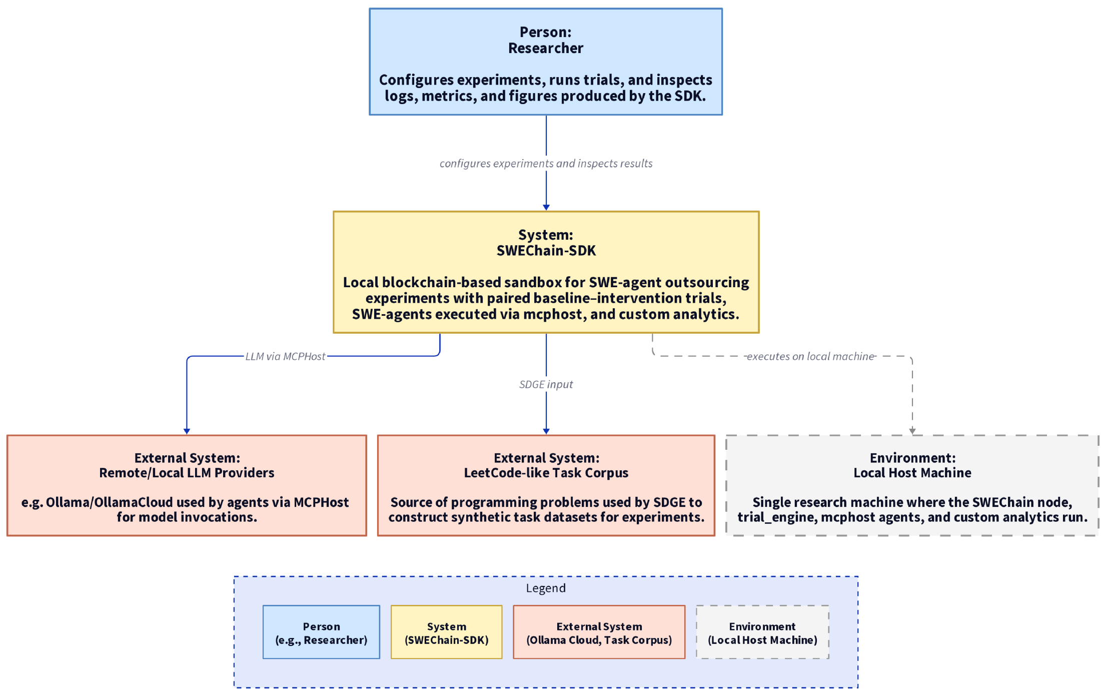

Person:
Researcher
Configures experiments, runs trials, and inspects logs, metrics, and figures produced by the SDK.
configures experiments and inspects results
System:
SWEChain-SDK
Local blockchain-based sandbox for SWE-agent outsourcing experiments with paired baseline–intervention trials, SWE-agents executed via mcphost, and custom analytics.
LLM via MCPHost
SDGE input
executes on local machine
External System:
Remote/Local LLM Providers
e.g. Ollama/OllamaCloud used by agents via MCPHost for model invocations.
External System:
LeetCode-like Task Corpus
Source of programming problems used by SDGE to construct synthetic task datasets for experiments.
Environment:
Local Host Machine
Single research machine where the SWEChain node, trial_engine, mcphost agents, and custom analytics run.
Legend
Person
(e.g., Researcher)
System
(SWEChain-SDK)
External System
(Ollama Cloud, Task Corpus)
Environment
(Local Host Machine)

Figure 1 – SWEChain-SDK system context. The researcher configures experiments and inspects results; SWEChain-SDK runs on a local host machine, consuming a LeetCode-like task corpus through SDGE and accessing LLM services through MCP tools.

Blockchain appchains provide a technically and economically expressive substrate for SWE-Agent experiments: they allow researchers to encode auction and settlement rules as auditable on-chain logic, isolate fee and ordering markets to reduce cross-application confounds, and produce canonical event streams suitable for detailed provenance. In this dissertation, SWEChain is used to execute "network-vs-network" comparisons in which baseline and intervention share rules, datasets, seeds, and agent pool, and differ only in a policy dimension such as specialization, price-signal utilization, or quality-signal utilization. The outcome is a controlled baseline–intervention experiment that can, in principle, be audited and re-run under comparable conditions using the same experimental setup, much like a software build.

3.1 Architectural Requirements

This section distills the architectural requirements that follow from the research agenda in the abstract and from the conceptual framing of SWE-Agent Economics in the background chapter. We state the requirements as architecturally significant requirements (ASRs) and then relate them to the SDK's realization.

From the perspective of this dissertation, the architecture is expected to satisfy at least the following ASRs:

- ❏ **R1 – Reproducibility under comparable conditions.** It should be possible to re-run a paired baseline–intervention experiment under comparable conditions from published artifacts and experiment logs that record chain and agent behavior at the level needed for the study, with rules, datasets, seeds, and toolchains held fixed as far as practicable.
- ❏ **R2 – Controlled variation (ceteris paribus).** Experiments should be able to vary a single policy dimension at a time (for example, specialization, price-signal utilization, quality-signal utilization) while holding all other aspects—mechanism, workload, agent pool, and random draws—fixed.
- ❏ **R3 – Mechanism transparency and auditability.** Market rules for admission, matching, pricing, and settlement should be encoded as verifiable code, with governed parameters and provenance that allow auditors to reconstruct outcomes and identify the effect of policy changes.
- ❏ **R4 – Agent heterogeneity and policy diversity.** The architecture should support heterogeneous SWE-Agents with different specializations, bidding heuristics, and information-use strategies, without compromising the determinism or safety of the underlying mechanism.
- ❏ **R5 – Observability and network analysis.** The system should expose structured data that support not only aggregate Key Performance Indicators (KPIs) such as throughput and cost, but also network-level analysis (for example, degree distributions, assortativity) and distributional analysis (for example, inequality in revenues).
- ❏ **R6 – Configurability and extensibility.** Researchers should be able to introduce new mechanisms, policies, tools, and analytics without breaking existing experiments or invalidating previous results, and with clear boundaries between mechanism-level and experiment-level changes.
- ❏ **R7 – Practicality for local experimentation.** The SDK should remain small and local-first so that experiments, including the A/B/C simulations in the abstract, can be run on a single machine using reproducible scripts and profiles.

These requirements link directly to the contributions stated in the abstract: R1 and R2 support controlled, paired baseline–intervention experiments under shared experimental setups; R3–R5 support SWE-Agent economic analysis with detailed provenance; R6 aligns with the claim that *SWEChain-SDK* is configurable and extensible; and R7 ensures the

SDK is usable as a practical research artifact. Later sections of this chapter revisit these requirements explicitly for SWEChain-SDK.

Table 1 maps conceptual needs in this domain to decentralized requirements and to concrete *SWEChain-SDK* design choices. It provides a bridge between the abstract domain framing and the ASRs.

Table 1 – Conceptual needs in decentralized SWE-Agent economics, corresponding decentralized requirements, and their realization in *SWEChain-SDK*.

Conceptual need	**Decentralized requirement**	**Realization in the SDK**
Mechanism as code	Verifiable, governed market rules; deterministic outcomes where possible	Cosmos appchain with `swechain` and `issuemarket` modules (messages, parameters, events), deterministic tie-breaking policies, escrow via `bank`
Ceteris-paribus identification	Vary one policy, hold others fixed	Paired baseline–intervention manifests; fixed seeds/datasets/schedules; pinned toolchains
Shared state & auditability	Auditable record of transitions	ABCI events keyed by height; dev-genesis and run headers; experiment-specific exports used by analytics tools
Agent heterogeneity	Specialization, information-use, bidding heuristics	Controllers (FSM/BT) with private tag sets; LLM subroutines confined to guarded states; policy described in files
Information & scoring	Transparent, governed policy and prompt changes	On-chain parameters (disclosure flags, scoring weights) via parameter-update messages, and off-chain prompt/controller configurations; all height-stamped or versioned
Network analysis	Structure beyond aggregates	Bipartite issue–agent graphs; degree/assortativity; inequality metrics (for example, Gini/HHI)
Comparable KPIs	Stable efficiency and quality metrics	Metrics engine computing throughput, $p95$ latency, cost per task, pass/rework, price dispersion
Reproducibility & provenance	Experiment logs and artifacts that support re-running paired trials under comparable conditions and auditing outcomes	Stable filenames; checksums; commit and genesis recorded in `run_header.json` and related headers

The remainder of the chapter describes how these requirements are addressed through design principles, the composition of on-chain and off-chain components, and the configuration and extension surfaces of *SWEChain-SDK*.

3.2 Design Principles

The design philosophy of *SWEChain-SDK* is explicitly Unix-like. The SDK favors a small, correctness-critical kernel, narrow and stable interfaces, plain and inspectable data, and compositions of simple programs over large, monolithic services. The architecture is intended to allow decentralized SWE-Agent outsourcing markets to be expressed as code on a local chain, while experiment policy and analysis stay in separate, scriptable layers that can evolve without compromising determinism or auditability where these are achievable.

On chain, the kernel is a Cosmos SDK application (`BaseApp` plus a small set of modules, including the custom `swechain` and `issuemarket` modules). Off chain, orchestration, agents, tools, dashboards, and analytics are standalone processes that communicate through typed files and streams. The `swechain-sdk` gateway repository does not collapse these into a single binary; instead, it collects the chain, MCP servers, agent runners, simulation drivers, dashboards, and analytics utilities into a coherent release with pinned versions and reproducible profiles. This split preserves a clear trust boundary between consensus state and research tooling and makes it possible to change methods at the edges without disturbing the deterministic core.

Doing one thing well starts at the module boundary. `issuemarket` is scoped to auctions and bids: it owns a namespaced store and a focused surface of messages, queries, parameters, and events. Identity, balances and escrow, and governance remain the responsibility of base modules. By keeping consensus logic minimal and explicit, the module remains testable and auditable at the level of detail required for this study. Cross-module effects are expressed via typed keeper calls and events rather than implicit side effects.

Composition is treated as dataflow. On chain, ABCI events form the primary stream: each state transition emits types and attributes sufficient to reconstruct relevant causal chains by height. Off chain, manifests, logs, and metrics are treated as text streams: JSONL and CSV with stable schemas that tools (and humans) can read, write, and pipe together. Agents consume events, maintain local memory in bbolt-backed stores, and invoke tools over stdio protocols; this mirrors the Unix "pipe simple programs together" idea in a setting oriented toward simulation-based studies of SWE-Agent economics. For each auction, the chain's event stream includes explicit events for bids, winner selection, and settlement transfers, so that the economic outcomes studied in this dissertation can be reconstructed from logs and analysis outputs.

Mechanism and policy are separated. Mechanism lives in the deterministic state machine—message handlers, keeper methods, and bounded *BeginBlock*/*EndBlock* hooks. Policy lives in governed parameters and off-chain configuration, including agent controllers and prompts. Market rule changes are gated by parameter-update messages and height-stamped; experimental conditions are captured in file-backed manifests with checksums and seeds. The same binary kernel can host several experiments, while differing policies do not need to compromise safety or liveness. This separation is important for credible baseline–intervention comparisons.

Interfaces are narrow, explicit, and stable. On chain, gRPC/REST expose module messages and queries, and event types and fields form a public contract.[2] Off chain, the simulation drivers, agents, and tools exchange versioned JSON messages and report failures as data. Any component is replaceable if it preserves its contract: a new sampler, agent state, or tool can be introduced without touching the chain or rewriting unrelated services.

Plain data is preferred over complex abstractions. Event and agent logs are JSONL; metrics are CSV; manifests are explicitly versioned. Artifacts are intended to be as easy to read as to write, enabling straightforward inspection. Configuration is declarative and file-backed; environment variables are used only for ephemeral secrets; every run writes a header with commit identifiers, genesis checksums, parameter snapshots, and profile versions. This shortens the path from "what happened?" to "show me the evidence." In practice, we combine automated analysis scripts with lightweight inspection tools such as a terminal block explorer for SWEChain and a TUI browser over per-agent bbolt files; screenshots of these tools are provided in Appendix D.

Reproducibility under fixed experimental setups is pursued through determinism and pinning. The chain is deterministic by construction; experiments fix seeds, pin toolchains, and wait for height 1 before orchestration. Node and build profiles codify timeouts, fee policy, and versions so developer, continuous-integration, and long-run modes can be aligned. In the Unix spirit, these profiles are parameterized invocations of the same programs, which reduces drift between environments.

Failure is treated as a source of data rather than something to hide. When work cannot proceed, the SDK logs inputs, decisions, and errors into the same structured streams that support analytics. On-chain invariants and ante handlers are intended to fail closed; off-chain processes fail fast with bounded retries and typed reasons; nothing mutates global state without an associated trail. Treating "errors as data" means that the files used to generate figures also record cases such as timeouts or mispriced bids.

Extensibility is incremental and orthogonal. New capability arrives as new modules, messages, parameters, queries, events, tools, states, or analytics sinks, rather than as silent reinterpretations of existing semantics. On chain, migration handlers and export/import parity support long-lived contracts; off chain, regression tests and artifact comparisons can be used to guard against unintended changes. Keeping pieces small, making contracts obvious, and connecting them with simple dataflows allows the SDK to accommodate new SWE-Agent market designs while preserving comparability with prior experiments.

Finally, the technology stack is chosen to support these aims. The kernel is implemented in Go to obtain static binaries, predictable performance, mature tooling (`go mod`, `vet`,

[2] In this chapter we refer to events by conceptual names such as "auction created" and "bid submitted"; the concrete Cosmos SDK event types follow the standard `type/attribute` naming and are documented in the SDK repository.

`lint`), and strong protobuf/gRPC support; Cosmos SDK and CometBFT provide a deterministic state-machine substrate with clear module boundaries. Around that verifiable core, the rest of *SWEChain-SDK* behaves like a toolbox: the chain is the journal, events are the pipes, manifests and logs are the files, and experiments are compositions of simple programs that instantiate the research agenda of SWE-Agent Economics in concrete, controlled simulation settings.

3.3 Components and Interconnections

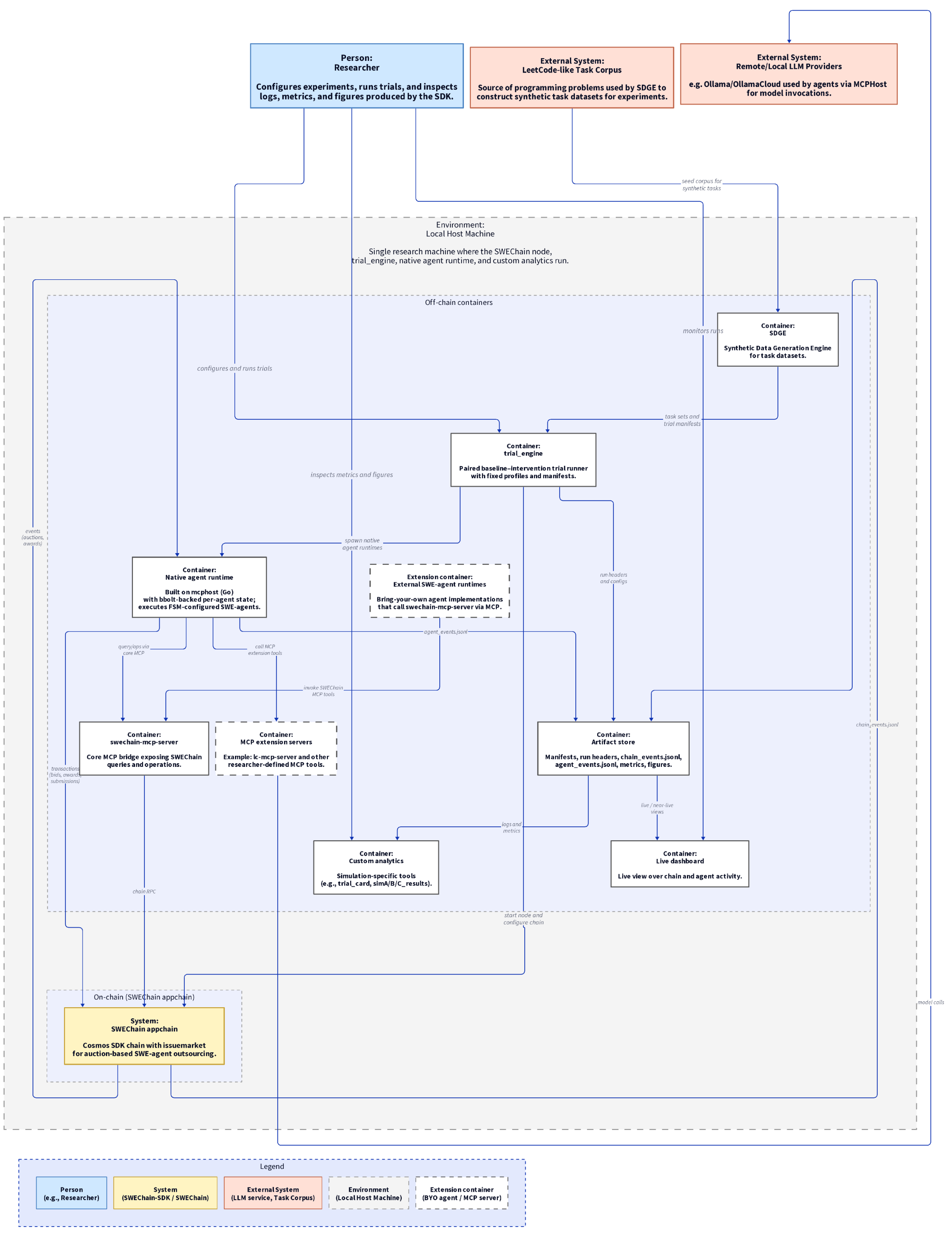

Person:
Researcher
Configures experiments, runs trials, and inspects logs, metrics, and figures produced by the SDK.
External System:
LeetCode-like Task Corpus
Source of programming problems used by SDGE to construct synthetic task datasets for experiments.
External System:
Remote/Local LLM Providers
e.g. Ollama/OllamaCloud used by agents via MCPHost for model invocations.
seed corpus for synthetic tasks
Environment:
Local Host Machine
Single research machine where the SWEChain node, trial_engine, native agent runtime, and custom analytics run.
Off-chain containers
monitors runs
Container:
SDGE
Synthetic Data Generation Engine for task datasets.
configures and runs trials
task sets and trial manifests
Container:
trial_engine
Paired baseline–intervention trial runner with fixed profiles and manifests.
inspects metrics and figures
events (auctions, awards)
spawn native agent runtimes
Container:
Native agent runtime
Built on mcphost (Go) with bbolt-backed per-agent state; executes FSM-configured SWE-agents.
Extension container:
External SWE-agent runtimes
Bring-your-own agent implementations that call swechain-mcp-server via MCP.
run headers and configs
agent_events.jsonl
query/ops via core MCP
call MCP extension tools
invoke SWEChain MCP tools
chain_events.jsonl
Container:
swechain-mcp-server
Core MCP bridge exposing SWEChain queries and operations.
Container:
MCP extension servers
Example: lc-mcp-server and other researcher-defined MCP tools.
Container:
Artifact store
Manifests, run headers, chain_events.jsonl, agent_events.jsonl, metrics, figures.
transactions (bids, awards, submissions)
live / near-live views
logs and metrics
Container:
Custom analytics
Simulation-specific tools (e.g., trial_card, simA/B/C_results).
Container:
Live dashboard
Live view over chain and agent activity.
chain RPC
start node and configure chain
On-chain (SWEChain appchain)
model calls
System:
SWEChain appchain
Cosmos SDK chain with issuemarket for auction-based SWE-agent outsourcing.
Legend
Person (e.g., Researcher)
System (SWEChain-SDK / SWEChain)
External System (LLM service, Task Corpus)
Environment (Local Host Machine)
Extension container (BYO agent / MCP server)

Figure 2 – SWEChain-SDK containers and responsibilities. Within a single local host environment, the SWEChain appchain provides the deterministic auction mechanism; off-chain containers implement SDGE, trial execution, native and external agent runtimes, MCP bridges, custom analytics, an artifact store, and a live dashboard. External services provide LLM inference and a LeetCode-like problem corpus.

To describe the structure of *SWEChain-SDK*, we adopt views in the spirit of the C4 model, moving from the system context in Figure 1 to a container-level view of the main runtime elements and then to component-level views of the SWEChain appchain and the native agent runtime used in the experiments.

At the context level, Figure 1 emphasizes three roles. The researcher, shown as a person node, configures experiments, runs trials, and inspects logs, metrics, and figures. SWEChain-SDK, shown as a system node, provides a local blockchain-based sandbox for SWE-Agent outsourcing experiments with paired baseline–intervention trials, SWE-Agents executed via a native agent runtime, and custom analytics. The environment node indicates that the entire system executes on a single research machine, which hosts the SWEChain node, the trial engine, the agent runtime, and the analytics tooling. Two external systems, the LLM services and the LeetCode-like task corpus, are connected as dependencies: the first provides model calls accessed through MCP tools, and the second provides problem data used by SDGE to construct synthetic task datasets.

At the container level, the SWEChain-SDK architecture separates the deterministic on-chain kernel from off-chain services and artifacts. Figure 2 provides this container view. The large dashed outline labeled "Environment: Local host machine" corresponds to the local host already introduced in the context diagram; within this environment, the figure distinguishes between an "On-chain: SWEChain node" region and an "Off-chain: SWEChain-SDK & experiments" region, and shows the researcher and external services at the top.

Inside the off-chain region, the SDGE container generates synthetic LeetCode-style task datasets and emits task sets and trial manifests. The *trial_engine* container consumes these manifests, initializes and configures the SWEChain node according to a given profile, and coordinates the execution of paired baseline and intervention runs. It is responsible for starting the node with a specific genesis file and configuration profile, waiting for the chain to reach a stable height, and launching the agent runtimes.

The *native agent orchestrator* container runs `swechain_agent` processes. It is built on top of `mcphost`, reads finite-state-machine (FSM) configurations and MCP-access configuration files, executes SWE-Agents according to these FSMs, and maintains per-agent state and journals in bbolt-backed files. During a trial, each native agent process obtains auction and chain state via MCP tools, calls additional tools as needed, and decides when to submit bids and solutions. The use of bbolt for local persistence does not change on-chain economics but makes it easier to inspect agent-level state and behavior; Appendix D includes an example of browsing these files with a TUI bbolt browser.

The figure shows two kinds of MCP server containers. The *swechain-mcp-server* is a core MCP bridge that wraps the `swechaind` command-line interface as tools for opening auctions, submitting bids, querying auction state, and similar operations. The *MCP extension servers* region collects two examples: a concrete `lc-mcp-server`, which exposes LeetCode-style tasks and metadata as tools, and an abstract "other MCP servers" node representing additional tools that could be contributed by other researchers. These extension containers are optional and live at the SDK's extension boundary: any MCP server that adheres to the same input/output contracts can be attached without modifying the chain.

The *External SWE-Agent runtimes* container indicates that researchers are not restricted to the native agent runtime. They may bring their own agent implementations—for example, written in other languages, using different controller architectures, or integrating with external lab infrastructures—and connect them to SWEChain-SDK via the same `swechain-mcp-server`. As long as these agents speak the same MCP tool protocols and respect the chain's transaction and event model, they can participate in the same auctions as the native agents and be evaluated under the same experimental conditions.

The custom analytics container represents simulation-experiment-specific tooling, such as the `trial_card` utility and the `simA_results`, `simB_results`, and `simC_results` programs used later in the dissertation. These tools read experiment logs (including agent journals) from the artifact store and emit metrics tables and figures. The artifact store itself is shown as a dedicated container: it holds manifests, run headers, agent event streams, chain event exports where needed, metrics tables, and generated figures, all under the researcher's control. A lightweight live dashboard container consumes the same streams and presents a near-real-time view of the current trial. Screenshots of the live dashboard interface are provided in Appendix D.

Within the on-chain region, the SWEChain appchain container is labeled as a system rather than a generic container to emphasize that it implements the deterministic mechanism under study. It is instantiated as a single-node Cosmos SDK application on the local host and exposes RPC, REST, and gRPC endpoints for queries and transactions. The appchain owns the on-chain state for accounts, balances, issues, bids, and settlements, and emits the events that drive the economic analysis. During development and debugging, a terminal block explorer is often attached to the running node to inspect blocks, transactions, and events; Appendix D includes a screenshot of such an explorer attached to a SWEChain simulation run.

At the bottom of Figure 2, the external services region mirrors the context diagram. LLM providers such as Ollama or OllamaCloud are shown as external systems that perform model calls on behalf of MCP tools, and the LeetCode-like task corpus is shown as the source corpus for SDGE synthetic tasks. The arrows from MCP servers to LLM providers indicate that some MCP tools perform LLM inference remotely rather than on the local machine; the arrow from the external corpus to SDGE indicates that SDGE is seeded from a pre-existing problem set but outputs synthetic datasets under fixed seeds.

The next two figures refine the container view by decomposing the SWEChain appchain and the native agent runtime into their main components.

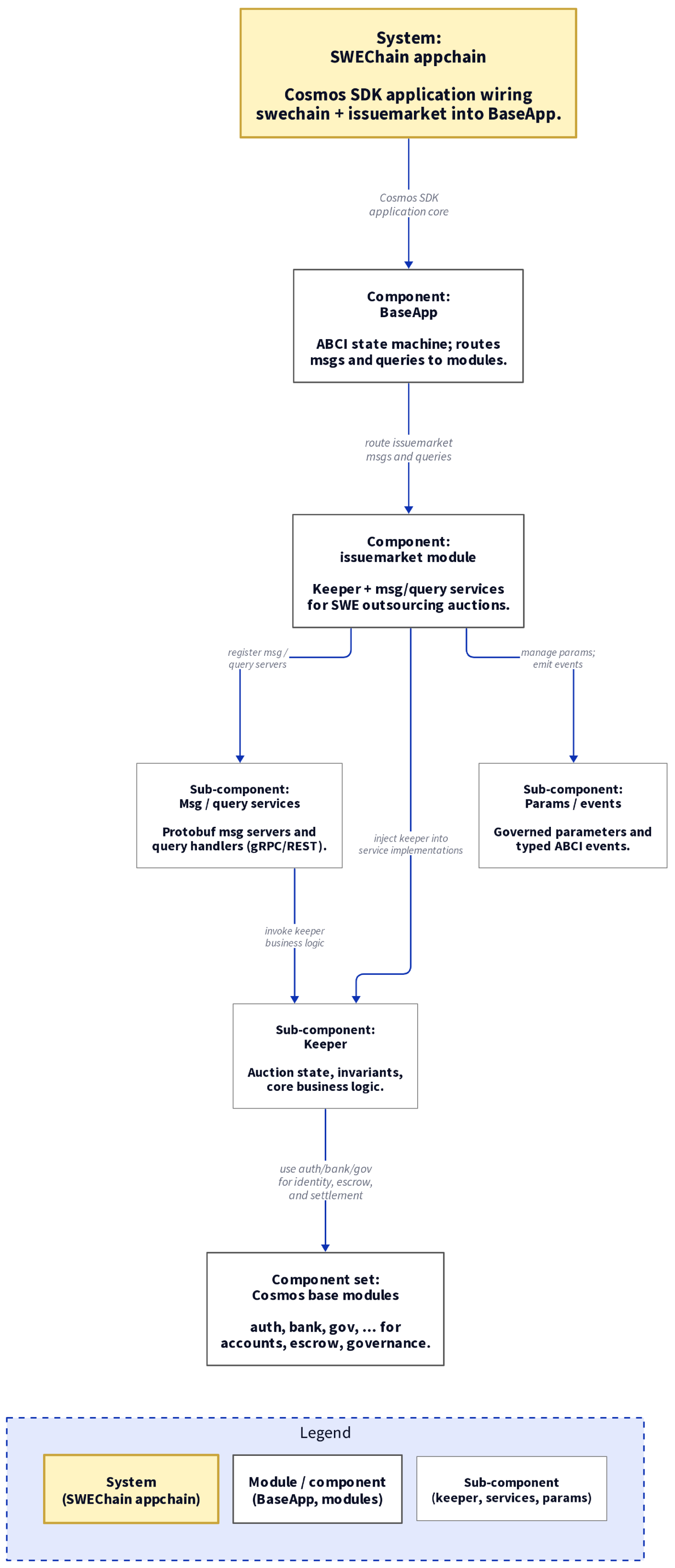


Figure 3 – SWEChain appchain components. The SWEChain application is a Cosmos SDK app that wires the `issuemarket` module into `BaseApp`, exposes messages and queries, and relies on base modules such as `auth`, `bank`, and `gov` for identity, balances, and governance.

Figure 3 focuses on the SWEChain appchain. The top-level system box represents the SWEChain application as a Cosmos SDK app that wires the `swechain` and `issuemarket` modules into `BaseApp`. `BaseApp` is responsible for routing transactions and queries through protobuf message and query services and for implementing the ABCI interface, while the `issuemarket` module owns auction state, enforces invariants, and coordinates module-level logic for opening issues, accepting bids, selecting winners, and settling payments. Message and query services provide the transaction and query endpoints for the issue market over gRPC and REST and invoke the keeper's business logic. Governed parameters capture policy (for example, auction duration, disclosure flags, scoring weights), and any bounded `BeginBlock`/`EndBlock` behavior can be configured through hooks. Typed ABCI events are emitted for issues, bids, awards, and settlements and can be consumed directly by analysis tools or exported from the node for downstream economic analysis. Base modules such as `auth`, `bank`, and `gov` provide identity, escrow, settlement, and governance functionality.

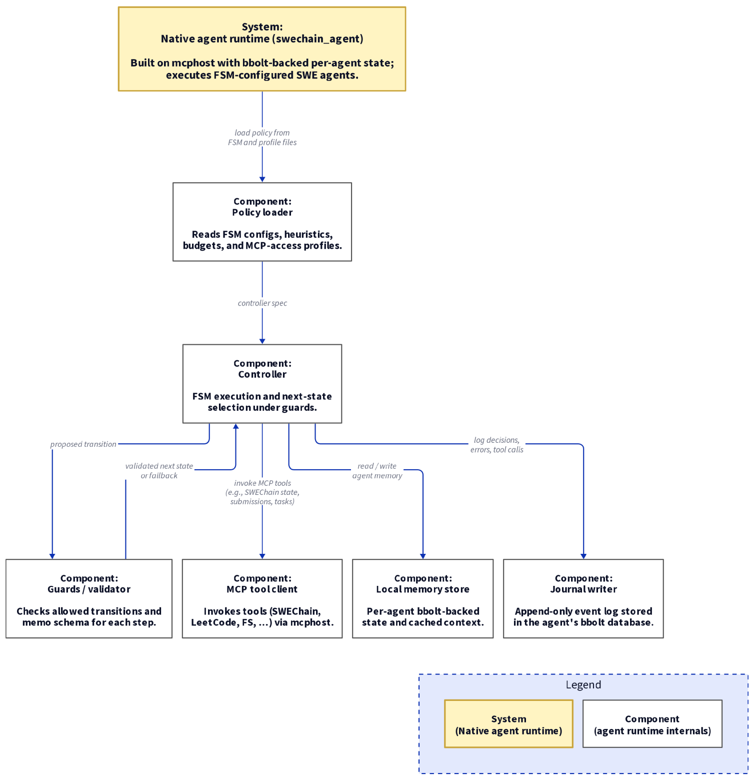

System:
Native agent runtime (swechain_agent)
Built on mcphost with bbolt-backed per-agent state; executes FSM-configured SWE agents.
load policy from FSM and profile files
Component: Policy loader
Reads FSM configs, heuristics, budgets, and MCP-access profiles.
controller spec
Component: Controller
FSM execution and next-state selection under guards.
proposed transition
validated next state or fallback
invoke MCP tools (e.g., SWEChain state, submissions, tasks)
read / write agent memory
log decisions, errors, tool calls
Component: Guards / validator
Checks allowed transitions and memo schema for each step.
Component: MCP tool client
Invokes tools (SWEChain, LeetCode, FS, ...) via mcphost.
Component: Local memory store
Per-agent bbolt-backed state and cached context.
Component: Journal writer
Append-only event log stored in the agent's bbolt database.
Legend
System (Native agent runtime)
Component (agent runtime internals)

Figure 4 – Native agent runtime components. The `swechain_agent` binary loads FSM-based policies, invokes MCP tools via `mcphost`, and maintains per-agent memory and journals in a local bbolt-backed store. The finite-state-machine structure constrains agent behavior while still allowing heterogeneous policies.

Figure 4 decomposes the native `swechain_agent` runtime used in the experiments. The agent runtime system loads policy files that describe finite-state-machine controllers, heuristics, and budgets, and MCP-access profiles that specify which MCP tools are available to each agent. A policy loader component translates these files into an internal controller specification. The controller then executes the finite-state machine step by step: given the current state, a view of local memory, and previous tool outputs, it proposes a next state. A guards/validator component checks that the proposed transition is allowed by the FSM graph and that any memo fields respect the expected schema; if a transition is not allowed, the validator forces a safe fallback state. This finite-state-machine mechanism is the main way in which SWEChain-SDK constrains agent behavior: every decision must correspond to a labeled state and a permitted transition, and only those states are allowed to trigger tool calls or transactions.

Within each step, the controller may read from or write to the local memory store, which is persisted in a bbolt-backed database as a single JSON object per agent, issue MCP tool calls via the MCP client (for example, to ask an LLM to propose a bid price, query SWEChain state through an MCP bridge, or attempt a code solution), and instruct a transaction-sending tool to build and submit bids or solutions. All decisions, tool calls, retries, and outcomes are recorded by the journal writer into an append-only journal stored in the same per-agent bbolt database; these structured event streams are later consumed by the analytics tools. Because the controller is driven by a fixed FSM and bounded heuristics, and because all randomness is either seeded or delegated to LLMs under fixed settings, the native agent runtime fits the architectural requirements for controlled experiments while still allowing heterogeneous policies and specializations across agents.

Although we do not include a separate sequence diagram, the runtime flow for a paired baseline–intervention trial can be summarized as follows. The researcher first prepares a manifest that specifies datasets, seeds, agent pool, and policy settings for baseline and intervention. The trial engine then initializes a SWEChain node with a specific genesis file and node profile, starts the SWEChain MCP server, the problem MCP server, and one or more native agent runtimes configured by their FSM and MCP-access files, and runs the baseline scenario to completion. During this run, SWE-Agents obtain auction state via MCP tools, update their local memory stores, call MCP tools from guarded FSM states, submit bids and solutions through MCP-exposed transaction tools, and write structured journals. The chain emits ABCI events that can be queried by tooling, and the agents emit journals that are written to the artifact store alongside a `run_header.json`. The system is then reset or reinitialized under identical conditions, except for the intended policy change (for example, specialization parameters, disclosure flags, or scoring weights), and the intervention scenario is run. External SWE-Agent runtimes, if present, can participate in the same flow by calling the `swechain-mcp-server` rather than being spawned by the native runtime.

Finally, analytics utilities read the paired artifacts and compute metrics and figures, including task completion, outsourcing cost, revenue concentration, and, where relevant, network-level statistics such as degree distributions or inequality indices. In each paired experiment, the baseline and intervention share the same manifests for datasets, seeds, and agent pool, and differ only in the policy dimension under study. The orchestration code, packaged under the `swechain-sdk` gateway repository, runs the baseline and intervention under identical node and build profiles, writing headers that pin the commit, genesis checksum, parameter snapshot, and configuration. This structure provides the architectural basis for the A/B/C simulations described in the abstract and supports R1, R2, and R7.

3.4 Configurability and Extensibility

The abstract describes *SWEChain-SDK* as configurable and extensible. This section makes those claims precise by outlining configuration surfaces and extension pathways in prose.

The SDK separates consensus-governed policy from experiment-level knobs and build profiles. Consensus-facing settings are typed and height-stamped on chain; orchestration and agent behavior are declared in file-backed manifests; build and node profiles pin toolchains and runtime tolerances.

On the on-chain side, configuration surfaces include `issuemarket` parameters such as auction duration, minimum decrement, reserve semantics, fee policy, disclosure switches, and scoring weights. These are intended to be governed by parameter-update messages and changed only by authorized actors, at known heights, with explicit parameter snapshots in exports. Genesis templates specify the chain identifier, initial accounts and balances, parameter baselines, and any pre-seeded state; they can be generated from templates and recorded with checksums in run headers. Node runtime configuration, such as RPC/REST/gRPC bindings, indexer settings, mempool options, minimum gas prices, and logging levels, is encapsulated in node and application configuration profiles that can be pinned per experiment profile.

On the off-chain side, configuration surfaces include trial manifests, agent policies, MCP profiles, build and toolchain profiles, and analytics profiles. Trial manifests fix seeds, task sets, specialization parameters, agent pool size and roles, and deadlines; they are file-backed, versioned, and hashed, so that the workload and policy for a run can be reconstructed. Agent policies specify controller topology (FSM or BT), guards, heuristics, budgets, and retry strategies; they are expressed as structured files and can be swapped without touching the chain. MCP profiles define endpoints, timeouts, concurrency limits, and tool allow/deny lists for each MCP server, with secrets injected via environment variables and the profile structure recorded as a file. Build and toolchain profiles pin versions of the core language runtime, libraries, and build tools (for example, Go, Cosmos SDK, CometBFT, and protocol buffer compilers) and are recorded via `go.mod`, continuous-integration workflows, and explicit versioning. Analytics profiles select metrics, binning schemes, and figure styles and formats and control how raw event and journal data are aggregated into tables and figures. Experiments are re-runnable: chain parameters are frozen for each trial, node profiles are fixed, and all settings are captured in manifests and run headers. The chain enforces escrow, award, and closure rules; orchestration stays declarative; tools are scoped per profile; and figures are regenerated from the same recorded inputs. Together, these mechanisms address R1–R3 and R6 by making configuration explicit and inspectable.

The SDK can evolve through well-bounded on-chain modules and off-chain plug-ins while maintaining comparability across experiments. On the on-chain side, new capabilities are introduced as explicit additions rather than as silent changes to existing behavior. For example, new market features are implemented as new modules or as new messages and queries that live alongside the existing ones, rather than by changing what the original messages do. Configuration switches and feature flags default to "off" so that simply upgrading the binary does not alter market behavior until policies are deliberately changed. When upgrades require changes to the structure of on-chain state, they are implemented as deterministic transformations that can be run and checked against archived chain states, so that historical experiments remain interpretable and comparable over time.

On the off-chain side, extension pathways include orchestrator plug-ins that operate as pure functions on manifests and emit additional logs or metrics without mutating chain state; new agent states and guards that expand the FSM or BT vocabulary while preserving determinism and recording topology snapshots; new MCP tools that introduce additional capabilities with stable input/output schemas, isolation, and resource caps; new analytics sinks that write additional metrics or figures under stable filenames and schemas without side effects on upstream artifacts; and new dashboards that consume existing event and metrics streams and render them differently without mutating chain or journal data. The native agent runtime and its bbolt-backed state, and the MCP-based block explorer and bbolt browsers illustrated in Appendix D, are examples of such extensions at the tooling layer.

New functionality in `issuemarket` is best introduced as additional messages, queries, and parameters rather than as silent changes to existing behavior. Winners, escrow, and closure semantics can remain invariant, while extensions such as dispute flows add new events and deadlines without altering basic bid and award behavior. Off chain, agent specialization and new tools expand capability without affecting the chain. Reproducibility under fixed experimental setups is supported by keeping seeds, manifests, and toolchains pinned and by validating changes with deterministic tests and artifact comparisons. These practices collectively address R6 while protecting R1–R3.

3.5 Discussion

The architecture of *SWEChain-SDK* operationalizes the conceptual framing of SWE-Agent Economics introduced earlier in the dissertation and provides the technical foundation for the simulation-based experiments presented in subsequent chapters. The public gateway repository `swechain-sdk` collects the appchain implementation, access bridges, agent runtimes, simulation drivers, dashboards, and analytics tooling described in this chapter into a single, versioned research artifact.

The architecture links intelligent software engineering and software-engineering economics through a configurable, extensible, and reproducible platform for studying decentralized, auction-based SWE-Agent outsourcing markets. Within the broader domain of SWE-Agent Economics, we focus on a specific and economically salient class of settings: blockchain-based SWE-Agent outsourcing auctions in which autonomous agents compete to complete software tasks under programmable market rules. Because issues, bids, allocations, and payments are all linked to agent identities in the chain state and logs, SWEChain-SDK naturally induces bipartite issue–agent graphs and derived network statistics (for example, degree distributions, assortativity, and revenue concentration) that are used in later chapters.

At a high level, the architecture turns this framing into a concrete experimental instrument. The Cosmos-based appchain encodes mechanism-level rules—admission, bidding, pricing, allocation, and settlement—as deterministic state-machine logic with governed parameters and typed events. Off-chain orchestration, agents, tools, dashboards, and analytics then act as programmable "lab equipment" around that mechanism: manifests define paired baseline–intervention runs under fixed seeds and datasets; agent controllers implement bidding and execution policies using finite-state machines and prompts; MCP servers expose problem-access and model-calling capabilities; and analytics programs turn experiment logs into metrics and figures. This division of responsibilities makes it feasible to realize the abstract requirement of a blockchain-native SDK for economic network simulations that can support controlled, paired baseline–intervention experiments under fixed conditions in decentralized SWE-Agent outsourcing markets.

Configurability and extensibility follow from the same principles. On-chain parameters capture market rules that are intended to be verifiable and governed—durations, fee and reserve policies, disclosure switches, scoring weights, and closure behavior—and can be changed at known heights. Experiment-level knobs for datasets, seeding, agent policies (including prompts), and tool profiles live in versioned files that can be edited, diffed, and reused. Extension points on both sides are explicit: on chain, new modules, messages, and hooks arrive as additions with migrations and event schemas; off chain, new samplers, agent states, MCP tools, dashboards, and analytics sinks compose against stable contracts. This makes it possible to introduce new SWE-Agent market variants or experimental policies while remaining within the same architectural envelope and retaining comparability with the experiments reported in this work.

Every state transition that matters to the study emits at least one typed event; every agent decision is logged as a structured journal entry; errors and timeouts are treated as data rather than silently swallowed. Figures in later chapters are therefore grounded in concrete files such as experiment logs, agent journals, manifests, and metrics tables. This makes it possible for other researchers, in principle, to rebuild the same plots, re-run paired trials under comparable conditions, or extend the experiments with modified policies using the same logs and manifests that back the evaluation-ready SDK release. Appendix D complements this chapter by illustrating the practical inspection tools used during development and experimentation.

The same architecture also admits a disciplined evolution path for the SDK itself. New agent behaviors, dashboards, or tooling integrations can be introduced off chain without compromising safety or reproducibility, provided their inputs and outputs adhere to existing schemas. Design patterns such as semantic versioning, explicit feature flags, and regression tests over artifacts are compatible with this architecture and can be used to manage change in a research tool that is expected to be reused and extended beyond this dissertation.

Finally, the architecture is intentionally neutral with respect to economic design within its design scope. *SWEChain-SDK* does not prescribe a single "correct" outsourcing mechanism or agent strategy; within the space of Cosmos-based auction markets with MCP-based SWE-Agents, it provides a concrete, blockchain-native SDK that makes it practical to implement different mechanisms, configure different policies (on chain and in agent controllers and prompts), and observe how SWE-Agents behave in decentralized outsourcing markets under controlled conditions derived from experiment logs. In this sense, the architecture helps connect the abstract domain of SWE-Agent Economics with the empirical simulations in the following chapters: it provides a concrete way to set up the paired baseline–intervention experiments described in the abstract, run them under fixed conditions, observe them in real time, and analyze them with detailed provenance.

Chapter **4**

SWEChain-SDK Simulation Experiments

We evaluate whether *SWEChain-SDK* supports controlled, paired baseline–intervention experiments under fixed conditions in decentralized Software-Engineering Agent (SWE-Agent) outsourcing markets. In all three experiments we run a baseline and an intervention on the same tasks, agents, seeds, and auction parameters; only one policy dimension varies. This matches the abstract's emphasis on blockchain-based SWE-Agent outsourcing markets, on a blockchain-native SDK for economic network simulations, and on paired experiments that highlight differences in task completion, outsourcing cost, and revenue concentration under fixed conditions.

Table 2 summarizes the three interventions and the primary outcome measures and visuals, aligned with the abstract's focus on task completion, outsourcing cost, and revenue concentration.

Table 2 – Simulation experiments, interventions, and main outcome measures.

Experiment	**Intervention**	**Metrics / visualizations**
A — Comparative advantage via agent specialization	Agents receive private specialization tags and small in-niche / out-of-niche cost multipliers (versus an all-generalist baseline).	Winner–task alignment (assortativity), assignment rate, revenue concentration; trial-card dashboards, agent finite-state machines, and baseline–intervention comparison plots.
B — Competitive bidding via bidders' price-signal utilization	Bidders' prompts explicitly instruct agents to use the current standing best admissible bid when deciding whether and how to bid.	Discount from reserve (level/dispersion), a simple convergence proxy, participation and competition indicators, revenue concentration; trial-card dashboards, bidder finite-state machines, and four-panel comparison plots.
C — Principal-weighted selection via quality-signal utilization	Principal replaces a price-only winner rule with a fixed price–quality scoring rule over structured bids.	Cost-to-completion, first-pass success, discount from reserve, participation, revenue concentration; trial-card dashboards, principal finite-state machines, and headline comparison plots.

Across all three experiments, baseline and intervention runs share the same tasks, agents, random seeds, and auction parameters; only the stated policy dimension is varied. The paired design is therefore intended to reduce confounding from incidental randomness and input changes, although the results remain single-case studies for the specific workloads considered rather than broad population estimates.

In all simulations, programmatic randomness is controlled through an explicit `random_seed` recorded in each trial's JavaScript Object Notation (JSON). The Synthetic Data Generation Engine (SDGE) generator distinguishes between an experiment-level base seed, supplied on the command line via `-seed`, and the per-trial seeds that are actually used during sampling. Internally, the seed for trial t is computed as `trialSeed = baseSeed + t`; for example, running SDGE with `-seed 1111` yields `random_seed = 1112` for `trial_001`, `1113` for `trial_002`, and so on. This offset is intentional: the base seed serves as a single handle for reproducing an entire experiment, while trial-specific seeds ensure that each trial has its own deterministic pseudo-random stream, even if downstream code were to forget to reseed. Given a fixed base seed, problem corpus, and configuration, SDGE therefore produces the same sequence of per-trial seeds, the same sampled task sets, the same agent pools (including specialization tags and accuracy profiles), and the same orchestration schedules. In the paired baseline–intervention design, the baseline and intervention manifests for a given trial index share the same per-trial seed, so all seeded randomness (for example, task selection and tag assignment) is aligned across the pair and the only intended difference is the policy under study (such as specialization or scoring). Randomness internal to Large Language Model (LLM) inference is not controlled by these seeds; instead, reproducibility under comparable conditions is supported by pinning model configurations and toolchains and by recording event-complete logs. This design allows a reviewer to verify that (i) a single base seed is sufficient to regenerate the full family of trials and (ii) within each baseline–intervention pair, seeded stochastic elements are synchronized, preserving the *ceteris paribus* structure of the experiments.

The SWEChain `issuemarket` environment faced by a single SWE-Agent has several properties that motivate the choice of metrics. From the agent's perspective, the environment is partially observable (agents see only the task description, a small history, and selected price signals, not the full global state), strategic (other agents are also optimizing bids, so outcomes depend on their behavior), sequential (bids and responses unfold over multiple steps), dynamic (the set of active auctions and bids evolves over time), discrete (actions and state changes occur at discrete blocks and steps), and inherently multi-agent. The trial-card and experiment-specific metrics are chosen to summarize how agents perform and interact in this kind of environment along three main dimensions: task completion, outsourcing cost, and revenue concentration.

Common experimental pipeline and tooling. All three simulation experiments rely on a common tool, `trial_card`, to extract and visualize economic and network outcomes from the blockchain. The goal is to give each run a single, reproducible summary that fixes the meaning of completion, cost, competition, and concentration across experiments and to use the same summary structure when comparing baseline and intervention conditions. For each experiment we therefore show four economic/network figures built from the canonical `trial_card` exports:

1. a baseline trial card;
2. an intervention trial card;
3. a paired trial-card comparison;
4. an experiment-specific "results" figure.

In addition, each experiment includes agent-level figures that show the finite-state machines (FSMs) executed by the relevant agents under baseline and intervention conditions. These FSMs clarify how the policy dimension is implemented at the control-flow and prompt level.

In its default compute mode, `trial_card` connects to the local SWEChain node and queries the `issuemarket` module for all auctions and bids at a chosen block height. It normalizes bid amounts, reconstructs the lowest admissible winning bid for each auction, and aggregates auction-level and bidder-level statistics. The tool writes a canonical JSON summary (`trial_card.json`) together with a single-run figure (`trial_card.svg`, converted to `trial_card.pdf`) that presents the main metrics in a compact dashboard.

Trial-card metrics and terminology

Because the same terminology appears in all trial-card and paired-comparison figures, this section summarizes the main quantities and how they are computed. Interpretations of high versus low values are discussed in the Simulation Results and Discussion sections for each experiment.

The trial-card *Volume* block reports how active the marketplace was in a run:

- ❏ *Auctions total* is the number of auctions instantiated.
- ❏ *Auctions with bids* counts auctions that received at least one bid.
- ❏ *Completed (with winning bid)* counts auctions that clear with a winner.
- ❏ *Completion rate* is the share of auctions that clear, that is, completed auctions divided by total auctions.

The *Auction competition* block characterizes how competitive the auctions are:

- *Bids total* counts all bids submitted in the run.
- *Mean bids per auction* averages the number of bids across all auctions.
- *Single- / multi-bid auctions* reports how many auctions have exactly one bid versus at least two bids, together with their shares.
- For multi-bid auctions, the *best–second gap* is the average difference between the winning price and the second-best price, and the *best–second ratio* is the average ratio of second-best to winning price.

The *Economic outcomes* block focuses on outsourcing cost and price dispersion:

- *Total outsourcing cost* is the sum of payments over all completed auctions in the run.
- *Mean cost per completed task* is total cost divided by the number of completed auctions.
- *Win price min/med/max* reports the minimum, median, and maximum winning prices.
- *Win price std dev / CV* report the standard deviation of winning prices and its coefficient of variation (the standard deviation divided by the mean winning price).

The *Network and participation* block reflects how agents share in the work:

- *Distinct bidders* counts how many agents submit at least one bid.
- *Distinct winning bidders* counts how many agents win at least one auction.
- *Bidders with at least one win* is the fraction of bidders who win at least once.
- *Unique bidder–auction pairs* counts bidder–auction combinations with at least one bid.
- *Mean distinct bidders per auction* is the average number of bidding agents per auction.
- *Mean auctions per bidder* is the average number of auctions on which an agent bids.
- *Mean wins per winning bidder* is the average number of wins among those bidders who win at least once.

The *Revenue concentration* block summarizes inequality in how revenue is distributed:

- ❑ Starting from each agent's total revenue (sum of payments over its wins), a *revenue Gini coefficient* is computed on a zero-to-one scale. A value of zero means perfectly equal revenue (all active agents earn the same); a value close to one means that almost all revenue goes to a single agent.

- ❑ The *revenue Herfindahl–Hirschman Index (Herfindahl–Hirschman Index (HHI))* sums squared revenue shares; higher values mean more concentration.

- ❑ *Top-1*, *Top-3*, and *Top-5 revenue shares* report the fraction of total revenue earned by the highest-earning agent, by the top three agents, and by the top five agents, respectively.

In these experiments, a revenue Gini reported as `0.000` usually has one of two interpretations: either the run has no completed auctions (so all agents earn zero, and revenue inequality is mechanically zero), or all winning agents receive exactly the same payment. Neither case is an error; both are boundary cases of perfectly equal revenue.

The paired baseline–intervention trial-card comparisons, generated in `-compare` mode, condense the same metrics into a side-by-side view: for each selected quantity, the card shows the baseline level, the intervention level, and their difference. For example, a paired card in Simulation C might report:

- ❑ Completion: B: 69.6%, I: 50.0%, $\Delta = -19.6$ percentage points;

- ❑ Mean cost per task: B: 22.05, I: 15.69, $\Delta = -6.36$ tokens;

- ❑ Revenue Gini: B: 0.343, I: 0.357, $\Delta = +0.014$;

- ❑ Revenue HHI: B: 0.0736, I: 0.0770, $\Delta = +0.0034$;

- ❑ Top-3 revenue share: B: 33.8%, I: 37.9%, $\Delta = +4.1$ percentage points.

Here "B" refers to the baseline run and "I" to the intervention. The differences are always reported as intervention minus baseline, so negative values mean the intervention is lower on that metric and positive values mean it is higher. The paired card format is reused across the three experiments; how to interpret high versus low values for the metrics that appear is discussed in the Simulation Results and Discussion sections for each experiment.

Taken together, these trial-card metrics provide a common vocabulary for the three simulation experiments. The Simulation Results and Discussion sections refer back to this vocabulary when interpreting baseline–intervention differences and headline figures for each policy dimension.

4.1 Simulation Experiment A — Comparative Advantage via Agent Specialization

Simulation A evaluates a minimal marketplace modification: each agent privately receives a small set of specialization tags (e.g., `DP`, `Graphs`) and is nudged—via prompts and small cost multipliers—to bid in those niches. The aim is to increase specialist–task alignment (who wins what) without lowering the auction clearance rate. This operationalizes comparative advantage in autonomous software-work routing while holding auction rules and infrastructure fixed.

4.1.1 Experimental Design

We ask whether private, agent-side specialization increases winner–task assortativity while keeping the assignment rate at least as high as a seed-matched baseline with no specialization. In the baseline, agents act as generalists: they see no private tags and face neutral cost multipliers. In the intervention, each agent i is assigned a private tag set S_i; prompts explicitly encourage bidding on tasks whose primary tag lies in S_i; and the simulator applies a small in-niche cost discount with a small out-of-niche penalty. All other marketplace rules—timing, eligibility, fees, and clearing logic—remain identical.

Matching quality is summarized by a categorical assortativity coefficient r over a winner-bucket$\times$task-tag contingency table $\{e_{kl}\}$ with margins $a_k = \sum_l e_{kl}$ and $b_l = \sum_k e_{kl}$:

$$r \;=\; \frac{\sum_k e_{kk} - \sum_k a_k b_k}{1 - \sum_k a_k b_k}.$$

Here the winner bucket equals a task's primary tag if the winner's private tags include it, and `Other` otherwise. Thus $r \approx 1$ indicates strong tag-level alignment (winners typically have tags that match the tasks they win), $r \approx 0$ resembles chance given the observed base rates, and $r < 0$ signals systematic mismatch (winners tend to lack the relevant tags). This metric is used because the central question in Simulation A is whether specialization helps route work to more suitable agents; assortativity provides a single, interpretable summary of that alignment.

Participation is tracked via the assignment rate (auctions with a winner divided by auctions launched), together with summary statistics on competition (bids per auction, share of multi-bid auctions, and distinct bidders) and realized outsourcing cost (mean cost per completed task). Revenue concentration is summarized by the Gini and HHI of agent revenue and by the revenue share of the top winners, using the definitions introduced earlier in the chapter. These metrics are chosen because they translate the qualitative idea of "specialists doing the right tasks" into concrete questions: does specialization increase or reduce coverage, how much more or less do buyers pay, and does specialization spread work across more agents or concentrate wins in a few specialists?

Agent control structure and specialization prompts

The specialization intervention is implemented at the agent controller level, using a finite-state machine that orchestrates perception, bidding, and error handling. In both baseline and intervention, agents follow a similar control structure: they receive a task, interpret it, decide whether to bid, and either submit a structured bid or decline. What changes is how specialization information is injected into that process.

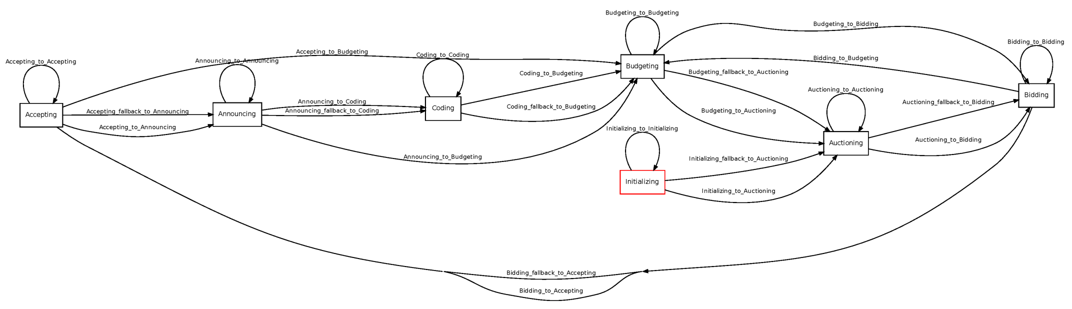


Figure 5 – Finite-state machine for the baseline agents in Simulation A. Control flow covers task intake, generic reasoning, bidding, submission, and error handling, with no specialization tags or niche-specific prompts.

Figure 5 shows the finite-state machine executed by the baseline agents. The main states handle task intake, reasoning, bidding, submission, and error recovery. The bidding state is parameterized by a prompt that summarizes the task and principal but treats all tasks symmetrically: there are no private tags and no explicit instructions to favor particular task types. Cost reasoning is generic and does not depend on any notion of niche.

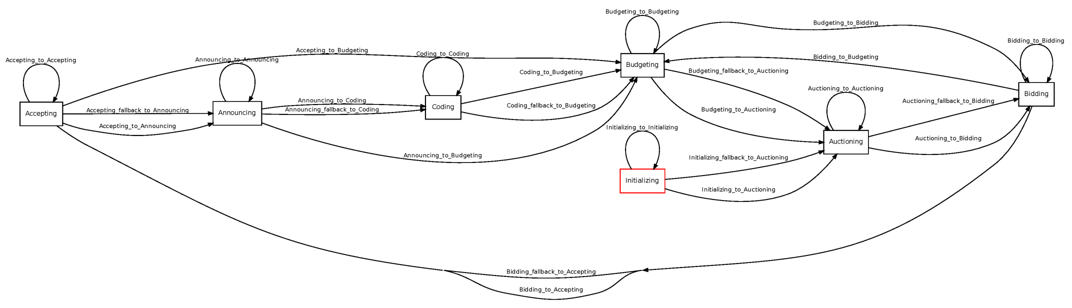
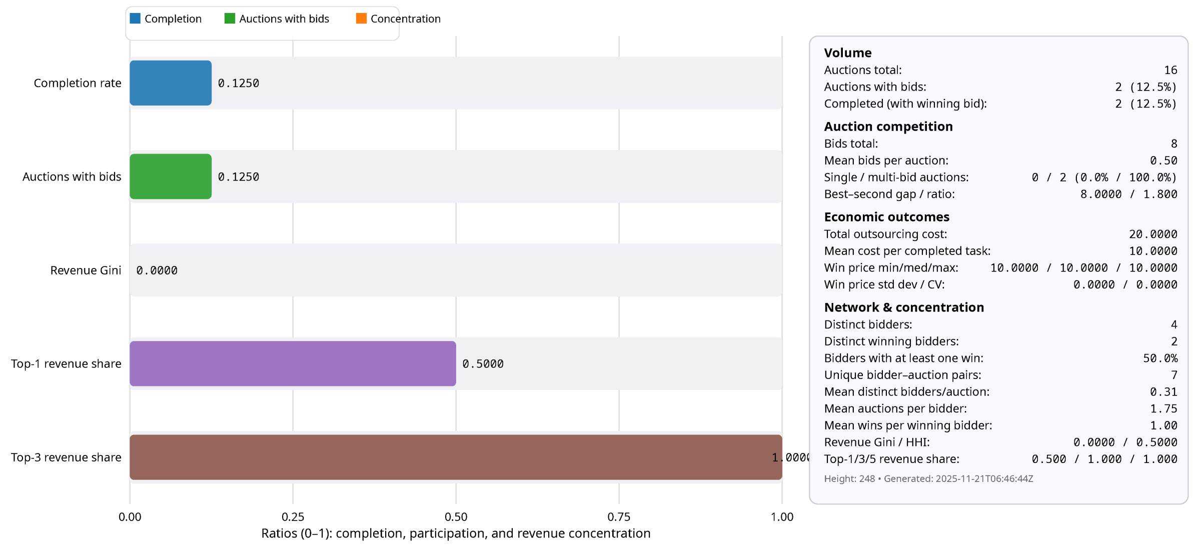


Figure 6 – Finite-state machine for specialization-enabled agents in Simulation A. The overall control structure mirrors the baseline, but the bidding state now accesses private tag sets and niche-sensitive cost multipliers, and the prompt explicitly encourages in-niche participation.

Figure 6 shows the corresponding finite-state machine for the specialization-enabled agents. The high-level control flow is kept intentionally similar to the baseline, but the bidding path now includes access to each agent's private tag set S_i and to small in-niche and out-of-niche cost multipliers. The bidding prompt includes an explicit block that lists the agent's tags, reminds the agent that tasks whose primary tag matches S_i are likely to be good fits, and asks the agent to consider these tags when deciding whether to bid and how to set the price. The cost reasoning state incorporates the multipliers so that in-niche bids tend to be cheaper and out-of-niche bids slightly more expensive, relative to the unspecialized baseline.

By presenting these two FSMs side by side, the chapter makes clear that Simulation A varies a specific component of agent behavior: the presence of private specialization tags and associated prompt and cost adjustments. Other aspects of the controller, such as retry logic and error handling, remain aligned between baseline and intervention.

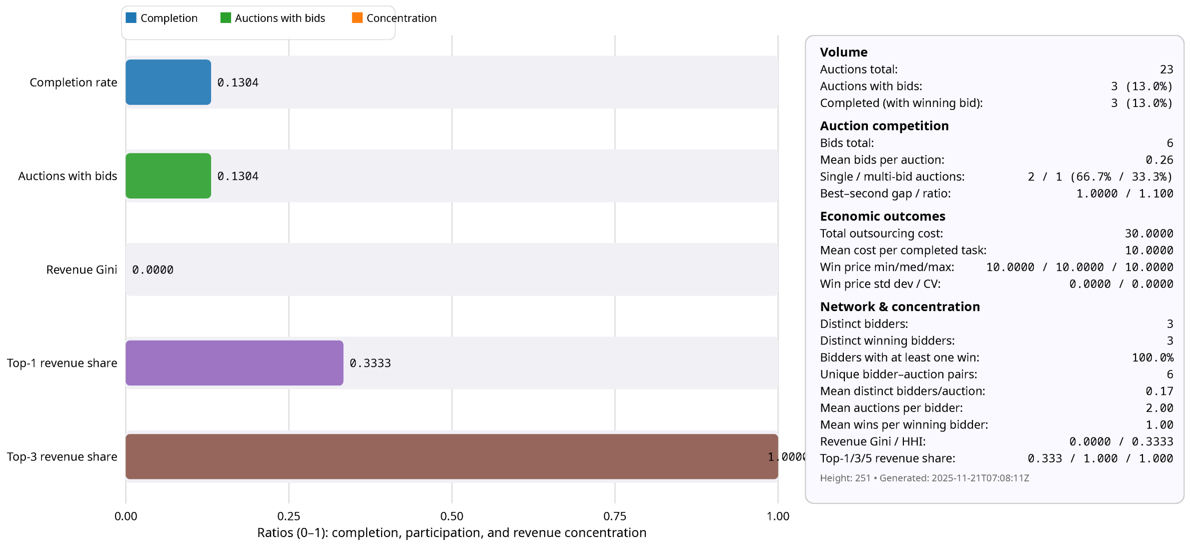
4.1.2 Simulation Results and Discussion



Volume	
Auctions total:	16
Auctions with bids:	2 (12.5%)
Completed (with winning bid):	2 (12.5%)
Auction competition	
Bids total:	8
Mean bids per auction:	0.50
Single / multi-bid auctions:	0 / 2 (0.0% / 100.0%)
Best–second gap / ratio:	8.0000 / 1.800
Economic outcomes	
Total outsourcing cost:	20.0000
Mean cost per completed task:	10.0000
Win price min/med/max:	10.0000 / 10.0000 / 10.0000
Win price std dev / CV:	0.0000 / 0.0000
Network & concentration	
Distinct bidders:	4
Distinct winning bidders:	2
Bidders with at least one win:	50.0%
Unique bidder–auction pairs:	7
Mean distinct bidders/auction:	0.31
Mean auctions per bidder:	1.75
Mean wins per winning bidder:	1.00
Revenue Gini / HHI:	0.0000 / 0.5000
Top-1/3/5 revenue share:	0.500 / 1.000 / 1.000

Height: 248 • Generated: 2025-11-21T06:46:44Z

Figure 7 – Simulation A baseline configuration (generalist agents). Canonical trial-card summary for a single run under the fixed SimA workload. The baseline clears almost no auctions, so realized cost and revenue concentration are effectively zero.

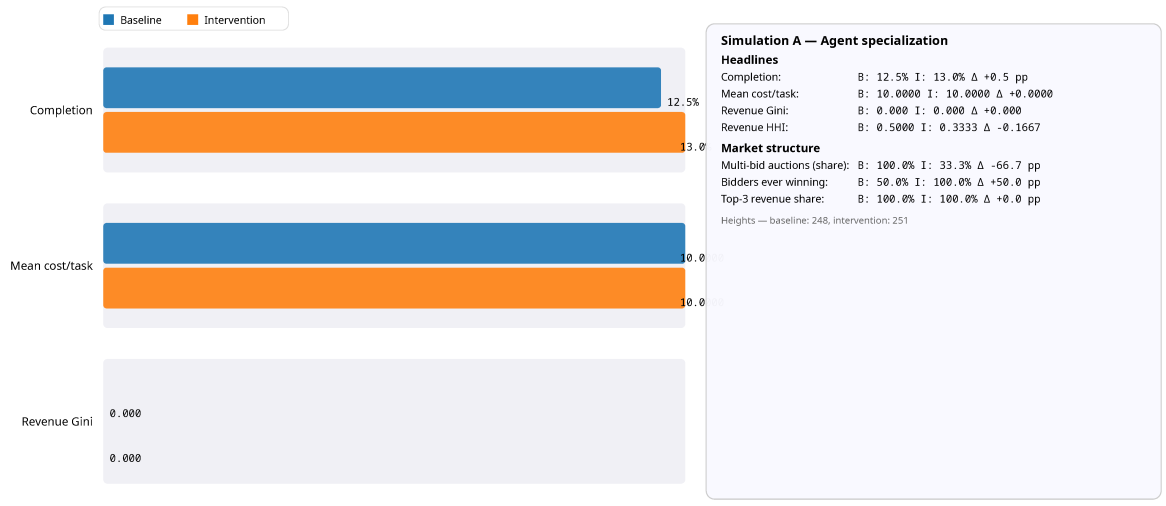


```
Volume
Auctions total:                                   23
Auctions with bids:                        3 (13.0%)
Completed (with winning bid):              3 (13.0%)
Auction competition
Bids total:                                        6
Mean bids per auction:                          0.26
Single / multi-bid auctions:     2 / 1 (66.7% / 33.3%)
Best–second gap / ratio:               1.0000 / 1.100
Economic outcomes
Total outsourcing cost:                      30.0000
Mean cost per completed task:                10.0000
Win price min/med/max:   10.0000 / 10.0000 / 10.0000
Win price std dev / CV:                0.0000 / 0.0000
Network & concentration
Distinct bidders:                                  3
Distinct winning bidders:                          3
Bidders with at least one win:                100.0%
Unique bidder–auction pairs:                       6
Mean distinct bidders/auction:                  0.17
Mean auctions per bidder:                       2.00
Mean wins per winning bidder:                   1.00
Revenue Gini / HHI:                    0.0000 / 0.3333
Top-1/3/5 revenue share:          0.333 / 1.000 / 1.000
Height: 251 • Generated: 2025-11-21T07:08:11Z
```

Figure 8 – Simulation A intervention configuration (specialization-enabled agents with private tags and cost multipliers). Compared to Figure 7, more auctions attract bids, some auctions clear, realized outsourcing cost becomes positive, and revenue is no longer degenerate.

Figures 7 and 8 show the single-run economic and network trial cards for the baseline and specialization-enabled configurations under the same tasks, agents, and seeds. In the baseline trial card, agents behave as generalists and rarely commit to numeric bids. Most auctions either receive no admissible bids or only non-executable content, leading to a completion rate close to zero, negligible realized outsourcing cost, and zero revenue concentration by construction (all agents earn zero, so the revenue Gini is `0.000`). Network-side metrics reflect the same picture: there are few bidder–auction pairs, low mean bids per auction, and almost no observed winner activity.

Under specialization (Figure 8), the pattern shifts. The combination of entry nudges and mild in-niche discounts increases the number of auctions with bids and produces a non-trivial share of cleared auctions. Mean cost per completed task becomes positive, and revenue concentration quantities such as the Gini, HHI, and top-k revenue shares become meaningful: a small set of agents now earn the available revenue. Participation metrics also become more informative: additional bidder–auction pairs are observed, and a subset of bidders converts bids into wins, increasing the share of bidders with at least one win.



Simulation A — Agent specialization

Headlines

```
Completion:                  B: 12.5% I: 13.0% Δ +0.5 pp
Mean cost/task:              B: 10.0000 I: 10.0000 Δ +0.0000
Revenue Gini:                B: 0.000 I: 0.000 Δ +0.000
Revenue HHI:                 B: 0.5000 I: 0.3333 Δ -0.1667
```

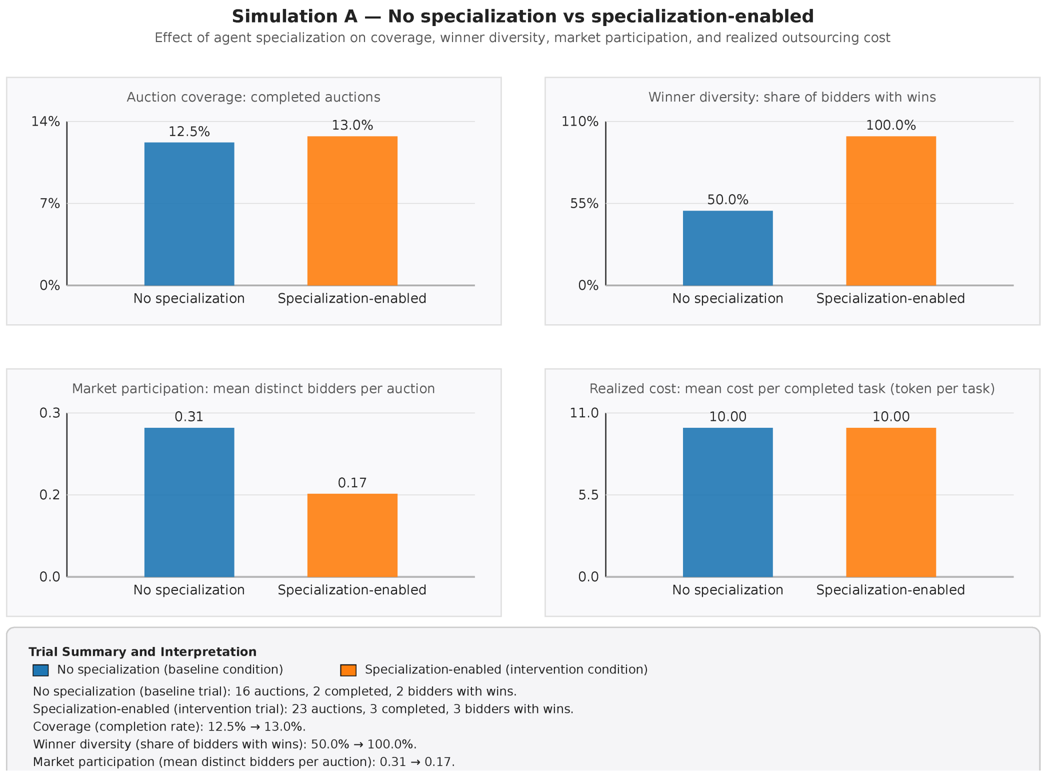
Market structure

```
Multi-bid auctions (share):  B: 100.0% I: 33.3% Δ -66.7 pp
Bidders ever winning:        B: 50.0% I: 100.0% Δ +50.0 pp
Top-3 revenue share:         B: 100.0% I: 100.0% Δ +0.0 pp
```

Heights — baseline: 248, intervention: 251

Figure 9 – Simulation A paired comparison. Baseline and intervention levels and differences on coverage, competition, cost, and revenue concentration under identical tasks, agents, and seeds.

The paired comparison (Figure 9) summarizes these changes as level-versus-delta panels. For this workload and time budget, specialization moves the system from a degenerate baseline (almost no completions) to a regime where a modest fraction of tasks clears. As soon as tasks begin to clear, realized outsourcing cost and revenue concentration cease to be trivially zero: buyers now pay positive amounts for completed work, and those payments accrue to the small set of agents who win. In other words, specialization is associated with more work getting done, with the resulting revenue flowing to the subset of agents that successfully operate in their niches.



No specialization (baseline trial): 16 auctions, 2 completed, 2 bidders with wins.
Specialization-enabled (intervention trial): 23 auctions, 3 completed, 3 bidders with wins.
Coverage (completion rate): 12.5% → 13.0%.
Winner diversity (share of bidders with wins): 50.0% → 100.0%.
Market participation (mean distinct bidders per auction): 0.31 → 0.17.

Figure 10 – Simulation A headline results. Four-panel comparison of the baseline (generalist agents) and the intervention (specialization-enabled agents) on auction coverage (completion rate), winner diversity (fraction of bidders with at least one win), mean distinct bidders per auction (participation), and mean cost per completed task.

Finally, the headline Simulation A figure (Figure 10) collects four run-level metrics that summarize the effect of specialization for this workload. In this figure, auction coverage is the completion rate, the share of launched auctions that clear with a winning bid; higher coverage means more tasks are successfully assigned. Winner diversity is the share of bidders with at least one win; higher winner diversity means that revenue is spread across more of the active bidders rather than being captured by just one or two specialists. Market participation is the mean number of distinct bidders per auction; higher participation indicates that, on average, more agents are willing to compete for each task. Realized cost is the mean cost per completed task in tokens, so higher values mean the buyer pays more on average for each completed task. These metrics are appropriate for Simulation A because the central policy lever is specialization: we want to know whether specialization helps more tasks clear, whether it pulls in more agents, whether it spreads or concentrates wins, and what cost buyers pay for these changes.

Taken together, Figures 7–10 suggest a simple interpretation for this particular paired run. First, specialization is associated with improved coverage: auctions that would otherwise remain idle now attract admissible bids and clear. Second, the distribution of wins broadens: a larger fraction of active bidders obtain at least one win, which shows up as higher winner diversity and more bidder–auction pairs than in the trivial baseline. Third, realized outsourcing cost becomes positive rather than trivially zero, which is expected once work is actually being performed. Fourth, revenue concentration is non-zero but mechanically sensitive to the small number of completed auctions: when only a few tasks clear, any agent who wins one of them captures a large revenue share. These observations are conditional on the specific configuration and workload; they illustrate how SWEChain-SDK can be used to probe the implications of a specialization policy under controlled but necessarily simplified conditions.

4.1.3 Threats to Validity and Limitations

The main threat to validity in Simulation A is the small effective sample size of completed auctions in the baseline and intervention runs. When only a handful of tasks clear, summary statistics such as the winner–task assortativity r, the Gini of revenue, or the share of bidders with wins become noisy and highly sensitive to individual auctions. We partially mitigate this by working with a seed-matched baseline–intervention pair and by publishing the underlying contingency tables and trial cards so that alternative statistics can be recomputed, but the small number of completions remains a limitation.

A second limitation is the stylized nature of both tasks and agents. Specialization tags are assigned rather than learned; agent policies are finite-state and prompt-driven; and the cost structure is simplified to highlight a single mechanism. This makes Simulation A a useful test bed for the SDK's experimental pipeline, but it also means that the quantitative effect sizes should not be read as direct estimates for production systems.

Third, the short time budget and fixed agent pool restrict the range of equilibrium behaviors that can arise. For instance, we do not model long-run adaptation, tag reallocation, or agents strategically reshaping their specialization to chase profitable tags. As a result, Simulation A underestimates dynamic phenomena such as agents drifting into dominant niches or principals adapting their posting behavior.

Finally, the winner inference rules and tag mappings introduce their own modeling choices. When explicit winner events are missing but valid numeric bids exist, we infer winners by selecting the lowest admissible bid; alternative tie-breaking or quality-weighted rules could produce different concentration patterns. The tag hierarchy and the mapping from raw task descriptions to primary tags are also simplified. These are acceptable compromises for a first experiment and for achieving reproducibility within the chosen environment, but they limit external validity. Subsequent experiments in the dissertation revisit some of these design choices under richer workloads and longer runs.

4.2 Simulation Experiment B — Competitive Bidding via Bidders' Price-Signal Utilization

Simulation B studies how bidder-facing prompts that explicitly call out the standing best admissible bid relate to pricing, competition, and participation when all other marketplace elements are held fixed. We compare a seed-matched baseline–intervention pair of runs in which bidders see the same task and principal fields and the same best-bid field in their input, but only the intervention prompt instructs them to take the current best bid into account when deciding whether and how to bid. The agent pool, effective costs (including any specialization), auction rules, and payment logic remain constant across the two conditions.

4.2.1 Experimental Design

The main question is whether prompting bidders to explicitly consider the current best admissible bid, on top of the usual task and principal information, is associated with tighter pricing and different competitive behavior, and how these changes relate to task completion.

All bidding agents share a common controller and JSON output schema. For each auction, the agent receives a structured description of the task (identifier, deadline, required skills, and a coarse difficulty estimate), basic principal-history fields (for example, past acceptance and repair rates), and the current standing best admissible bid when one exists. The chain and event stream expose the same fields in both arms; the only difference is how strongly the prompt tells the agent to use the best-bid information.

We define two regimes. In the baseline (*best-bid neutral*) condition, the prompt describes the task and principal and instructs the agent to propose a bid based on its private costs, the deadline, and principal reputation. The current best admissible bid is present in the input schema but is not highlighted in the system prompt and is not mentioned in the instructions, and the agent is not explicitly asked to react to it. In the intervention (*best-bid attentive*) condition, the same task and principal fields and best-bid field are provided, but the system prompt includes a dedicated section for the current best admissible bid and explicitly instructs the agent to take it into account: the agent is asked to assess whether the current best bid looks plausible given the task and principal, and then to decide whether to undercut, match, or abstain given its own cost structure and risk tolerance. The intervention is therefore purely prompt-level: both arms have access to the same information, but only the intervention makes the best-bid field salient and normatively relevant for bidding.

Beyond raw prices, we instrument two derived quantities for each auction. A buyer-discount-from-reserve score

$$s_a = \frac{p_a^{\text{clear}} - p_a^{\text{reserve}}}{p_a^{\text{reserve}}}$$

captures how far the clearing price p_a^{clear} falls below a coarse reserve p_a^{reserve} implied by task difficulty. Here $s_a < 0$ means the buyer pays less than the reserve (a discount), $s_a = 0$ means the buyer pays exactly the reserve, and $s_a > 0$ would mean paying above reserve. Smaller (more negative) values of s_a correspond to better prices for the buyer, and lower dispersion of s_a across auctions indicates more predictable pricing. This metric is used in Simulation B because the treatment is all about how agents react to price signals; *discount from reserve* provides a normalized way to compare prices across tasks with different implied costs.

A simple convergence proxy

$$\tau_a = t_a^{\text{close}} - t_a^{\star}$$

measures the time between the first appearance of the final clearing price $t_a^{\star}$ and the auction close t_a^{close}. Smaller τ_a means the auction effectively "settles" earlier relative to close, while larger τ_a indicates that prices continue to change closer to the deadline. This proxy is included because one practical goal of using price signals is to stabilize auctions sooner, reducing unnecessary bidding late in the process.

Guardrail metrics mirror the chapter-level focus on participation and concentration. We track the number of distinct bidders per auction, the assignment rate (fraction of auctions with a winner), and the concentration of wins across agents via the Herfindahl–Hirschman Index (HHI) and top-k revenue shares. Task-level acceptance and rework metrics are derived from the events stream in the same way as in Simulation A. As in Simulation A, baseline and intervention use the same tasks, agents, random seeds, timing budgets, and auction parameters. The resulting paired differences in the reported metrics are therefore at least consistent with the prompt-level treatment of best-bid information playing a central role, although they should still be interpreted as specific to this workload and configuration.

Bidder finite-state machines and price signals

Agent behavior in Simulation B is controlled by two agent specification files, `agent_B_baseline.json` and `agent_B_intervention.json`, stored under `simB/config/` in the repository. The two specifications are deliberately kept as close as possible: they share the same finite-state machine, retry policies, memory filters, and language-model sampling configuration. The main design choice is how the current best admissible bid is surfaced and used inside the bidding state.

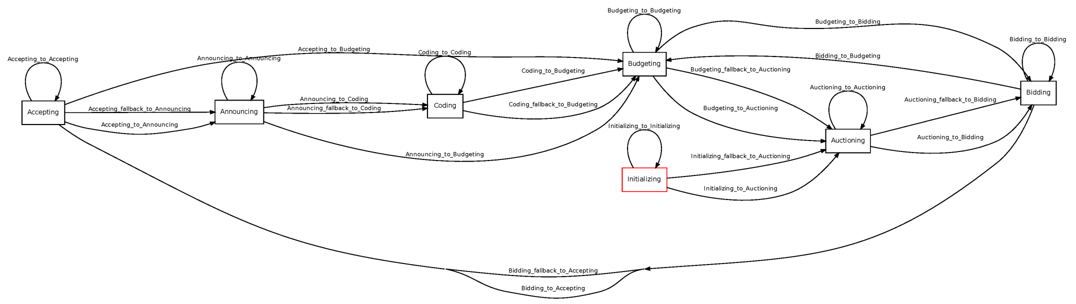


Figure 11 – Finite-state machine for the baseline bidder in Simulation B. The current best admissible bid is present in the structured input but is not emphasized in the bidding prompt, which focuses on costs, deadlines, and principal reputation.

Figure 11 shows the finite-state machine executed by the baseline bidder. States for task intake, reasoning, bidding, submission, and error handling are the same as in the other experiments, and the bidding state is parameterized by a prompt that summarizes the task, principal, and historical context without highlighting the current best admissible bid as a decision driver. The best-bid value is present in the structured input but is treated as an ordinary field rather than a focal piece of information.

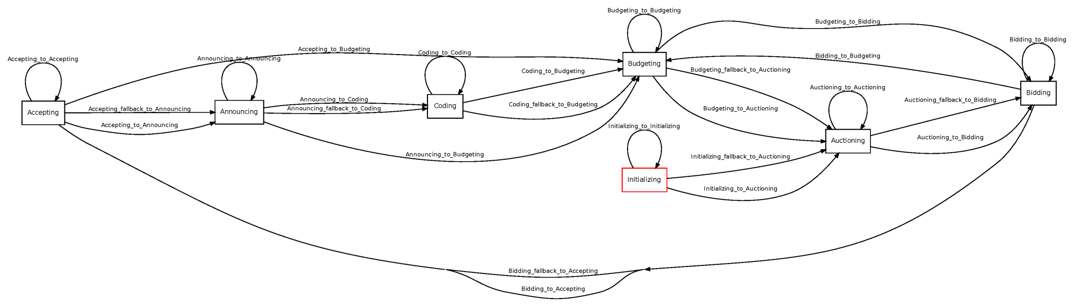


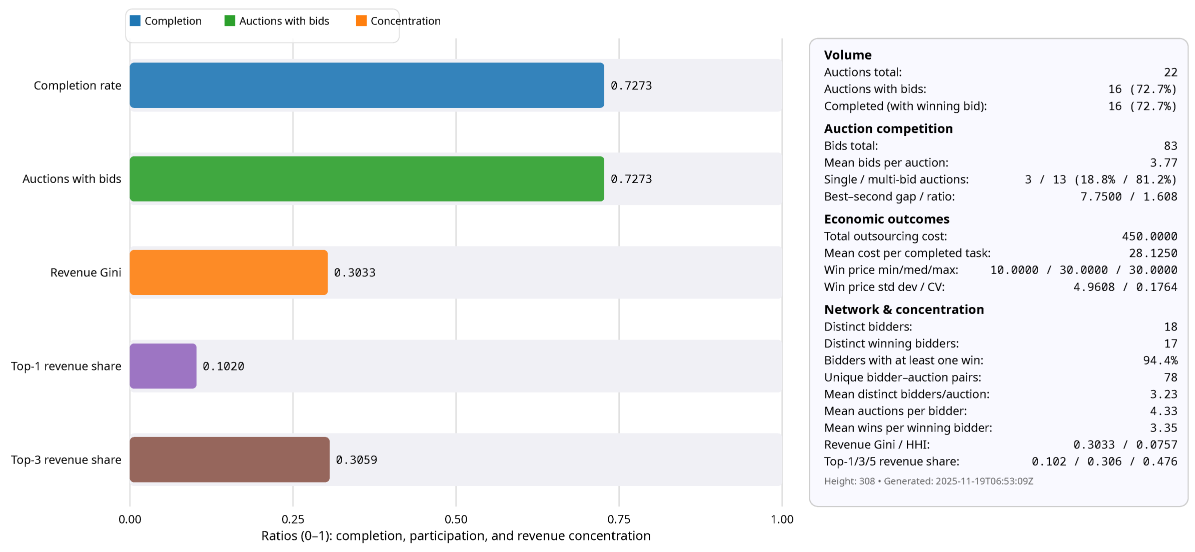
Figure 12 – Finite-state machine for the intervention bidder in Simulation B. The control flow is identical to the baseline, but the bidding state includes an explicit price-signal block that makes the current best admissible bid salient in the prompt and asks the agent to react to it.

Figure 12 shows the corresponding finite-state machine for the intervention. The control-flow structure is intentionally identical, but the bidding state now includes a dedicated price-signal block that surfaces the current best admissible bid together with explicit instructions: the agent is told to consider whether the best bid is plausible given the task and principal, and then to decide whether to undercut, match, or abstain. In this sense, the intervention does not add new information; instead, it changes how existing information is framed and prioritized in the bidding logic.

By keeping the finite-state machines and sampling configuration fixed and changing only the contents of the bidding prompts, the experiment is designed to highlight the role of price-signal utilization. Any paired differences in completion, outsourcing cost, competition, or revenue concentration between the two runs are consistent with that design choice being an important driver, but they should still be read as exploratory evidence rather than as definitive causal estimates.

Trial execution follows the shared pattern introduced in Simulation A. The task set and agent pool are prepared; the auction configuration, runtime budget, and random seeds for task order, bid timing, and tie-breaks are fixed; the baseline run is executed and per-auction and run-level metrics are computed; the intervention run is executed under the same seeds and inputs; and auction pairs are then formed and compared. Prompt templates and CLI commands for this experiment are documented in Appendix B, together with a reproducibility protocol that reconstructs the trial artifacts and figures.

4.2.2 Simulation Results and Discussion



Volume

Auctions total:	22
Auctions with bids:	16 (72.7%)
Completed (with winning bid):	16 (72.7%)

Auction competition

Bids total:	83
Mean bids per auction:	3.77
Single / multi-bid auctions:	3 / 13 (18.8% / 81.2%)
Best–second gap / ratio:	7.7500 / 1.608

Economic outcomes

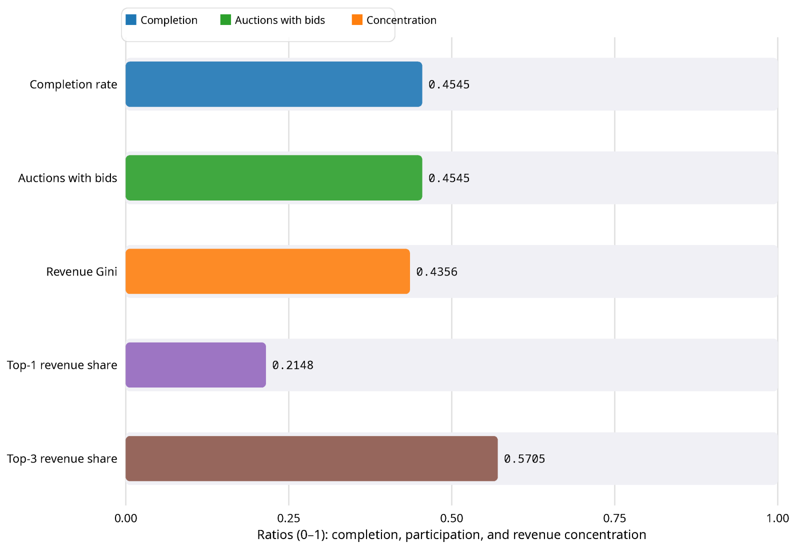
Total outsourcing cost:	450.0000
Mean cost per completed task:	28.1250
Win price min/med/max:	10.0000 / 30.0000 / 30.0000
Win price std dev / CV:	4.9608 / 0.1764

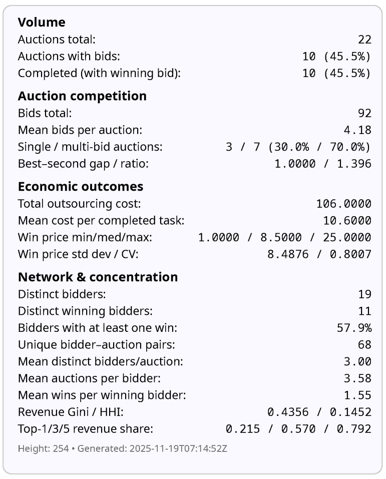
Network & concentration

Distinct bidders:	18
Distinct winning bidders:	17
Bidders with at least one win:	94.4%
Unique bidder–auction pairs:	78
Mean distinct bidders/auction:	3.23
Mean auctions per bidder:	4.33
Mean wins per winning bidder:	3.35
Revenue Gini / HHI:	0.3033 / 0.0757
Top-1/3/5 revenue share:	0.102 / 0.306 / 0.476

Height: 308 • Generated: 2025-11-19T06:53:09Z

Figure 13 – Simulation B baseline configuration (best-bid neutral prompts). Canonical trial-card summary for a single run under the fixed SimB workload.





Volume

Auctions total:	22
Auctions with bids:	10 (45.5%)
Completed (with winning bid):	10 (45.5%)

Auction competition

Bids total:	92
Mean bids per auction:	4.18
Single / multi-bid auctions:	3 / 7 (30.0% / 70.0%)
Best–second gap / ratio:	1.0000 / 1.396

Economic outcomes

Total outsourcing cost:	106.0000
Mean cost per completed task:	10.6000
Win price min/med/max:	1.0000 / 8.5000 / 25.0000
Win price std dev / CV:	8.4876 / 0.8007

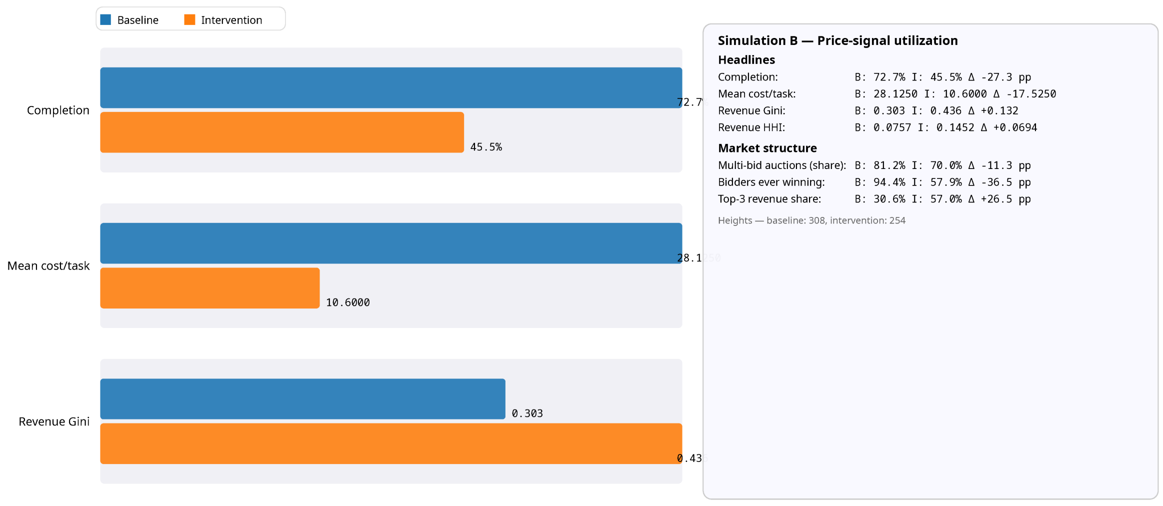
Network & concentration

Distinct bidders:	19
Distinct winning bidders:	11
Bidders with at least one win:	57.9%
Unique bidder–auction pairs:	68
Mean distinct bidders/auction:	3.00
Mean auctions per bidder:	3.58
Mean wins per winning bidder:	1.55
Revenue Gini / HHI:	0.4356 / 0.1452
Top-1/3/5 revenue share:	0.215 / 0.570 / 0.792

Height: 254 • Generated: 2025-11-19T07:14:52Z

Figure 14 – Simulation B intervention configuration (best-bid attentive prompts that explicitly expose and highlight the current standing best admissible bid). Canonical trial-card summary for the paired intervention run.

Figures 13 and 14 present the single-run economic and network trial cards for the baseline and intervention conditions. Each card summarizes the number of auctions and bids, completion and assignment rates, the distribution of winning prices, competition indicators such as the share of multi-bid auctions and mean bids per auction, and revenue-concentration metrics including the Gini coefficient, HHI, and top-k revenue shares, all defined in the chapter introduction. The two single-run cards show that the baseline and intervention configurations are broadly comparable in scale and structure. The most visible differences appear in completion, competition intensity, and the distribution of revenue across bidders.



Simulation B — Price-signal utilization

Headlines

```
Completion:                  B: 72.7% I: 45.5% Δ -27.3 pp
Mean cost/task:              B: 28.1250 I: 10.6000 Δ -17.5250
Revenue Gini:                B: 0.303 I: 0.436 Δ +0.132
Revenue HHI:                 B: 0.0757 I: 0.1452 Δ +0.0694
```

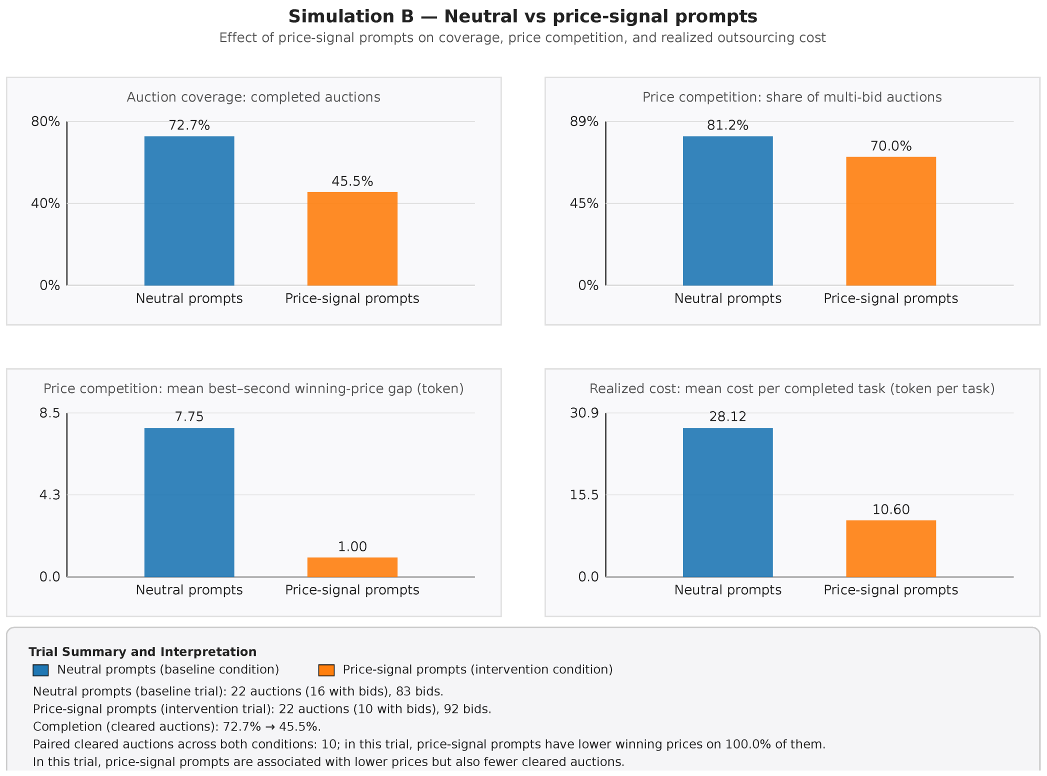
Market structure

```
Multi-bid auctions (share):  B: 81.2% I: 70.0% Δ -11.3 pp
Bidders ever winning:        B: 94.4% I: 57.9% Δ -36.5 pp
Top-3 revenue share:         B: 30.6% I: 57.0% Δ +26.5 pp
```

Heights — baseline: 308, intervention: 254

Figure 15 – Simulation B paired comparison. Baseline versus intervention levels and differences on completion, competition, pricing, and revenue concentration for the seed-matched trial pair.

The paired comparison in Figure 15 makes these run-level differences more explicit. In this seed-matched trial pair, enabling price-signal prompts is associated with a substantial reduction in the mean cost per completed task and a downward shift in the distribution of winning prices, but also with a reduction in the completion rate and weaker competition as measured by the share of multi-bid auctions and the mean number of bids per auction. The paired metrics for participation and revenue show how wins are redistributed across bidders when the prompt makes price signals salient: in this configuration, more of the total revenue is captured by the subset of agents that remain competitive under the new regime. These patterns are consistent with a story in which price-signal prompts make bidders more selective in when they participate and more aggressive when they do bid.



Trial Summary and Interpretation

🟦 Neutral prompts (baseline condition) 🟧 Price-signal prompts (intervention condition)

Neutral prompts (baseline trial): 22 auctions (16 with bids), 83 bids.
Price-signal prompts (intervention trial): 22 auctions (10 with bids), 92 bids.
Completion (cleared auctions): 72.7% → 45.5%.
Paired cleared auctions across both conditions: 10; in this trial, price-signal prompts have lower winning prices on 100.0% of them.
In this trial, price-signal prompts are associated with lower prices but also fewer cleared auctions.

Figure 16 – Simulation B headline results. Four-panel comparison of baseline (best-bid-neutral prompts) and intervention (best-bid-attentive prompts) on auction coverage (completion rate), share of multi-bid auctions, mean best–second winning-price gap in multi-bid auctions, and mean cost per completed task. The summary box at the bottom restates the number of auctions, the number of auctions with bids, and the share of paired auctions in which price-signal prompts lower the winning price.

Figure 16 complements these views with a compact, Simulation-B-specific summary constructed directly from the canonical `trial_card` exports. In this figure, auction coverage is again the completion rate, the share of launched auctions that clear with a winning bid. The share of multi-bid auctions measures how often auctions attract genuine competition (at least two bids) rather than behaving as take-it-or-leave-it offers; higher values mean competition is more common. The mean best–second winning-price gap is the average difference (in tokens) between the winning price and the second-best price in multi-bid auctions; a smaller gap indicates tighter price competition, while a larger gap suggests that the winner could have bid higher without being undercut. The mean cost per completed task is the average winning price across completed auctions, so lower values mean cheaper outsourcing for the buyer at the observed completion rate. The boxed summary at the bottom of the figure restates the number of auctions, the number of auctions with bids, and the completion rates in each condition, together with the share of paired auctions on which price-signal prompts lead to a lower winning price.

These headline metrics are chosen for Simulation B because they make the main trade-offs visible in simple terms: does making price signals salient help the buyer by lowering prices, does it change how often auctions behave as real competitions rather than singleton offers, and how does it affect the share of tasks that actually clear? For this particular seeded workload, the figure shows that price-signal prompts reduce completion but improve prices on the auctions that do clear, and they narrow the best–second gap, indicating tighter competition where bidding occurs.

From an engineering perspective, Simulation B illustrates that SWEChain-SDK can express bidder-side information policies at the prompt level without changing the underlying auction code or the on-chain state schema. The best-bid field already exists as on-chain state; the mechanism and datasets are unchanged; and only the way in which that information is surfaced in the prompt is altered. The resulting changes in completion, pricing, and revenue concentration can be traced back to a specific, versioned configuration that can be inspected and rerun in a controlled environment.

4.2.3 Threats to Validity and Limitations

Several caveats qualify the interpretation of Simulation B. First, the number of usable auction pairs after applying the outlier rule is modest, and the experiment reports a single seed-matched baseline–intervention trial run for this configuration. This was a deliberate scope, compute, and time choice: the dissertation focuses on building and validating the platform and the experimental protocol, and leaves larger-scale experimentation with many independent trials and parameter sweeps as future work. The figures should therefore be read as an internally consistent case study of what the platform can express rather than as definitive population-level estimates of the effect of price-signal prompts in all SWE-Agent markets.

Second, the treatment works purely through prompt text. In both arms, the best-bid field is technically available in the input; the difference is whether the prompt highlights it and instructs the agent to reason about it. This is appropriate for evaluating SWEChain-SDK as an experimentation platform for prompt-level policy changes, but it also means that effect sizes depend on how responsive the underlying language model is to prompt wording and on how stable that responsiveness remains over time and across model updates.

Third, agent policies are fixed for the duration of a trial and do not adapt across auctions. This isolates the effect of the prompt change in the short run but abstracts away learning dynamics and longer-run strategy shifts, such as tacit collusion, chronic underbidding, or strategic avoidance of certain principals, that might emerge in repeated play with price-signal information. Incorporating adaptive or meta-learning agents is an important direction for subsequent work.

Third, the convergence proxy τ_a depends on orchestration choices such as how frequently bids are submitted and evaluated within the fixed runtime budget. Different timing parameters could change absolute levels of τ_a even if the qualitative contrast between neutral and price-signal prompts remains similar. By keeping timing constant across arms and publishing the underlying event logs, the experiment permits alternative operational definitions of convergence and post hoc sensitivity analysis, but the metrics reported here correspond to one specific orchestration regime.

Finally, as in Simulation A, these experiments are run under a particular choice of model, hardware, and network configuration. While the reproducibility protocol in Appendix B is designed to support reruns on similar machines with the same model identifier, different inference stacks or hardware constraints could interact with timing and budget parameters in ways that shift completion and pricing levels, even when the qualitative comparison between neutral and price-signal prompts remains broadly similar.

4.3 Simulation Experiment C — Principal-Weighted Selection via Quality-Signal Utilization

The third experiment moves from bidder-side information to principal-side selection. Here we test whether replacing a price-only winner rule with a transparent price–quality scoring rule is associated with changes in cost-to-completion and first-pass success under the same tasks, agents, bids, and runtime parameters.

4.3.1 Experimental Design

The research question is whether a fixed, documented price–quality scoring rule, applied by the principal agent to structured bid summaries, can be implemented within SWEChain-SDK and whether, for the chosen workload, it is associated with lower total cost-to-completion and higher first-pass success, without clear signs of degraded price efficiency or participation.

In all conditions, bidders return a structured record containing: (i) a bid amount, (ii) a predicted probability that the submission will pass acceptance tests, and (iii) a predicted implementation duration. The auction format, reserve prices, payment rule (pay-as-bid), and eligibility thresholds match the earlier experiments.

In the baseline (price-only) condition, the principal is instructed, via its system prompt, to select the lowest admissible bid, with deterministic tie-breaking based on bidder identifiers. Predicted quality and duration are logged but ignored in the decision. In the intervention (price–quality scoring) condition, the principal instead applies a linear scoring rule of the form

$$\text{Score}_a(i) = w_q \cdot \widehat{\text{pass\_prob}}_{a,i} - w_c \cdot \text{bid\_amount}_{a,i} - w_\ell \cdot \widehat{\text{lateness\_hours}}_{a,i},$$

with weights (w_q, w_c, w_ℓ) fixed ex ante and lateness measured as predicted delay beyond the task deadline. The prompt instructs the principal to compute this score for each bid, to select the highest-scoring bidder, and to log both the chosen rule and the per-bid scores for audit. Higher scores mean that, according to the chosen weights, the bid offers better expected quality-for-price-and-timeliness; lower scores mean the opposite. This metric is used because Simulation C asks whether principals can move beyond pure price competition toward multi-attribute selection, and how such a move relates to cost-to-completion and success rates.

For each auction a, we define

$$C_a = c_a^{(0)} + r_a,$$

where $c_a^{(0)}$ is the initial payment and r_a aggregates penalties and rework costs; C_a is interpreted as cost-to-completion. A higher C_a means that, for that auction, the buyer had to pay more in total (initial payment plus any penalties or repeats) before the task was accepted; a lower C_a means cheaper completion. First-pass success (First-Pass Success (FPS)) is

$$\text{FPS} = \frac{1}{A}\sum_a F_a,$$

where $F_a = 1$ if the first submission passes acceptance tests and 0 otherwise. FPS close to one means that most tasks succeed on the first attempt; FPS close to zero means that most tasks require rework. These metrics are central in Simulation C because they capture exactly what a quality-sensitive principal might care about: how often the first try is good enough, and how much it ultimately costs to get tasks over the finish line.

Pricing is monitored via the same discount-from-reserve metric s_a as in Simulation B, and participation is summarized by the median number of distinct bidders per auction and the HHI of wins. As in the previous experiments, metrics are computed on auction pairs that pass the outlier rule, and randomness is controlled via seed-matched runs.

Principal finite-state machines and scoring policy

The selection rule in Simulation C is implemented at the principal agent level, through a finite-state machine that consumes structured bids, evaluates them, and chooses a winner. Bidders reuse the bidding controllers from the previous experiments; the main variation lies in how the principal processes their submissions.

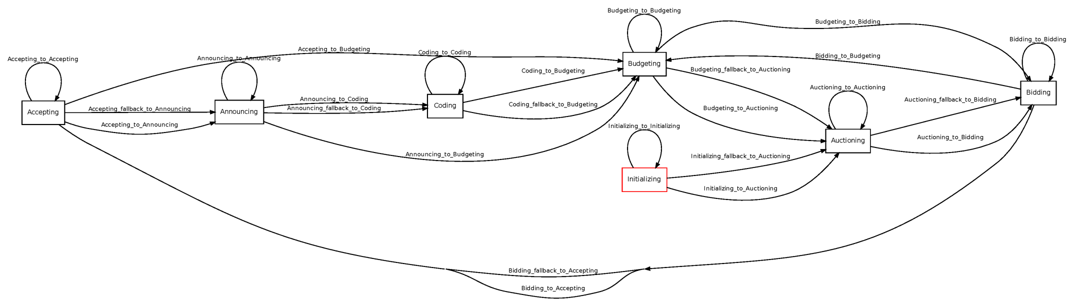


Figure 17 – Finite-state machine for the baseline principal in Simulation C. After collecting and validating bids, the principal applies a price-only selection rule: it chooses the lowest admissible bid and logs predicted quality and lateness without using them in the decision.

Figure 17 shows the baseline principal controller. The FSM includes states for collecting bids until the auction closes, parsing and validating them, and then selecting a winner. In the baseline configuration, the winner-selection state applies a simple rule: among all admissible bids, it chooses the one with the lowest price, breaking ties deterministically by bidder identifier. Predicted pass probabilities and lateness estimates are passed through and logged but do not influence the selection decision.

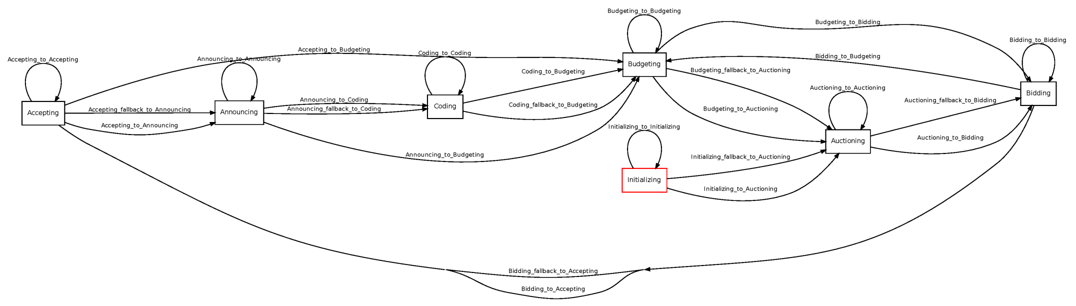
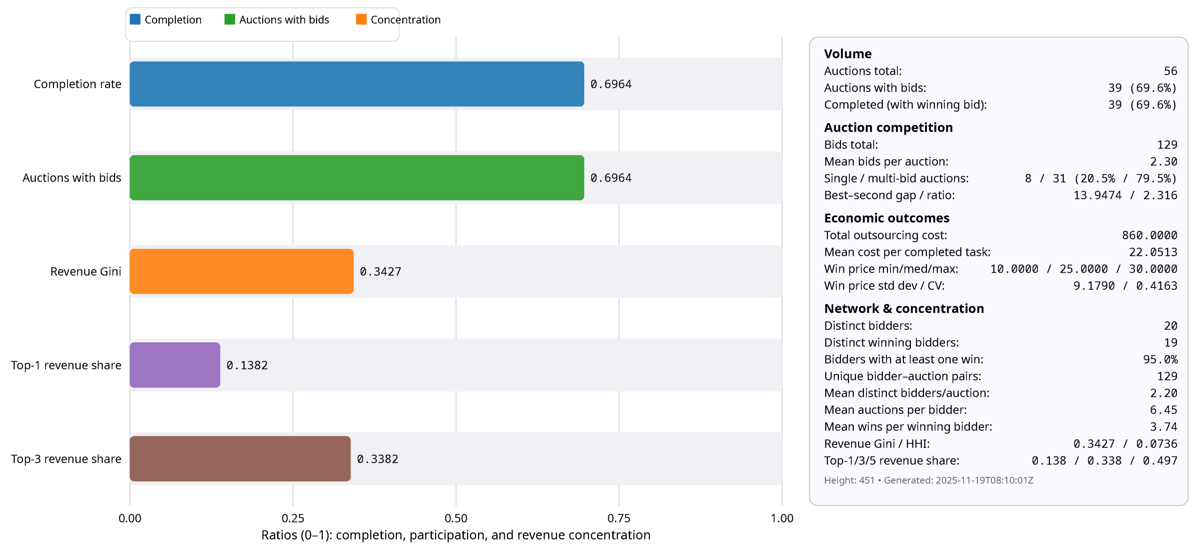


Figure 18 – Finite-state machine for the intervention principal in Simulation C. The control flow for collecting and validating bids is unchanged, but the winner-selection state is replaced by a price–quality scoring rule that combines predicted pass probability, price, and predicted lateness according to fixed weights.

Figure 18 shows the principal finite-state machine under the price–quality scoring intervention. The early stages of the controller (bid collection and validation) mirror the baseline, but the winner-selection state is replaced by a scoring state that computes $\text{Score}_a(i)$ for each bid, using the fixed weights (w_q, w_c, w_ℓ) and the structured fields in the bid. The principal then selects the highest-scoring bidder, logs the score components for each bid, and records the chosen rule. Prompt text in the scoring state reminds the principal of the trade-offs between predicted quality, price, and lateness and instructs it to apply the stated scoring rule rather than ad hoc heuristics.

Presenting these FSMs clarifies that Simulation C varies a specific part of the principal's control logic while leaving the underlying auction code and bidder behavior unchanged. The comparison between baseline and intervention runs thus speaks to how this scoring policy behaves under the chosen workload and configuration, rather than to a more general class of quality-sensitive rules.

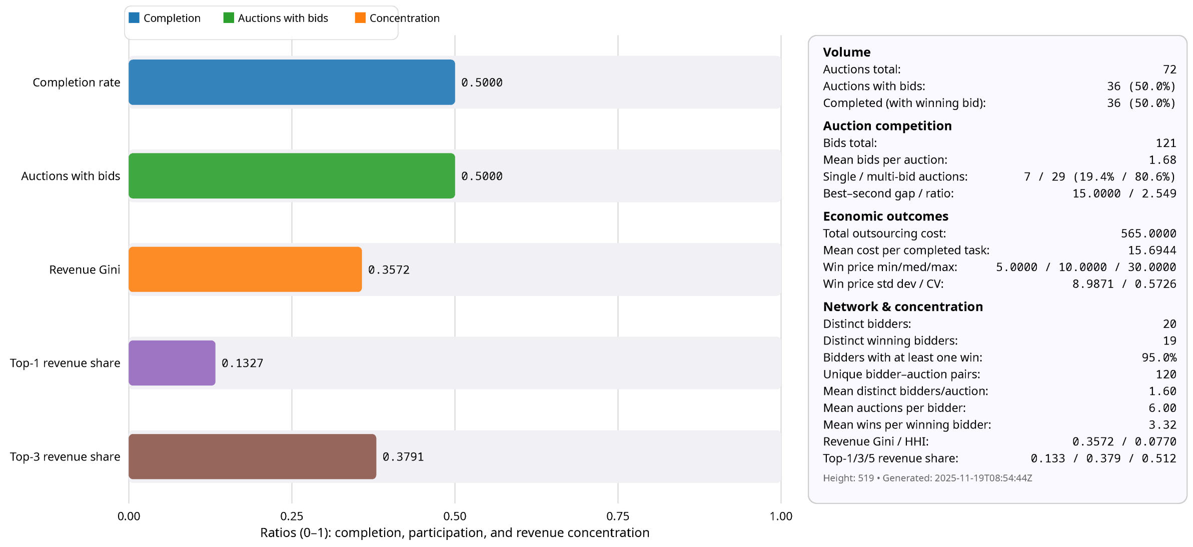
4.3.2 Simulation Results and Discussion



Volume	
Auctions total:	56
Auctions with bids:	39 (69.6%)
Completed (with winning bid):	39 (69.6%)
Auction competition	
Bids total:	129
Mean bids per auction:	2.30
Single / multi-bid auctions:	8 / 31 (20.5% / 79.5%)
Best–second gap / ratio:	13.9474 / 2.316
Economic outcomes	
Total outsourcing cost:	860.0000
Mean cost per completed task:	22.0513
Win price min/med/max:	10.0000 / 25.0000 / 30.0000
Win price std dev / CV:	9.1790 / 0.4163
Network & concentration	
Distinct bidders:	20
Distinct winning bidders:	19
Bidders with at least one win:	95.0%
Unique bidder–auction pairs:	129
Mean distinct bidders/auction:	2.20
Mean auctions per bidder:	6.45
Mean wins per winning bidder:	3.74
Revenue Gini / HHI:	0.3427 / 0.0736
Top-1/3/5 revenue share:	0.138 / 0.338 / 0.497

Height: 451 • Generated: 2025-11-19T08:10:01Z

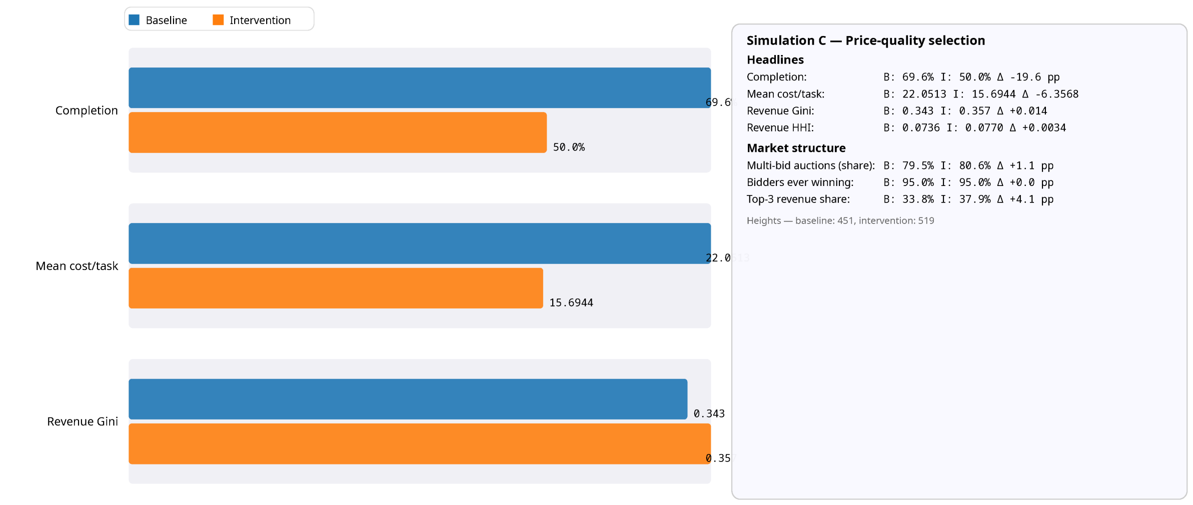
Figure 19 – Simulation C baseline configuration: canonical trial-card summary for price-only selection, including completion, outsourcing cost, participation, and revenue concentration.



```
Volume
Auctions total:                                   72
Auctions with bids:                       36 (50.0%)
Completed (with winning bid):             36 (50.0%)
Auction competition
Bids total:                                      121
Mean bids per auction:                          1.68
Single / multi-bid auctions:    7 / 29 (19.4% / 80.6%)
Best–second gap / ratio:              15.0000 / 2.549
Economic outcomes
Total outsourcing cost:                     565.0000
Mean cost per completed task:                15.6944
Win price min/med/max:      5.0000 / 10.0000 / 30.0000
Win price std dev / CV:                8.9871 / 0.5726
Network & concentration
Distinct bidders:                                 20
Distinct winning bidders:                         19
Bidders with at least one win:                 95.0%
Unique bidder–auction pairs:                     120
Mean distinct bidders/auction:                  1.60
Mean auctions per bidder:                       6.00
Mean wins per winning bidder:                   3.32
Revenue Gini / HHI:                    0.3572 / 0.0770
Top-1/3/5 revenue share:          0.133 / 0.379 / 0.512
Height: 519 • Generated: 2025-11-19T08:54:44Z
```

Figure 20 – Simulation C intervention configuration: canonical trial-card summary for principal-weighted selection under the same tasks, agents, and seeds as the baseline.

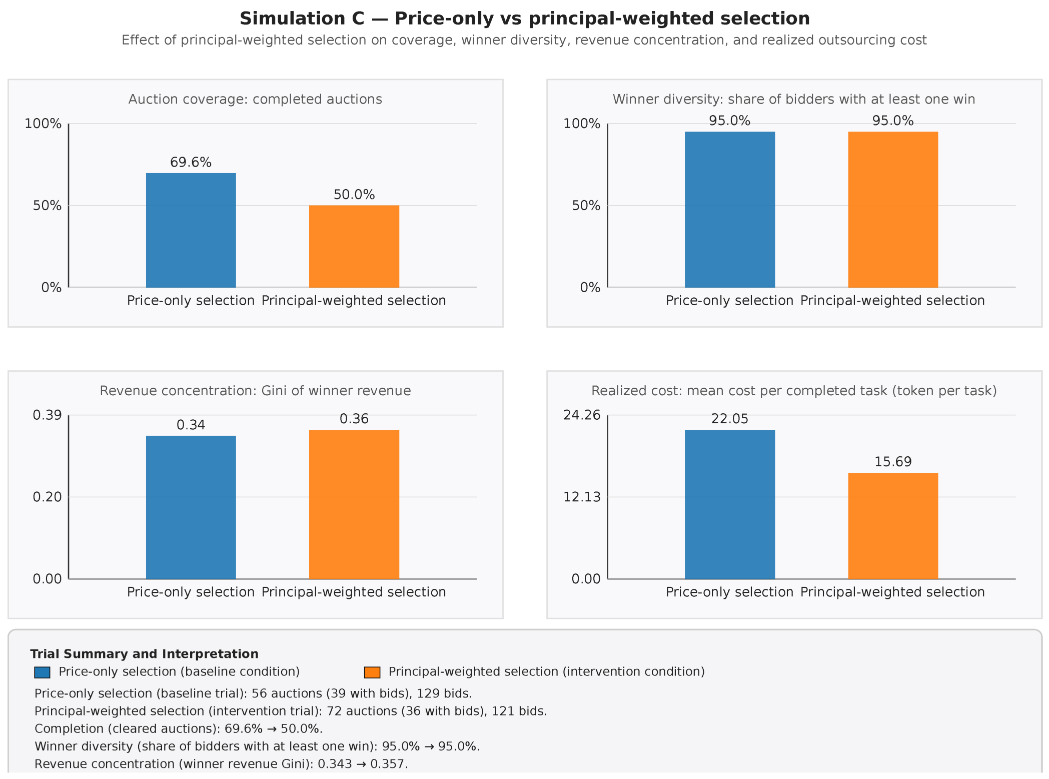
Figures 19 and 20 show the canonical trial-card summaries for the baseline and intervention runs, respectively. As in the earlier experiments, each card reports completion, outsourcing cost, competition, participation, and revenue-concentration metrics under the common definitions introduced at the start of the chapter.



```
Simulation C — Price-quality selection
Headlines
Completion:                  B: 69.6% I: 50.0% Δ -19.6 pp
Mean cost/task:              B: 22.0513 I: 15.6944 Δ -6.3568
Revenue Gini:                B: 0.343 I: 0.357 Δ +0.014
Revenue HHI:                 B: 0.0736 I: 0.0770 Δ +0.0034
Market structure
Multi-bid auctions (share):  B: 79.5% I: 80.6% Δ +1.1 pp
Bidders ever winning:        B: 95.0% I: 95.0% Δ +0.0 pp
Top-3 revenue share:         B: 33.8% I: 37.9% Δ +4.1 pp
Heights — baseline: 451, intervention: 519
```

Figure 21 – Simulation C paired comparison. Baseline versus intervention differences on auction-level economic and network metrics after applying the outlier rule.

The paired comparison in Figure 21 reports baseline–intervention differences at the auction level for the same core metrics: cost-to-completion C_a, FPS, discount from reserve, participation, and concentration. Each panel shows baseline levels, intervention levels, and paired differences for the auctions that pass the outlier rule, under the shared seed-matched protocol.



Trial Summary and Interpretation

🟦 Price-only selection (baseline condition) 🟧 Principal-weighted selection (intervention condition)

Price-only selection (baseline trial): 56 auctions (39 with bids), 129 bids.
Principal-weighted selection (intervention trial): 72 auctions (36 with bids), 121 bids.
Completion (cleared auctions): 69.6% → 50.0%.
Winner diversity (share of bidders with at least one win): 95.0% → 95.0%.
Revenue concentration (winner revenue Gini): 0.343 → 0.357.

Figure 22 – Simulation C headline results. Four-panel comparison of baseline (price-only winner rule) and intervention (principal-weighted price–quality scoring) on auction coverage (completion rate), winner diversity (fraction of bidders with at least one win), revenue concentration (revenue Gini), and mean cost per completed task.

Finally, Figure 22 condenses the main results into four headline metrics that parallel the Simulation A and B summaries. Auction coverage is again the completion rate, the share of auctions that clear with a winner; higher coverage means more tasks are completed. Winner diversity is the share of bidders with at least one win, so higher winner diversity means that more of the active bidders receive some revenue instead of a few agents dominating the market. Revenue concentration is measured by the Gini coefficient of winner revenue; values closer to zero indicate more equal revenue shares, while higher values indicate that payments are more concentrated in a small subset of agents. Realized cost is the mean cost per completed task in tokens, which summarizes how expensive it is, on average, to get a task to completion under each winner-selection rule. These metrics are appropriate for Simulation C because the policy lever directly targets the trade-off between price and expected quality: we want to see whether quality-sensitive selection changes how many tasks succeed, how much they cost, and how unevenly rewards are distributed across agents.

For the chosen configuration, the quality-adjusted rule appears to change both cost-to-completion and FPS relative to the price-only baseline, while keeping participation within a similar range. The detailed effect sizes are documented in the appendix; at a high level, the experiment illustrates how principal-side policies implemented as prompt-level evaluation rules over structured bids can be expressed and examined in SWEChain-SDK without modifying the underlying auction code.

From an engineering standpoint, Simulation C shows how marketplaces can move beyond pure price competition while retaining explicit, inspectable decision criteria. The scoring rule is fixed, documented, and encoded in prompts; per-bid scores are logged; and runs can be reconstructed from the published manifests and logs. In practice, designers could sweep over alternative weight vectors (w_q, w_c, w_ℓ), introduce non-linear penalties for severe lateness, or couple the scoring rule with specialization and price-signal treatments to explore richer design spaces. The key point for this dissertation is that such policies can be encoded, versioned, and evaluated within SWEChain-SDK under controlled, paired conditions and that they have visible consequences for task completion, outsourcing cost, and revenue concentration in the simulated markets considered here.

4.3.3 Threats to Validity and Limitations

Several considerations qualify the interpretation of Simulation C. First, FPS is sensitive to acceptance-test strictness and timeout parameters; while these are fixed across arms, alternative acceptance harnesses could shift absolute levels even if paired differences persist. Second, the structured fields used in scoring (predicted pass probability and lateness) are model outputs and may be miscalibrated; miscalibration can attenuate or distort the apparent benefits of quality-sensitive scoring. Third, the chosen linear scoring with fixed weights (w_q, w_c, w_ℓ) is only one of many plausible rules; different weights or non-linear penalties could change effect sizes and trade-offs. Fourth, as in the earlier experiments, the number of usable auction pairs after the outlier rule can be modest, so small-sample fluctuations remain an important concern. Finally, bidders do not adapt strategies across auctions in this setup; longer-run equilibria (for example, shading against quality-sensitive principals or shifting participation toward principals with particular scoring rules) are out of scope for this controlled comparison.

4.4 Discussion

Simulation A operationalizes comparative advantage via private specialization tags and cost multipliers on the bidder side. Simulation B studies bidder-side price-signal utilization by making the current best admissible bid salient in the bidding prompt. Simulation C moves to principal-side policy, introducing a price–quality scoring rule over structured bids in place of a pure price winner rule. Across these three simulation experiments we vary a single policy dimension at a time under a tightly specified environment and examine how the observed outcomes differ between baseline and intervention runs. Each experiment is based on a single seed-matched baseline–intervention pair, and all results are conditional on the specific workloads, parameter settings, and models used. Because we report a single seed-matched baseline–intervention pair per experiment, the results are illustrative for the specific seeded workloads and should not be interpreted as averaging over many independent random seeds.

Within these limits, the experiments provide concrete case studies of how simple, programmable policy changes relate to task completion, outsourcing cost, and revenue concentration in decentralized SWE-Agent outsourcing markets. The chapter keeps the focus on the three outcome families highlighted in the abstract and introduction. For task completion, we monitor both assignment rates (completion as a share of launched auctions) and, where relevant, first-pass success. For outsourcing cost, we track mean cost per completed task and related price statistics, sometimes normalized by rough reserve levels. For revenue concentration, we use the revenue Gini, HHI, and top-k revenue shares derived from the trial cards. Presenting these quantities through the same trial-card dashboards, paired-comparison cards, and experiment-specific summary figures allows us to compare baseline and intervention regimes within each simulation and, at a high level, to align the three experiments under a common empirical vocabulary.

Taken together, the experiments support the dissertation's claim that a blockchain-native SDK such as SWEChain-SDK can host controlled, paired baseline–intervention studies of decentralized SWE-Agent outsourcing markets with event-level provenance. In this setting, market-design ideas familiar from economics and operations—comparative advantage, price-signal utilization, and multi-attribute scoring—are not just informal guidelines; they are encoded as concrete configurations of code and prompts that can be versioned, executed, and examined. For each experiment, we hold tasks, seeds, and infrastructure fixed; toggle a single policy dimension; and compare the resulting trial-card summaries and derived plots. The observed differences are not guaranteed to generalize beyond the specific workloads and models considered, but they illustrate how SWEChain-SDK can be used to study SWE-Agent markets in a way that is explicit about mechanisms, transparent about data, and grounded in event-level logs.

Finally, the chapter highlights a practical pattern for future work in SWE-Agent Economics. Mechanisms and agent policies can be encoded as code and prompts; tasks, seeds, and infrastructure can be fixed in manifests; single policy dimensions can be varied while others are held constant; and events can be logged at sufficient granularity to support re-analysis. SWEChain-SDK provides the concrete tooling for this pattern; the experiments in this chapter demonstrate how it can be applied in a small set of illustrative scenarios. Subsequent work can build on this pattern by enlarging the space of workloads and policies, by increasing the number of independent runs, and by introducing adaptive agents and richer equilibrium behaviors, while retaining the same emphasis on controlled comparisons, transparent mechanisms, and event-level provenance.

Chapter 5

Related Work

AI agents' strategic interactions in economic settings—a key line of inquiry in AI-agent economics—have attracted increasing attention amid recent advances in generative AI. We focus on blockchain-based SWE-Agent outsourcing markets, which link intelligent software engineering and software-engineering economics through programmable, transparent, and auditable mechanisms that centralized platforms rarely expose. We evaluate whether a blockchain-native SDK for economic network simulations can support controlled, paired baseline–intervention experiments under fixed conditions in decentralized SWE-Agent outsourcing markets. Against this backdrop, this chapter situates the dissertation within three strands of related work. First, it reviews work that models agents as decision-makers embedded in markets and networks. Second, it surveys recent efforts that explicitly frame *AI-agent economics* and construct virtual agent markets or exchanges. Third, it discusses experimental platforms and SDKs for AI agents and positions *SWEChain-SDK* with respect to these systems by highlighting the remaining research gap.

5.1 Agents, Markets, and Computational Economics

Classical work on autonomous agents and multi-agent systems provides the conceptual basis for viewing software entities as decision-makers with state, goals, and policies. Cognitive and symbolic architectures such as Soar (LAIRD; NEWELL; ROSENBLOOM, 1987), BDI agents (RAO; GEORGEFF, 1995; WOOLDRIDGE, 2000), and planning formalisms like STRIPS and PDDL (FIKES; NILSSON, 1971; MCDERMOTT, 1998) model agents as goal-directed problem solvers under constraints, while behavior-based approaches emphasize reactive control and layered architectures (BROOKS, 1986; BROOKS, 1991; GAT, 1998; BONASSO et al., 1997). Multi-agent systems work then formalizes coordination, communication, and organizational structures (WOOLDRIDGE, 2009; JENNINGS; SYCARA; WOOLDRIDGE, 1998; JENNINGS, 2000), and standard reinforcement-learning and decision-theoretic formulations treat agents as policy-bearing processes in stochastic environments with partial observability (KAELBLING; LITTMAN; CASSANDRA, 1998;

SUTTON; BARTO, 2018). These ideas align with viewing SWE-Agents as bounded, tool-using processes that must make economic decisions under interface, budget, and information constraints.

Agent-based modeling and simulation extend these concepts to large populations of interacting agents. Classic models such as Schelling's segregation dynamics (SCHELLING, 1971b), El Farol (ARTHUR, 1994), and Axelrod's work on cooperation (AXELROD, 1984) show how simple local rules can generate rich macro-level patterns. General-purpose ABM frameworks and surveys (WILENSKY, 1999; NIAZI; HUSSAIN, 2011) codify best practices for building and analyzing agent-based models, including seed control, sensitivity analysis, and careful reporting of outcomes. Methodological work in simulation (LAW, 2015; EPSTEIN; AXTELL, 1996; RAILSBACK; GRIMM, 2019; SARGENT, 2013; OBERKAMPF; ROY, 2010) emphasizes verification, validation, and experimental design so that simulation results are robust and interpretable. These practices motivate the design of SDKs that support reproducible runs under fixed conditions, with explicit control over randomness, timing, and configuration.

Agent-based computational economics (ACE) applies ABM techniques to markets and macroeconomic environments (TESFATSION, 2002; TESFATSION, 2006; TESFATSION, 2022). Agents are firms, households, or intermediaries that follow behavioral rules instead of solving analytically tractable optimization problems. Work by Axtell, Farmer, Tesfatsion, and others argues that agent-based models are well suited to studying out-of-equilibrium dynamics, institutional detail, and distributional outcomes (AXTELL, 2000; FARMER; GEANAKOPLOS, 2009; ARTHUR et al., 1997; TESFATSION, 2002). Classical economic foundations on incentives, information, and institutions—from Robbins and Samuelson's definitions of economic behavior (ROBBINS, 1932; SAMUELSON; NORDHAUS, 2010) to Coase, Hurwicz, and Laffont–Martimort on firms, mechanisms, and principal–agent relationships (COASE, 1937; HURWICZ, 1972; LAFFONT; MARTIMORT, 2002)—provide the language used in this dissertation to discuss task allocation, information disclosure, and incentive compatibility.

Networked views of markets further enrich this picture. Work on social and economic networks shows how topology shapes diffusion, bargaining power, and inequality (JACKSON, 2008; JACKSON, 2010; EASLEY; KLEINBERG, 2010; CALVÓ-ARMENGOL; JACKSON, 2004; GABAIX, 2009; BARABÁSI; ALBERT, 1999; ACEMOGLU et al., 2012). Inequality and concentration measures such as the Gini coefficient, Atkinson index, and Herfindahl–Hirschman index (HHI) provide compact diagnostics of how rewards are distributed across agents (GINI, 1912; ATKINSON, 1970; COWELL, 2011). For software-engineering markets, this suggests viewing SWE-Agents and task issuers as a bipartite economic network whose structure and flows (tasks, tokens, artifacts) are shaped by mechanism rules and agent policies, and whose efficiency and equity can be measured with standard tools from econometrics and network science.

5.2 AI-Agent Economics and Virtual Agent Markets

Recent work proposes *AI-agent economics* as a distinct research agenda that studies autonomous AI agents as economic decision-makers. Yang and Zhai articulate ten principles for AI-agent economics (YANG; ZHAI, 2025), emphasizing that agents should be modeled with explicit objectives, constraints, and information sets; that mechanisms and environments should be formally specified; and that evaluation should consider welfare, stability, robustness, and alignment with human objectives. Their framing calls for experimental substrates that expose economic interfaces (allocations, prices, budgets, constraints) rather than only task-level accuracy, and that enable systematic study of how mechanism design and agent policies interact.

Building on this conceptual foundation, Tomasev et al. introduce *Virtual Agent Economies* (TOMASEV et al., 2025), which provide environments where learning agents interact through markets for virtual goods and services. These economies are designed to support policy evaluation, distributional analysis, and stress-testing of multi-agent systems. Yang et al.'s *Agent Exchange* (YANG et al., 2025) similarly proposes an exchange architecture in which AI agents trade services under defined market rules to study pricing, liquidity, and strategic behavior. In both cases, agents are embedded in explicit markets, but the domains tend to focus on generic goods, media, or synthetic services rather than software-engineering tasks.

Complementary studies place language models directly into auction and bargaining environments. Chen et al. evaluate strategic planning and execution in an auction arena where language-model agents bid for items under different rules (CHEN et al., 2024). Shah et al. examine language models as auction participants, comparing their strategic behaviors and outcomes across mechanisms (SHAH et al., 2025). These settings treat auctions as testbeds for reasoning and responsiveness to incentives, and they highlight both the promise and fragility of using LLM agents in economic contexts.

A broader ecosystem of agent benchmarks evaluates AI agents on web, desktop, and app interaction rather than explicit markets. WebShop, WebArena, VisualWebArena, and BrowserGym assess agents on shopping or navigation tasks (YAO et al., 2022; ZHOU et al., 2024; KOH et al., 2024; BrowserGym Team, 2024), while OSWorld, AndroidWorld, AppWorld, and related platforms focus on desktop or mobile interaction and multi-step tool use (WANG; LI et al., 2024; CHANG et al., 2025; TRIVEDI et al., 2024). Other environments such as MineDojo, BEHAVIOR-1K, and OmniGibson provide rich physical or virtual worlds for open-ended tasks (FAN et al., 2022; LI et al., 2024; OmniGibson Contributors, 2024). These platforms typically measure success rates, completion times, or reward in task-centric benchmarks. They rarely expose explicit market structures, prices, or welfare metrics, and they do not aim to model software-engineering marketplaces.

Within software engineering, systems such as SWE-Agent (YANG et al., 2024; SWE-agent Contributors, 2025; SWE-AGENT..., 2025) and OpenHands (WANG et al., 2025;

OpenHands Contributors, 2025) focus on automated software-development workflows, repository-level evaluation, and human-in-the-loop integration. They are often evaluated against benchmarks like SWE-bench (JIMENEZ et al., 2024; YANG et al., 2023), which measure an agent's ability to resolve real-world issues and pass associated tests. These systems bring agents closer to realistic SWE workflows, but their evaluation pipelines emphasize correctness and throughput rather than explicit economic interactions, prices, or surplus. Economic aspects, when present, are implicit (e.g., time-to-completion or resource usage), not encoded as market mechanisms.

Taken together, this body of work establishes AI-agent economics as a conceptual umbrella and provides a variety of agent environments—from virtual economies and exchanges to SWE-focused tools and generic agent benchmarks. However, these environments either focus on generic goods or financial assets, or treat economic interactions as evaluation tools rather than primary objects of study in software-engineering task markets.

5.3 Research Gap and Positioning

Existing experimental platforms and SDKs for AI agents provide important building blocks, but they leave a gap for domain-specific, economics-aware infrastructures tailored to software-engineering markets.

Multi-agent orchestration frameworks such as AutoGen, CAMEL, MetaGPT, Voyager, and related systems (WU; YANG; AL., 2023; LI et al., 2023; HONG et al., 2023; WANG et al., 2023; DU et al., 2023; SHEN et al., 2023) offer abstractions for role decomposition, tool invocation, and conversation-based coordination. They demonstrate that orchestration choices (roles, tool contracts, memory scopes, retry policies) can significantly affect task outcomes. However, they typically run on general-purpose compute platforms, log events in ad hoc formats, and do not treat economic observables (bids, allocations, payments, penalties) or mechanism parameters as first-class, versioned entities. Reproducing experiments that compare mechanism variants or policy interventions across time is therefore difficult.

Blockchain infrastructures offer an alternative substrate with stronger guarantees about ordering, finality, and auditability. Systems such as Bitcoin and Ethereum (NAKAMOTO, 2008; BUTERIN, 2014; WOOD, 2014; GARAY; KIAYIAS; LEONARDOS, 2015; EYAL; SIRER, 2014; KIAYIAS et al., 2017; BUTERIN; GRIFFITH, 2017) establish a transaction log with consensus-based ordering. Appchain frameworks like Tendermint/CometBFT and the Cosmos SDK (KWON, 2014; KWON; BUCHMAN, 2016; CONTRIBUTORS, 2023; FOUNDATION, 2020) allow designers to fix consensus rules, block cadence, and application logic, and to encode domain-specific modules (e.g., auctions, governance) as explicit state machines. Research on MEV and transaction-fee mechanisms (DAIAN et al., 2020b; ROUGHGARDEN, 2021; ANGERIS et al., 2020) shows that general-purpose mempools and fee markets can introduce non-stationary noise and strategic behavior, which is undesirable for tightly controlled experiments. Agent-like actors in DeFi (liquidators, arbitrageurs, searchers) demonstrate the feasibility of on-chain economic agents but focus on financial assets rather than software-engineering tasks, and they typically operate in environments where mempool dynamics are part of the object of study rather than a controlled background.

Reproducible computational research and provenance work argues that scientific software systems should expose manifests, dataflow, and configuration metadata to enable independent recomputation (GIL et al., 2007; WILSON et al., 2017). Simulation methodology in software engineering and related fields (MONTGOMERY, 2017; WOHLIN et al., 2012; LAW, 2015; SARGENT, 2013; OBERKAMPF; ROY, 2010) stresses clear estimands, controlled confounders, and transparent reporting of design choices. Recent efforts in energy and resource accounting for machine-learning experiments (STRUBELL; GANESH; MCCALLUM, 2019; MASANET et al., 2020; HENDERSON et al., 2020) further argue for tracking computational cost as part of experimental results. Yet most AI-agent platforms and orchestration frameworks do not integrate these concerns into their core abstractions; manifests, seeds, and configuration hashes are usually handled by external scripts or documentation rather than enforced by the execution environment.

In software engineering, there is also a gap between AI-assisted developer tools and economic analyses of platforms and labor markets. Developer-focused systems such as GitHub Copilot and related tools (GitHub, 2023; Microsoft, 2023; PENG et al., 2023; KALLIAMVAKOU, 2022; YETIŞTIREN et al., 2023; PEARCE et al., 2021; PERRY et al., 2023; ASARE; NAGAPPAN; ASOKAN, 2024; TIAN et al., 2023) emphasize productivity and correctness but do not model or expose the economic mechanisms by which work is allocated, priced, and settled. Platform-economics and labor studies, in turn, analyze centralized systems (e.g., Upwork-style markets) where mechanisms and data are only partially observable (BERG et al., 2018; GRAY; SURI, 2019; EINAV; FARRONATO; LEVIN, 2016; TADELIS, 2016; AGRAWAL et al., 2015). This limits the possibility of controlled experimental manipulation of mechanism rules in realistic software-task settings.

Against this background, AI-agent economics provides conceptual foundations and high-level platforms for studying AI agents as economic actors (YANG; ZHAI, 2025; TOMASEV et al., 2025; YANG et al., 2025; CHEN et al., 2024; SHAH et al., 2025), but there is still a missing piece:

- ❏ Existing virtual agent economies and exchanges focus on generic goods or services, not software-engineering tasks bound to repositories, tests, and artifacts.

- ❏ Existing SWE-Agent frameworks focus on automation and evaluation of software workflows, not on explicit market mechanisms, prices, or welfare metrics.

- ❏ Existing blockchain systems demonstrate agent-like behavior in financial domains, but rely on public mempools and fee markets that complicate controlled, seed-matched experimentation.

This dissertation addresses that gap by evaluating whether a blockchain-native SDK for economic network simulations can support controlled, paired baseline–intervention experiments under fixed conditions in decentralized SWE-Agent outsourcing markets. We present *SWEChain-SDK*, a configurable, extensible blockchain-native SDK that provides a local sandbox chain for SWE-Agent outsourcing auctions and event-level logging of bids, allocations, payments, and artifacts, enabling analysis and scripted re-runs of paired trials under comparable conditions. We conduct paired simulation experiments that vary one policy dimension at a time—(A) comparative advantage via agent specialization, (B) competitive bidding via bidders' price-signal utilization, or (C) principal-weighted selection via quality-signal utilization—while holding rules, datasets, random seeds, time base, and the agent pool fixed, and report the observed paired differences in routing, pricing, participation, and concentration. In doing so, the dissertation extends emerging AI-agent economics work into blockchain-based SWE-Agent outsourcing markets and contributes a reusable, evaluation-ready SDK and artifact set for reproducible studies of decentralized SWE-Agent economic networks.

Chapter 6
Conclusion and Future Work

AI agents' strategic interactions in economic settings—a key line of inquiry in AI-agent economics—have attracted increasing attention amid recent advances in generative AI. This dissertation focuses on blockchain-based SWE-Agent outsourcing markets, which link intelligent software engineering and software-engineering economics through programmable, transparent, and auditable mechanisms that centralized platforms rarely expose. Within this setting, the central question is whether a blockchain-native SDK for economic network simulations can support controlled, paired baseline–intervention experiments under fixed conditions in decentralized SWE-Agent outsourcing markets.

The work is organized around *SWEChain-SDK*, a blockchain-native software development kit that provides a local sandbox chain for SWE-Agent outsourcing auctions and event-complete logging of bids, allocations, payments, and artifacts. On top of this chain, the dissertation designs and runs paired simulation experiments that vary exactly one policy dimension at a time while holding rules, datasets, random seeds, time base, and the agent pool constant. This chapter revisits the main contributions, summarizes the empirical findings together with their limitations, and outlines directions for future work grounded in the configurability and extensibility of *SWEChain-SDK*.

6.1 Summary and Contributions

The first contribution is a conceptual and terminological framing of *SWE-Agent Economics*. SWE-Agent Economics, as defined in Chapter 2, studies Software-Engineering Agents (SWE-Agents) as economic decision-makers in auction-based task markets implemented on blockchain-style infrastructures. In this framing, SWE-Agents are software-engineering agents that participate in outsourcing markets whose admission, matching, pricing, information, and settlement rules are explicit and programmable. By treating these markets as economic networks, the dissertation links recent work on AI-agent economics to software engineering and software-engineering economics, and positions SWE-Agent markets as a concrete setting in which strategic interactions between autonomous software systems can be studied empirically.

The second contribution is the design and implementation of *SWEChain-SDK* as a blockchain-native experimental platform. The SDK is implemented as an application-specific chain that encodes reverse procurement auctions with pay-as-bid pricing, fixed step cadence, deterministic tie-breaking, and explicit, versioned parameters for reserves, eligibility thresholds, penalties, and information visibility. This design provides a controlled environment in which SWE-Agents can bid for tasks, receive allocations, and settle payments under transparent rules. Event-complete logging of bids, allocations, payments, and artifacts, together with configuration manifests and shared seeds, supports scripted, seed-matched reruns and re-analysis under comparable conditions and enables external inspection, visualization, and verification.

The third contribution is an evaluation-ready artifact package and a set of illustrative simulation experiments built on top of this platform. The primary artifact bundle is released as an open-source repository at <https://github.com/lascam-UFU/swechain-sdk>, which hosts the SDK code, configuration files, seeds, logs, and scripts for figure generation and analysis. These materials enable scripted reruns and extensions of the experiments under comparable conditions. Using this bundle, the dissertation conducts three paired experiments in decentralized SWE-Agent outsourcing markets. Experiment A varies comparative advantage via agent specialization, Experiment B varies bidders' price-signal utilization, and Experiment C varies principal-weighted selection via quality-signal utilization. Across all three, the methodology emphasizes matched baseline–intervention comparisons under fixed conditions rather than one-off performance claims or mechanism optimization.

Taken together, these contributions answer the dissertation's research questions. The primary question asked whether a blockchain-native SDK for economic network simulations can support controlled, paired baseline–intervention experiments under fixed conditions in decentralized SWE-Agent outsourcing markets. The architecture in Chapter 3, the simulation workflow in Chapter 4, and the accompanying appendices show that SWEChain-SDK can host seed-matched baseline and intervention runs with fixed tasks, agent pools, timing, and auction parameters, and can record event-complete logs and manifests that support scripted reruns and re-analysis; within the scope of the workloads and infrastructure considered here, this provides an affirmative answer to that question.

A second line of inquiry asked how bidder-side policy levers relate to task completion, outsourcing cost, and revenue concentration. Simulation A shows that introducing private specialization tags and mild in-niche cost discounts is associated with slightly higher coverage and much higher winner diversity, while leaving mean cost per completed task similar and concentrating realized revenue in the relatively small set of agents that successfully operate in their niches. Simulation B shows that making the standing best admissible bid salient in bidder prompts reduces completion and participation on some tasks but improves prices on the auctions that do clear and narrows the best–second price gap, consistent with tighter competition where bidding occurs. A third question asked how principal-side policies that combine price and predicted quality behave relative to a price-only winner rule. Simulation C demonstrates that replacing a pure price rule with a fixed, documented price–quality scoring rule can be implemented within SWEChain-SDK and, for the chosen configuration, is associated with changes in cost-to-completion and first-pass success and with shifts in revenue concentration, while keeping participation within a similar range. Within the limits of the single seed-matched trial pair per experiment, these results collectively illustrate how SWEChain-SDK can be used to encode, execute, and examine bidder-side and principal-side policies and to study their effects on completion, cost, and concentration in decentralized SWE-Agent outsourcing markets.

6.2 Findings and Limitations

The simulation experiments evaluate whether a blockchain-native Software Development Kit (SDK) for economic network simulations can support controlled, paired baseline–intervention experiments under fixed conditions in decentralized SWE-Agent outsourcing markets, as stated in the abstract. Across all three experiments, *SWEChain-SDK* provides a local sandbox chain, seed-matched runs, and a common trial-card and paired-comparison pipeline that report differences in task completion, outsourcing cost, and revenue concentration.

Experiment A shows that *SWEChain-SDK* can encode a simple agent-side specialization policy and reveal its effects through the standard metrics. Specialization tags, prompt nudges, and small in-niche and out-of-niche cost multipliers are introduced by configuration and prompts only, while the auction code and workload are held fixed. For the seeded workload studied here, the paired trial cards and summary figures show that this change is associated with different assignment rates, winner–task alignment, and winner diversity.

Experiment B shows that bidder-facing price-signal policies can be expressed and examined without modifying the on-chain mechanism. The only difference between baseline and intervention is how the current best admissible bid is surfaced and emphasized in bidding prompts. Under fixed tasks, agents, seeds, and auction parameters, *SWEChain-SDK* records paired differences in completion, competition, pricing, and revenue concentration that are at least consistent with this prompt-level treatment of price signals mattering.

Experiment C shows that principal-side winner-selection policies can also be implemented and compared under fixed conditions. A price-only rule and a fixed price–quality scoring rule are both realized as principal controllers that consume the same structured bids. Using the same trial-card and comparison tooling, the SDK reports how cost-to-completion, first-pass success, participation, and revenue concentration differ between these two configurations for the chosen workload.

Across all experiments, baseline and intervention runs share the same chain configuration, tasks, agents, and seeds; only the policy under study changes. The released logs and manifest files are intended to let others recompute the reported metrics and run additional comparisons under comparable conditions.

These findings come with clear limitations. The experiments are run at modest scale with synthetic workloads and a simplified task taxonomy; the SWE-Agents follow finite-state controllers with fixed prompts and simple bidding strategies; and mechanism coverage is restricted to pay-as-bid reverse auctions with simple reserves and penalties. The evaluation emphasizes efficiency, participation, and revenue concentration, and does not model human developers, clients, or governance processes. Within these constraints, the experiments should be read as an evaluation of what *SWEChain-SDK* can express and measure under controlled, paired conditions, rather than as direct estimates of behavior in deployed SWE-Agent marketplaces.

6.3 Future Work

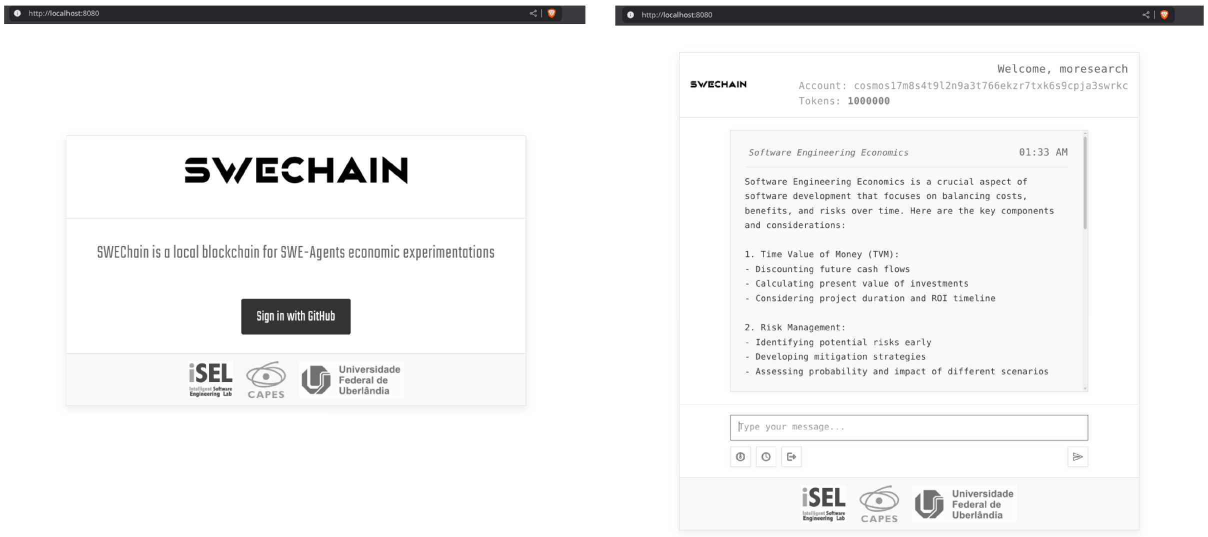
The configurability and extensibility of *SWEChain-SDK* open several directions for future work that deepen both the agent side and the mechanism side of SWE-Agent Economics.

One direction is to move from fixed policies to learning and adaptive SWE-Agents. Because the chain exposes clear economic signals—such as allocations, payments, and penalties—it is natural to integrate reinforcement learning or bandit-style adaptation, allowing agents to tune their bidding, participation, and task-selection strategies over time. This would make it possible to study questions about convergence, stability, and robustness when many learning agents interact in the same market, and to examine how different mechanisms shape or constrain emergent strategies.

A second direction is to broaden the family of mechanisms implemented in the SDK while retaining the matched-trial methodology. Uniform-price auctions, VCG-like mechanisms, combinatorial or package auctions, and richer reputation and history-dependent eligibility policies can all be encoded as variations in modules and configuration. With these in place, SWEChain-SDK could support systematic comparisons of efficiency, revenue, and distributional properties across a wider design space, as well as robustness studies under collusion, sybil attacks, or adversarial bidding behavior.

A third direction is to extend the system-level architecture beyond a single chain. By connecting SWEChain-SDK to other application-specific chains through inter-blockchain communication, future work could explore cross-chain outsourcing, settlement, and security, and could vary latency, finality, and fee regimes across chains. Embedding the SDK into larger experimental pipelines—with schedulers, logging services, and orchestrators for large batches of trials—would push the platform toward a full-scale laboratory for AI-agent economics in software engineering.

A fourth direction is to add explicit human-in-the-loop components and real repositories. A dedicated human-facing layer on top of SWEChain-SDK is being prototyped in the `swechain-human` repository at <https://github.com/lascam-UFU/swechain-human>, with the goal of allowing users to submit tasks, review artifacts, approve or reject outcomes, and override or re-allocate work while decisions remain logged on-chain. Another natural step is to bind auctions to actual issues, branches, and continuous-integration pipelines, so that some tasks correspond to real changes in real codebases. This would support evaluation of mixed human–agent workflows and oversight policies that combine automated markets with human judgment.


http://localhost:8080
SWECHAIN
Sign in with GitHub
iSEL Intelligent Software Engineering Lab
CAPES
Universidade Federal de Uberlândia
http://localhost:8080
SWECHAIN
Type your message...
iSEL Intelligent Software Engineering Lab
CAPES
Universidade Federal de Uberlândia

SWEChain is a local blockchain for SWE-Agents economic experimentations

```
Welcome, moresearch
Account: cosmos17m8s4t9l2n9a3t766ekzr7txk6s9cpja3swrkc
Tokens: 1000000

Software Engineering Economics                    01:33 AM

Software Engineering Economics is a crucial aspect of
software development that focuses on balancing costs,
benefits, and risks over time. Here are the key components
and considerations:

1. Time Value of Money (TVM):
- Discounting future cash flows
- Calculating present value of investments
- Considering project duration and ROI timeline

2. Risk Management:
- Identifying potential risks early
- Developing mitigation strategies
- Assessing probability and impact of different scenarios
```

Figure 23 – Illustrative view of human participation alongside SWE-Agents in future SWEChain-SDK deployments.

A fifth direction concerns evaluation methodology and governance. Future work can enrich welfare and fairness metrics to better reflect the objectives of clients, developers, and platform operators; examine long-run dynamics such as entry, exit, specialization, and concentration under different policy regimes; and prototype governance processes—on-chain or off-chain—for updating parameters and modules in SWEChain-SDK. These steps would connect the technical design of SWE-Agent marketplaces to institutional questions about accountability, alignment, and control.

A sixth, more open-ended direction is to turn SWEChain-SDK into the core of an ongoing tournament-style testchain, in the spirit of Axelrod's iterated-prisoner's-dilemma tournaments. In such a setting, researchers could bring their own agents and task distributions, deploy them under shared rules and schedules, and observe how strategies and mechanisms perform over long horizons. A persistent testchain of this kind would act as a living benchmark suite for SWE-Agent Economics, enabling comparative studies of agent designs, prompting strategies, and mechanisms across many contributors, and providing a common reference point for reproducible experiments that extend beyond a single dissertation. All of these extensions deepen the same SWE-Agent Economics agenda defined in Chapter 2, by exploring how different mechanisms and agent behaviors shape task completion, outsourcing cost, and revenue concentration in decentralized SWE-Agent markets.

Taken together, the dissertation treats decentralized SWE-Agent outsourcing markets as a concrete setting for AI-agent economics in software engineering and provides evidence that a blockchain-native SDK can support controlled, paired baseline–intervention experiments under fixed conditions. It presents *SWEChain-SDK*, a configurable, extensible blockchain-native SDK that provides a local sandbox chain for SWE-Agent software outsourcing auctions and event-complete logging of bids, allocations, payments, and artifacts, together with configuration metadata sufficient for re-runnable paired runs under comparable conditions, and makes these artifacts publicly available at <https://github.com/lascam-UFU/swechain-sdk>. It conducts paired simulation experiments that vary one policy dimension at a time—comparative advantage via agent specialization, competitive bidding via bidders' price-signal utilization, and principal-weighted selection via quality-signal utilization—while holding rules, datasets, random seeds, time base, and the agent pool fixed, and reports the resulting paired differences in efficiency (task completion and outsourcing cost), network structure, and inequality (revenue concentration) in SWE-Agent outsourcing markets. The hope is that this combination of conceptual framing, experimental platform, and initial evidence can serve as a foundation for broader work at the intersection of intelligent software engineering, software-engineering economics, and AI-agent economics.

References

ABDEL-HAMID, T. S.; MADNICK, S. E. The dynamics of software project staffing: A system dynamics based simulation approach. **IEEE Transactions on Software Engineering**, v. 17, n. 2, p. 109–135, 1991.

ACEMOGLU, D. et al. The network origins of aggregate fluctuations. **Econometrica**, v. 80, n. 5, p. 1977–2016, 2012.

AGRAWAL, A. K. et al. Digitization and the contract labor market: A research agenda. **NBER Working Paper 19525**, 2015.

AKERLOF, G. A. The market for “lemons”: Quality uncertainty and the market mechanism. **The Quarterly Journal of Economics**, v. 84, n. 3, p. 488–500, 1970.

ANGERIS, G. et al. When does the uniswap amm work? In: **IEEE ICC**. [S.l.: s.n.], 2020.

Anthropic. **The Claude 3 Model Family: Opus, Sonnet, Haiku**. 2024. <https://www.anthropic.com/claude-3-model-card>. Model card, accessed 10 November 2025.

Anysphere. **Cursor Documentation**. 2024. <https://cursor.com/docs>. Accessed 10 November 2025.

ARTHUR, W. B. Inductive reasoning and bounded rationality (the el farol problem). **American Economic Review**, v. 84, n. 2, p. 406–411, 1994. Papers and Proceedings.

ARTHUR, W. B. et al. Asset pricing under endogenous expectations in an artificial stock market. In: ARTHUR, W. B.; DURLAUF, S. N.; LANE, D. A. (Ed.). **The Economy as an Evolving Complex System II**. [S.l.]: Addison-Wesley, 1997. p. 15–44.

ASARE, O.; NAGAPPAN, M.; ASOKAN, N. A user-centered security evaluation of copilot. In: **Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE ’24)**. Lisbon, Portugal: IEEE/ACM, 2024. Disponível em: <https://arxiv.org/abs/2308.06587>.

ATKINSON, A. B. On the measurement of inequality. **Journal of Economic Theory**, v. 2, n. 3, p. 244–263, 1970.

AXELROD, R. **The Evolution of Cooperation**. [S.l.]: Basic Books, 1984.

AXTELL, R. L. Why agents? on the varied motivations for agent computing in the social sciences. **Center on Social and Economic Dynamics Working Paper**, n. 17, 2000.

AZEVEDO, E. S. F.; MENDONÇA, J. R. C. Algorithmic management on digital labour platforms: A systematic review. **Contextus – Revista Contemporânea de Economia e Gestão**, 2023. Forthcoming / early access.

BANKER, R. D.; KAUFFMAN, R. J.; MOREY, R. C. Measuring gains in operational efficiency from information technology: A study of the positively sloped portion of the quality curve. **Journal of Management Information Systems**, v. 11, n. 2, p. 43–68, 1994.

BARABÁSI, A.-L.; ALBERT, R. Emergence of scaling in random networks. **Science**, v. 286, n. 5439, p. 509–512, 1999.

BERG, J. et al. **Digital Labour Platforms and the Future of Work: Towards Decent Work in the Online World**. Geneva: International Labour Office, 2018. ISBN 978-92-2-131551-0.

BOEHM, B. W. **Software Engineering Economics**. [S.l.]: Prentice Hall, 1981.

BOEHM, B. W. et al. **Software Cost Estimation with COCOMO II**. [S.l.]: Prentice Hall, 2000.

BOMMASANI, R. et al. On the opportunities and risks of foundation models. **arXiv preprint**, arXiv:2108.07258, 2021. Disponível em: <https://arxiv.org/abs/2108.07258>.

_____. The 2023 foundation model transparency index. **Transactions on Machine Learning Research**, 2024. Published version of arXiv:2310.12941. Disponível em: <https://transacl.org/ojs/index.php/tlmir/article/view/612>.

BONASSO, R. P. et al. Experiences with an architecture for intelligent, reactive agents. **Journal of Experimental & Theoretical Artificial Intelligence**, v. 9, n. 2–3, p. 237–256, 1997.

BROOKS, R. A. A robust layered control system for a mobile robot. **IEEE Journal on Robotics and Automation**, v. 2, n. 1, p. 14–23, 1986.

_____. Intelligence without representation. **Artificial Intelligence**, v. 47, p. 139–159, 1991.

BrowserGym Team. The browsergym ecosystem for web agent research. **arXiv preprint arXiv:2412.05467**, 2024.

BUCHMAN, E. **Tendermint: Byzantine Fault Tolerant State Machines**. 2018. ArXiv:1807.04938.

BUDISH, E.; CRAMTON, P.; SHIM, J. The high-frequency trading arms race: Frequent batch auctions as a market design response. **The Quarterly Journal of Economics**, v. 130, n. 4, p. 1547–1621, 2015.

_____. The high-frequency trading arms race: Frequent batch auctions as a market design response. **Quarterly Journal of Economics**, v. 130, n. 4, p. 1547–1621, 2015.

BUTERIN, V. Ethereum: A next-generation smart contract and decentralized application platform. In: **White Paper**. [S.l.: s.n.], 2014.

BUTERIN, V.; GRIFFITH, V. **Casper the Friendly Finality Gadget**. 2017.

CALVÓ-ARMENGOL, A.; JACKSON, M. O. The effects of social networks on employment and inequality. **American Economic Review**, v. 94, n. 3, p. 426–454, 2004.

CASPER, S. et al. Black-box access is insufficient for rigorous AI audits. In: **Proceedings of the 2024 ACM Conference on Fairness, Accountability, and Transparency (FAccT)**. [S.l.]: ACM, 2024. Preprint available as arXiv:2402.08787.

CHANG, S. et al. Androidworld: Benchmarking multi-modal agents on realistic mobile tasks. **arXiv preprint arXiv:2501.10830**, 2025. Alias key for consistency with your text.

CHE, Y.-K. Design competition through multidimensional auctions. **RAND Journal of Economics**, v. 24, n. 4, p. 668–680, 1993.

CHEN, J. et al. **Put Your Money Where Your Mouth Is: Evaluating Strategic Planning and Execution of LLM Agents in an Auction Arena**. 2024. ArXiv:2310.05746.

CHEN, M. et al. Evaluating large language models trained on code. **arXiv:2107.03374**, 2021.

COASE, R. H. The nature of the firm. **Economica**, v. 4, n. 16, p. 386–405, 1937.

COLLEDANCHISE, M.; ÖGREN, P. **Behavior Trees in Robotics and AI**. [S.l.]: CRC Press, 2018.

CONTRIBUTORS, C. S. **Cosmos SDK Documentation**. 2023. Https://docs.cosmos.network.

COSENTINO, V.; IZQUIERDO, J. L. C.; CABOT, J. A systematic mapping study of software development with GitHub. **IEEE Access**, v. 5, p. 7173–7192, 2017.

COWELL, F. A. **Measuring Inequality**. 3. ed. [S.l.]: Oxford University Press, 2011.

CRAMTON, P.; SHOHAM, Y.; STEINBERG, R. (Ed.). **Combinatorial Auctions**. [S.l.]: MIT Press, 2006.

CUNNINGHAM, W. **The WyCash Portfolio Management System**. 1992. OOPSLA ’92 Experience Report. Introduces the “technical debt” metaphor.

DAIAN, P. et al. Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability. In: **IEEE S&P**. [S.l.: s.n.], 2020.

______. Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability. In: **IEEE Symposium on Security and Privacy**. [S.l.: s.n.], 2020.

DIJKSTRA, E. W. **A Discipline of Programming**. [S.l.]: Prentice Hall, 1976.

DOHMATOB, E.; PEZESHKI, M.; ASKARI-HEMMAT, R. Why less is more (sometimes): A theory of data curation. **arXiv preprint**, arXiv:2511.03492, 2025. Disponível em: <https://arxiv.org/abs/2511.03492>.

DU, Y. et al. Improving factuality and reasoning in language models through multiagent debate. In: **ICLR**. [S.l.: s.n.], 2023.

EASLEY, D.; KLEINBERG, J. **Networks, Crowds, and Markets: Reasoning About a Highly Connected World**. [S.l.]: Cambridge University Press, 2010.

EINAV, L.; FARRONATO, C.; LEVIN, J. Peer-to-peer markets. **Annual Review of Economics**, v. 8, p. 615–635, 2016.

EPSTEIN, J. M.; AXTELL, R. **Growing Artificial Societies: Social Science from the Bottom Up**. [S.l.]: Brookings Institution Press / MIT Press, 1996.

EYAL, I.; SIRER, E. G. Majority is not enough: Bitcoin mining is vulnerable. In: **Financial Cryptography**. [S.l.: s.n.], 2014. p. 436–454.

FAN, L. et al. Minedojo: Building open-ended embodied agents with internet-scale knowledge. In: **NeurIPS**. [S.l.: s.n.], 2022.

FARMER, J. D.; GEANAKOPLOS, J. The economy needs agent-based modelling. **Nature**, v. 460, p. 685–686, 2009.

FEILDEN, E.; OLTEAN, A.; JOHNSTON, P. **Why We Should Train AI in Space**. 2024. <https://starcloudinc.github.io/wp.pdf>. White paper, Lumen Orbit (now Starcloud), accessed 10 November 2025.

FENTON, N.; BIEMAN, J. **Software Metrics: A Rigorous and Practical Approach**. 3. ed. [S.l.]: CRC Press, 2014.

FIKES, R. E.; NILSSON, N. J. Strips: A new approach to the application of theorem proving to problem solving. In: **Artificial Intelligence**. [S.l.: s.n.], 1971. v. 2, n. 3–4, p. 189–208.

FOUNDATION, I. **Inter-Blockchain Communication Protocol (IBC) Specification**. 2020. Https://github.com/cosmos/ibc.

GABAIX, X. Power laws in economics and finance. **Annual Review of Economics**, v. 1, p. 255–294, 2009.

GARAY, J. A.; KIAYIAS, A.; LEONARDOS, N. The bitcoin backbone protocol: Analysis and applications. In: **EUROCRYPT**. [S.l.: s.n.], 2015. p. 281–310.

GAT, E. On three-layer architectures. In: KORTENKAMP, D.; BONASSO, R. P.; MURPHY, R. (Ed.). **Artificial Intelligence and Mobile Robots**. [S.l.]: MIT Press, 1998. p. 195–210.

GIL, Y. et al. Examining the challenges of scientific workflows. In: **IEEE Computer**. [S.l.: s.n.], 2007.

GINI, C. Variabilità e mutabilità. **Studi Economico-Giuridici della R. Università di Cagliari**, 1912.

Gitcoin. **Gitcoin Grants and Bounties Documentation**. 2022. <https://gitcoin.co>. Platform documentation, accessed 10 November 2025.

GitHub. **GitHub Copilot: Your AI Pair Programmer**. 2023. <https://github.com/features/copilot>. Accessed 10 November 2025.

GRAY, M. L.; SURI, S. **Ghost Work: How to Stop Silicon Valley from Building a New Global Underclass**. New York: Houghton Mifflin Harcourt, 2019. ISBN 978-1328566249.

GUO, D. et al. Deepseek-coder: When the large language model meets programming. **arXiv preprint**, arXiv:2401.14196, 2024. Disponível em: <https://arxiv.org/abs/2401.14196>.

HAREL, D. Statecharts: A visual formalism for complex systems. **Science of Computer Programming**, v. 8, n. 3, p. 231–274, 1987.

HENDERSON, P. et al. **Towards the Systematic Reporting of the Energy and Carbon Footprints of Machine Learning**. 2020.

HOARE, C. A. R. An axiomatic basis for computer programming. **Communications of the ACM**, v. 12, n. 10, p. 576–580, 1969.

HOLLAND, J. H. **Hidden Order: How Adaptation Builds Complexity**. [S.l.]: Addison-Wesley, 1995.

HOLMSTRÖM, B.; MILGROM, P. Multitask principal–agent analyses: Incentive contracts, asset ownership, and job design. **Journal of Law, Economics, & Organization**, v. 7, n. Special Issue, p. 24–52, 1991.

HONG, W. et al. **MetaGPT: Meta Programming for Multi-Agent Collaborative Framework**. 2023.

HOU, Y. et al. A systematic review of large language models for software engineering. **ACM Transactions on Software Engineering and Methodology**, 2024.

HUMBLE, J.; FARLEY, D. **Continuous Delivery**. [S.l.]: Addison-Wesley, 2010.

HURWICZ, L. On informationally decentralized systems. In: MCGUIRE, C. B.; RADNER, R. (Ed.). **Decision and Organization**. Amsterdam: North-Holland, 1972. p. 297–336.

JACKSON, M. O. **Social and Economic Networks**. [S.l.]: Princeton University Press, 2008.

______. **Social and Economic Networks**. [S.l.]: Princeton University Press, 2010.

JENNINGS, N. R. On agent-based software engineering. **Artificial Intelligence**, v. 117, n. 2, p. 277–296, 2000.

JENNINGS, N. R.; SYCARA, K.; WOOLDRIDGE, M. A roadmap of agent research and development. **Autonomous Agents and Multi-Agent Systems**, v. 1, n. 1, p. 7–38, 1998.

JIMENEZ, C. E. et al. SWE-bench: Can language models resolve real-world github issues? In: **International Conference on Learning Representations (ICLR)**. [s.n.], 2024. Disponível em: <https://arxiv.org/abs/2310.06770>.

JR., F. P. B. **The Mythical Man-Month: Essays on Software Engineering**. [S.l.]: Addison-Wesley, 1975.

KAELBLING, L. P.; LITTMAN, M. L.; CASSANDRA, A. R. Planning and acting in partially observable stochastic domains. **Artificial Intelligence**, v. 101, n. 1–2, p. 99–134, 1998.

KALLIAMVAKOU, E. **Research: quantifying GitHub Copilot's impact on developer productivity and happiness**. 2022. GitHub Blog. Accessed 2025-11-01. Disponível em: <https://github.blog/news-insights/research/research-quantifying-github-copilots-impact-on-developer-productivity-and-happiness/>.

KALLIAMVAKOU, E. et al. The promises and perils of mining GitHub. In: **Proceedings of the 11th Working Conference on Mining Software Repositories (MSR)**. [S.l.]: ACM, 2014. p. 92–101.

KIAYIAS, A. et al. Ouroboros: A provably secure proof-of-stake blockchain protocol. In: **CRYPTO**. [S.l.: s.n.], 2017. p. 357–388.

KITANO, H.; VELOSO, M. et al. Robocup: The robot world cup initiative. **AI Magazine**, v. 18, n. 1, p. 73–85, 1997.

KITCHENHAM, B. et al. Systematic literature reviews in software engineering. **Information and Software Technology**, v. 51, n. 1, p. 7–15, 2009.

KLEMPERER, P. **Auctions: Theory and Practice**. [S.l.]: Princeton University Press, 2004.

KLEPPMANN, M. **Designing Data-Intensive Applications**. [S.l.]: O'Reilly, 2017.

KOH, J. et al. Visualwebarena: Evaluating multimodal agents on realistic visual web tasks. In: **Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL)**. [S.l.: s.n.], 2024.

KRISHNA, V. **Auction Theory**. 2. ed. [S.l.]: Academic Press, 2009.

_____. Auction theory. **Academic Press**, 2009. 2nd ed., book.

KWON, J. **Tendermint: Consensus without Mining**. 2014. Whitepaper.

KWON, J.; BUCHMAN, E. **Cosmos: A Network of Distributed Ledgers**. 2016. White paper.

LAFFONT, J.-J.; MARTIMORT, D. **The Theory of Incentives: The Principal–Agent Model**. [S.l.]: Princeton University Press, 2002.

LAIRD, J. E.; NEWELL, A.; ROSENBLOOM, P. S. Soar: An architecture for general intelligence. **Artificial Intelligence**, v. 33, n. 1, p. 1–64, 1987.

LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. **Communications of the ACM**, v. 21, n. 7, p. 558–565, 1978.

LAW, A. M. **Simulation Modeling and Analysis**. 5. ed. [S.l.]: McGraw–Hill, 2015.

LI, C. et al. Camel: Communicative agents for “mind” exploration of large language model society. In: **NeurIPS**. [S.l.: s.n.], 2023.

______. Behavior-1k: A human-centered, embodied AI benchmark with 1,000 everyday activities and realistic simulation. **arXiv preprint arXiv:2403.09227**, 2024.

LI, R.; WERRA, L. von; AL., T. W. et. **StarCoder: may the source be with you!** 2023.

MASANET, E. et al. Recalibrating global data center energy-use estimates. **Science**, v. 367, n. 6481, p. 984–986, 2020.

MCDERMOTT, D. Pddl—the planning domain definition language. In: **AIPS**. [S.l.: s.n.], 1998.

MERKEL, D. Docker: Lightweight linux containers for consistent development and deployment. **Linux Journal**, n. 239, 2014.

Microsoft. **Visual Studio with GitHub Copilot: AI Pair Programming**. 2023. <https://visualstudio.microsoft.com/github-copilot/>. Accessed 10 November 2025.

MONTGOMERY, D. C. **Design and Analysis of Experiments**. 9. ed. Hoboken: Wiley, 2017.

MYERSON, R. B. Optimal auction design. **Mathematics of Operations Research**, v. 6, n. 1, p. 58–73, 1981.

NAKAMOTO, S. Bitcoin: A peer-to-peer electronic cash system. In: **Cryptography Mailing List**. [S.l.: s.n.], 2008.

NAUR, P.; RANDELL, B. Software engineering: Report on a conference sponsored by the nato science committee. In: **NATO Software Engineering Report**. [S.l.]: NATO, 1969.

NIAZI, M. A.; HUSSAIN, A. Agent-based computing from multi-agent systems to agent-based models: A visual survey. **Scientometrics**, v. 89, n. 2, p. 479–499, 2011.

OBERKAMPF, W. L.; ROY, C. J. **Verification and Validation in Scientific Computing**. Cambridge: Cambridge University Press, 2010.

OmniGibson Contributors. **OmniGibson: A Development Platform for Embodied AI in Photorealistic Scenes**. 2024. <https://github.com/StanfordVL/OmniGibson>.

OPENAI. **GPT-4 Technical Report**. 2023.

OpenHands Contributors. **OpenHands: Code Less, Make More**. 2025. <https://github.com/OpenHands/OpenHands>. MIT-licensed project repository (formerly OpenDevin).

OUYANG, S. et al. An empirical study of the non-determinism of chatgpt in code generation. **arXiv preprint arXiv:2308.02828**, 2024.

PARNAS, D. L. On the criteria to be used in decomposing systems into modules. **Communications of the ACM**, v. 15, n. 12, p. 1053–1058, 1972.

PEARCE, H. et al. Asleep at the keyboard? assessing the security of github copilot's code contributions. **arXiv preprint arXiv:2108.09293**, 2021. Journal version: *Communications of the ACM*, 68(2), 2025. doi:10.1145/3610721. Disponível em: <https://arxiv.org/abs/2108.09293>.

PENG, S. et al. The impact of ai on developer productivity: Evidence from github copilot. **arXiv preprint arXiv:2302.06590**, 2023. Disponível em: <https://arxiv.org/abs/2302.06590>.

PERRY, N. et al. Do users write more insecure code with ai assistants? In: **Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23)**. Copenhagen, Denmark: ACM, 2023. Disponível em: <https://dl.acm.org/doi/10.1145/3576915.3623157>.

PUTNAM, L. H.; MYERS, W. **Measures for Excellence: Reliable Software On Time, Within Budget**. [S.l.]: Prentice Hall, 1992.

Radicle Team. **Radicle: A Peer-to-Peer Stack for Code Collaboration**. 2021. <https://radicle.xyz>. Project documentation, accessed 10 November 2025.

RAILSBACK, S. F.; GRIMM, V. **Agent-Based and Individual-Based Modeling: A Practical Introduction**. 2. ed. [S.l.]: Princeton University Press, 2019.

RAO, A. S.; GEORGEFF, M. P. Bdi agents: From theory to practice. In: **ICMAS**. [S.l.: s.n.], 1995. p. 312–319.

ROBBINS, L. **An Essay on the Nature and Significance of Economic Science**. London: Macmillan, 1932.

ROUGHGARDEN, T. Transaction fee mechanism design for the ethereum blockchain: An economic analysis of eip-1559. **ACM SIGecom Exchanges**, v. 19, n. 1, p. 52–60, 2021.

ROZIÈRE, B.; GEHRING, J.; AL., F. N. et. **Code Llama: Open Foundation Models for Code**. 2023.

SAAD-FALCON, J. et al. Intelligence per watt: Measuring intelligence efficiency of local ai. **arXiv preprint arXiv:2511.07885**, 2025. Disponível em: <https://arxiv.org/abs/2511.07885>.

SAMUELSON, P. A.; NORDHAUS, W. D. **Economics**. 19. ed. New York: McGraw-Hill, 2010.

SARGENT, R. G. Verification and validation of simulation models. In: **Proceedings of the 2013 Winter Simulation Conference (WSC)**. [S.l.]: IEEE/ACM, 2013. p. 1–12.

SCAO, T. L. et al. BLOOM: A 176b-parameter open-access multilingual language model. **arXiv preprint**, arXiv:2211.05100, 2022. Disponível em: <https://arxiv.org/abs/2211.05100>.

SCHELLING, T. C. Dynamic models of segregation. **Journal of Mathematical Sociology**, v. 1, n. 2, p. 143–186, 1971.

_____. Dynamic models of segregation. **Journal of Mathematical Sociology**, v. 1, n. 2, p. 143–186, 1971.

SHAH, D. et al. **Language Models as Auction Participants**. 2025. ArXiv:2507.09083.

SHEN, Y. et al. **HuggingGPT: Solving AI Tasks with ChatGPT and its Friends in HuggingFace**. 2023.

SHOHAM, Y.; LEYTON-BROWN, K. **Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations**. [S.l.]: Cambridge University Press, 2009.

SIGELMAN, B. H. et al. **Dapper, a Large-Scale Distributed Systems Tracing Infrastructure**. 2010. Technical report, Google.

STRUBELL, E.; GANESH, A.; MCCALLUM, A. Energy and policy considerations for deep learning in NLP. In: **ACL**. [S.l.: s.n.], 2019. p. 3645–3650.

SUTTON, R. S.; BARTO, A. G. **Reinforcement Learning: An Introduction**. 2. ed. [S.l.]: MIT Press, 2018.

SWE-agent Contributors. **SWE-agent**. 2025. <https://github.com/SWE-agent/SWE-agent>. MIT-licensed research codebase; documentation and releases linked from repository.

SWE-AGENT Documentation. 2025. <https://swe-agent.com/>. Getting started, benchmarking guides, and configuration reference.

TADELIS, S. Reputation and feedback systems in online platform markets. **Annual Review of Economics**, v. 8, p. 321–340, 2016.

TESFATSION, L. Agent-based computational economics: Growing economies from the bottom up. **Artificial Life**, v. 8, n. 1, p. 55–82, 2002.

_____. Agent-based computational economics. In: JUDD, K. L.; TESFATSION, L. (Ed.). **Handbook of Computational Economics, Vol. 2**. [S.l.]: Elsevier, 2006. p. 831–880.

_____. **ACE (Agent-Based Computational Economics) Overview and Annotated Guide**. 2022. <https://www2.econ.iastate.edu/tesfatsi/ace.htm>. Accessed 2025-11-01.

TIAN, H. et al. Is chatgpt the ultimate programming assistant – how far is it? **arXiv preprint arXiv:2304.11938**, 2023.

TOMASEV, N. et al. Virtual agent economies. **arXiv preprint arXiv:2509.10147**, 2025.

TRIVEDI, H. et al. Appworld: A controllable world of apps and people for benchmarking interactive coding agents. In: **International Conference on Learning Representations (ICLR)**. [S.l.: s.n.], 2024.

VARIAN, H. R. **Intermediate Microeconomics: A Modern Approach**. 9. ed. New York: W. W. Norton, 2014.

VASWANI, A. et al. Attention is all you need. In: **NIPS**. [S.l.: s.n.], 2017.

WANG, G. et al. **Voyager: An Open-Ended Embodied Agent with Large Language Models**. 2023.

WANG, X. et al. Openhands: An open platform for AI software developers as generalist agents. In: **International Conference on Learning Representations (ICLR)**. [s.n.], 2025. See also arXiv:2407.16741, DOI:10.48550/arXiv.2407.16741. Disponível em: <https://openreview.net/forum?id=OJd3ayDDoF>.

WANG, Z.; LI, Y. et al. OSWorld: Benchmarking computer-use agents on 369 realistic desktop tasks. **arXiv preprint arXiv:2404.07972**, 2024.

WELLMAN, M. P. Methods for empirical game-theoretic analysis. In: **AAAI Workshop on Game Theoretic and Decision Theoretic Agents**. [S.l.: s.n.], 2006.

WILENSKY, U. **NetLogo**. 1999. Center for Connected Learning and Computer-Based Modeling, Northwestern University.

WILSON, G. et al. Good enough practices in scientific computing. **PLOS Computational Biology**, v. 13, n. 6, p. e1005510, 2017.

WOHLIN, C. et al. **Experimentation in Software Engineering**. Berlin: Springer, 2012.

WOOD, G. **Ethereum: A Secure Decentralised Generalised Transaction Ledger**. 2014. Yellow Paper.

WOOLDRIDGE, M. **Reasoning about Rational Agents**. [S.l.]: MIT Press, 2000.

_____. **An Introduction to MultiAgent Systems**. 2. ed. [S.l.]: Wiley, 2009.

WU, T.; YANG, C.; AL., S. Z. et. **AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation**. 2023.

XIE, T. Intelligent software engineering. In: **IEEE/ACM International Conference on Software Engineering (ICSE) — ISEC Extended Abstract**. [S.l.: s.n.], 2018. Short vision/position piece.

YANG, J. et al. SWE-agent: Agent-computer interfaces enable automated software engineering. In: **Advances in Neural Information Processing Systems (NeurIPS)**. [s.n.], 2024. See also arXiv:2405.15793. Disponível em: <https://proceedings.neurips.cc/paper_files/paper/2024/file/5a7c947568c1b1328ccc5230172e1e7c-Paper-Conference.pdf>.

YANG, K. et al. **SWEBench: Evaluating LLMs on Real-World Software Issues**. 2023.

YANG, K.; ZHAI, C. Ten principles of AI agent economics. **arXiv preprint arXiv:2505.20273**, 2025.

YANG, Y. et al. Agent exchange: Shaping the future of AI agent economics. **arXiv preprint arXiv:2507.03904**, 2025.

YAO, S. et al. Webshop: Towards scalable real-world web interaction with language models. **arXiv preprint arXiv:2207.01206**, 2022.

_____. React: Synergizing reasoning and acting in language models. In: **ICLR**. [S.l.: s.n.], 2023.

YETIŞTIREN, B. et al. Evaluating the code quality of ai-assisted code generation tools: An empirical study on github copilot, amazon codewhisperer, and chatgpt. **arXiv preprint arXiv:2304.10778**, 2023. Disponível em: <https://arxiv.org/abs/2304.10778>.

ZHANG, F. et al. Large language models for software engineering: A survey. **arXiv:2310.03533**, 2023.

ZHOU, S. et al. Webarena: A realistic web environment for building autonomous agents. In: **International Conference on Learning Representations (ICLR)**. [s.n.], 2024. Disponível em: <https://proceedings.iclr.cc/paper_files/paper/2024/file/4410c0711e9154a7a2d26f9b3816d1ef-Paper-Conference.pdf>.

Appendix

APPENDIX A

Simulation Experiment A

A.1 Overview

This appendix documents all technical steps required to reproduce Simulation A, including data generation, chain initialization, agent execution, economic and network metric extraction, and Simulation A-specific specialization analysis. All commands and artifacts correspond to the public repository.

Simulation A varies one policy dimension: whether SWE-Agents behave as generalists or as specialists with private tag sets and small in-niche/out-of-niche cost multipliers. The baseline configuration treats all agents as generalists; the intervention configuration enables specialization in the trial data. All other elements, including dataset, agent pool, random seeds, auction rules, and execution timing, are held fixed. Thus, observed differences reflect the marginal effect of comparative advantage via specialization under otherwise identical conditions.

Simulation A uses two layers of results:

1. an experiment-independent layer of economic and network metrics, extracted by the tool `trial_card` (completion, cost, inequality, and participation structure) from each run's on-chain auctions and bids; and
2. a Simulation A-specific specialization visualization, extracted by the tool `simA_results` from the canonical `trial_card.json` files for baseline and intervention.

The dependency is one-way: `trial_card` in compute mode talks to the local `swechaind` node and writes canonical, simulation-agnostic `trial_card.json` files that contain both aggregate metrics and auction- and bid-level microdata. `simA_results` is a pure offline tool that reads these JSON files to produce the Simulation A specialization figure. The compare mode of `trial_card` also reads the same `trial_card.json` files to produce a paired economic trial card. Neither compare mode nor `simA_results` accesses the chain.

In the specialization figure for Simulation A, the focus is on how enabling specialist tags changes who wins which tasks and how often auctions clear.

A.2 Data Engineering

Simulation A uses a deterministic dataset snapshot `data/problems.json`. Paired trial files are generated using:

```
./bin/sdge --paired \
  -problems-file ./data/problems.json \
  -trials 1 -problems 200 -agents 20 -seed 1111 \
  -specialist -spec-tags 2 \
  -output-dir ./simA/data
```

This emits, with identical task order and agent pool:

❏ `simA/data/baseline/trial_001.json`

❏ `simA/data/intervention/trial_001.json`

The baseline and intervention files share the same sampled tasks and sequence; the specialization intervention is encoded in the trial metadata (private tag sets and cost multipliers), not by changing the agent finite-state machine.

Each condition also has a manifest listing trial files and hashes. These manifests support checking that the paired datasets match and that no files changed between runs.

A.3 Host Execution Environment

Simulation A was executed on:

❏ Debian GNU/Linux 12 (x86_64)

❏ AMD Ryzen 7 5800H, 27 GiB RAM

❏ Go 1.24.7, Git 2.39.5

❏ Ollama 0.12.9 running model `gpt-oss:120b-cloud`

Language model sampling configuration (temperature, top-p, maximum tokens) is pinned in the agent specification file and reused for both conditions. Any reproduction should use the same model identifier and similar hardware, or at least document the differences.

A.4 Blockchain Configuration

Simulation A uses a dedicated SWEChain instance with fixed:

- ❏ genesis file and chain identifier;
- ❏ account balances and initial token distribution;
- ❏ software outsourcing auction parameters (duration, grace periods, tick size);
- ❏ consensus and gas settings.

The experiment requires that baseline and intervention runs use identical chain parameters. This is enforced by exporting and comparing parameter snapshots before each run.

A.5 Agent Configuration

Simulation A uses a single agent specification:

- ❏ `simA/config/agent.json`

The same agent configuration is used for both baseline and intervention. The difference between conditions comes from the data: in the intervention trial file, agents receive private specialization tags and cost multipliers in their local state and prompts; in the baseline trial file, they behave as generalists. Finite-state machine, retry policies, memory filters, and language model settings are identical across arms.

The finite-state machine makes the agent behavior explicit and ensures that any differences in outcomes can be traced to the presence or absence of specialization in the data rather than unrelated control-flow changes.

A.6 Reproducibility Protocol

This section enumerates the steps needed to regenerate all Simulation A artifacts, including runs, economic and network metrics, the paired economic trial card, and the Simulation A-specific specialization figure. All paths are relative to the repository root.

Phase 1 — Build Binaries

```
go build -o bin/sdge           ./src/sdge.go
go build -o bin/trial          ./src/trial.go
go build -o bin/swechain_agent ./src/swechain_agent.go
go build -o bin/trial_card     ./src/trial_card.go
go build -o bin/simA_results   ./simA/src/simA_results.go
```

This produces the deterministic data generator (`sdge`), the trial orchestrator (`trial`), the agent binary (`swechain_agent`), the canonical metrics tool (`trial_card`), and the Simulation A-specific specialization tool (`simA_results`).

Phase 2 — Generate Paired Trials

```
./bin/sdge --paired \
  -problems-file ./data/problems.json \
  -trials 1 -problems 200 -agents 20 -seed 1111 \
  -specialist -spec-tags 2 \
  -output-dir ./simA/data
```

This produces:

❏ `simA/data/baseline/trial_001.json`

❏ `simA/data/intervention/trial_001.json`

plus manifests that record which trial files belong to each condition. The baseline and intervention trial files share task order and agent lists by construction.

Phase 3 — Initialize SWEChain for Baseline

```
./start_swechain.sh 20
swechaind query auction params -o json \
  > simA/baseline_params.json
```

The `start_swechain.sh` script starts a local SWEChain node with 20 validators and a fixed configuration. The exported parameters in `simA/baseline_params.json` are used later to confirm that the intervention run uses the same mechanism.

Phase 4 — Run the Baseline Trial

```
./bin/trial -minutes 20 \
  -agent-bin ./bin/swechain_agent \
  -spec ./simA/config/agent.json \
  -trial ./simA/trials/baseline \
  -dbdir ./simA/trials/baseline_db \
  ./simA/data/baseline/trial_001.json
```

This runs the baseline configuration (generalists) for 20 minutes of wall-clock time, producing per-agent databases and journals under `simA/trials/baseline_db/` and all on-chain auction and bid events for the baseline condition.

Phase 5 — Baseline Economic and Network Metrics (`trial_card` Compute Mode)

While the baseline chain state is still present, compute the canonical economic and network metrics and microdata:

```
./bin/trial_card \
  --bin swechaind \
  --node tcp://localhost:26657 \
  --home "${HOME}/.swechain" \
  --outdir ./simA/artifact/baseline \
  --output text
```

This writes:

❏ `simA/artifact/baseline/trial_card.json`

❏ `simA/artifact/baseline/trial_card.svg`

❏ `simA/artifact/baseline/trial_card.pdf`

The single-run baseline card is copied as `figs/simA_baseline_trial_card.pdf` for use in the dissertation figures. The JSON file `trial_card.json` provides the canonical baseline metrics and microdata for all downstream paired analyses.

Phase 6 — Reset Chain for Intervention

Before running the intervention condition, restart the chain and verify that the mechanism parameters match the baseline:

```
./start_swechain.sh 20
swechaind query auction params -o json \
  > simA/intervention_params.json
diff -u simA/baseline_params.json \
       simA/intervention_params.json
```

The `diff` command should report no differences. Any discrepancy indicates a configuration drift that must be fixed before proceeding.

Phase 7 — Run the Intervention Trial

```
./bin/trial -minutes 20 \
  -agent-bin ./bin/swechain_agent \
  -spec ./simA/config/agent.json \
  -trial ./simA/trials/intervention \
  -dbdir ./simA/trials/intervention_db \
  ./simA/data/intervention/trial_001.json
```

This uses the same dataset, agents, random seeds, and timing as the baseline run, but with specialization enabled in `simA/data/intervention/trial_001.json` (private tag sets and cost multipliers). The resulting on-chain history is the intervention counterpart of the baseline history.

Phase 8 — Intervention Economic and Network Metrics (`trial_card` Compute Mode)

While the intervention chain state is still present, compute the canonical metrics and microdata:

```
./bin/trial_card \
  --bin swechaind \
  --node tcp://localhost:26657 \
  --home "${HOME}/.swechain" \
  --outdir ./simA/artifact/intervention \
  --output text
```

This writes:

- `simA/artifact/intervention/trial_card.json`
- `simA/artifact/intervention/trial_card.svg`
- `simA/artifact/intervention/trial_card.pdf`

The single-run intervention card is copied as `figs/simA_intervention_trial_card.pdf` for use in the dissertation figures. At this point, both conditions have their own `trial_card.json` files, generated under identical chain parameters.

Phase 9 — Paired Economic Comparison (`trial_card` Compare Mode)

With both `trial_card.json` files in place, the paired baseline–intervention economic trial card is generated offline, without accessing the chain:

```
./bin/trial_card --compare \
  --baseline ./simA/artifact/baseline/trial_card.json \
  --intervention ./simA/artifact/intervention/trial_card.json \
  --outdir ./simA/artifact/compare \
  --label "Simulation A - Agent specialization" \
  --output text
```

This writes:

- `simA/artifact/compare/trial_pair.json`
- `simA/artifact/compare/trial_pair.svg`
- `simA/artifact/compare/trial_pair.pdf`

The paired economic trial card is copied as `figs/simA_trial_pair.pdf` for use in the dissertation figures.

Phase 10 — Specialization Visualization (`simA_results`)

Finally, the Simulation A-specific specialization figure is generated offline from the same canonical metrics files. The `simA_results` program reads the baseline and intervention `trial_card.json` files under `simA/artifact/` and emits an SVG that is converted to PDF via `rsvg-convert`:

```
./bin/simA_results \
  -artifact-dir ./simA/artifact \
  -out-pdf     ./simA/artifact/compare/simA_results.pdf
```

This writes:

❏ `simA/artifact/compare/simA_results.svg`

❏ `simA/artifact/compare/simA_results.pdf`

The PDF is copied as `figs/simA_results.pdf` for use in the dissertation figures. Because `simA_results` operates purely on the `trial_card.json` microdata, this phase does not depend on any particular chain height being kept alive.

A.7 Limits, Assumptions, and Validation

Simulation A enforces that the only policy change between conditions is the presence of private specialization tags and cost multipliers in the trial data. All other parameters, including dataset, seeds, agent pool, prompts, auction rules, timing, and chain parameters, are held fixed.

Language model stochasticity means individual token sequences may differ across reruns, even with the same seeds and configuration. However, the recorded configuration and protocol are strong enough to reproduce the experimental setup and to check whether the qualitative and quantitative patterns in task completion, assortativity, outsourcing cost, and revenue concentration persist.

Validation consists of:

1. checking that baseline and intervention trial files share identical task order and agent lists, and that the manifests show the expected pairing;

2. verifying that `simA/baseline_params.json` and `simA/intervention_params.json` are identical, confirming that the blockchain mechanism did not change across conditions;

3. recomputing economic and network metrics via `trial_card` for both conditions and confirming that the regenerated single-run cards match the reported values up to ordinary numeric and rendering tolerances;

4. recomputing the paired economic trial card via `trial_card` in compare mode and the specialization figure via `simA_results` from the stored `trial_card.json` files, and confirming that the observed patterns in completion, cost, and revenue concentration are consistent with those in the dissertation figures;

5. for the specialization figure specifically, checking that the counts in any summary box (numbers of auctions, auctions with bids, bidders, and paired problems) match those implied by the underlying `trial_card.json` microdata, and that any tag–winner patterns agree with the recorded allocations and revenues.

Under this protocol, a reader with access to the public repository, the same model identifier, and a similar machine can rerun Simulation A, regenerate both the economic trial cards and the Simulation A specialization figure from the canonical JSON artifacts, and verify the main economic and network findings associated with enabling SWE-Agent specialization.

APPENDIX B

Simulation Experiment B

B.1 Overview

This appendix documents all technical steps required to reproduce Simulation B, including data generation, chain initialization, agent execution, economic and network metric extraction, and Simulation B-specific bid-landscape analysis. All commands and artifacts correspond to the public repository.

Simulation B varies one policy dimension: whether bidding prompts explicitly expose and emphasize on-chain price signals. The baseline configuration keeps the current best admissible bid in the input schema but does not highlight it in the prompts; the intervention configuration adds a dedicated price-signal block and instructs bidders to use it. All other elements, including dataset, agent pool, random seeds, auction rules, and execution timing, are held fixed. Thus, observed differences reflect the marginal effect of price-signal utilization under otherwise identical conditions.

Simulation B uses two layers of results:

1. an experiment-independent layer of economic and network metrics, extracted by the tool `trial_card` (completion, cost, inequality, and participation structure) from each run's on-chain auctions and bids; and

2. a Simulation B-specific paired bid-landscape visualization, extracted by the tool `simB_results` from the canonical `trial_card.json` files for baseline and intervention.

The dependency is one-way: `trial_card` in compute mode talks to the local `swechaind` node and writes canonical, simulation-agnostic `trial_card.json` files that contain both aggregate metrics and bid-level microdata. `simB_results` is a pure offline tool that reads these JSON files to produce the Simulation B bid-landscape figure. The compare mode of `trial_card` also reads the same `trial_card.json` files to produce a paired economic trial card. Neither compare mode nor `simB_results` accesses the chain.

In the bid-landscape figure for Simulation B, horizontal position always represents bid or winning price, measured in SWEChain tokens per task. The vertical axis is categorical: separate horizontal bands are used to distinguish baseline and intervention, and non-winning bids are given small vertical jitter within their band purely for visibility. There is no second quantitative axis beyond price, which avoids over-interpreting vertical placement inside each band.

B.2 Data Engineering

Simulation B uses a deterministic dataset snapshot `data/problems.json`. Paired trial files are generated using:

```
./bin/sdge --paired \
  -problems-file ./data/problems.json \
  -trials 1 -problems 200 -agents 20 -seed 1111 \
  -output-dir ./simB/data
```

This emits:

- `simB/data/baseline/trial_001.json`
- `simB/data/intervention/trial_001.json`

with identical task order and agent pool but separate configuration metadata. The file-level metadata makes the condition labels explicit and supports checking that the paired datasets match.

B.3 Host Execution Environment

Simulation B was executed on:

- Debian GNU/Linux 12 (x86_64)
- AMD Ryzen 7 5800H, 27 GiB RAM
- Go 1.24.7, Git 2.39.5
- Ollama 0.12.9 running model `gpt-oss:120b-cloud`

Language model sampling configuration (temperature, top-p, maximum tokens) is pinned in the agent specification files and used for both conditions. Any reproduction should use the same model identifier and similar hardware, or at least document the differences.

B.4 Blockchain Configuration

Simulation B uses a dedicated SWEChain instance with fixed:

- ❏ genesis file and chain identifier;
- ❏ account balances and initial token distribution;
- ❏ software outsourcing auction parameters (duration, grace periods, tick size);
- ❏ consensus and gas settings.

The experiment requires that baseline and intervention runs use identical chain parameters. This is enforced by exporting and comparing parameter snapshots before each run.

B.5 Agent Configuration

Two agent specifications are used:

- ❏ `simB/config/agent_B_baseline.json`
- ❏ `simB/config/agent_B_intervention.json`

They differ only in the presence of an explicit price-signal block in the bidding prompts. Both share the same finite-state machine, retry policies, memory filters, and language model settings.

The finite-state machines make the agent behavior explicit and ensure that any differences in outcomes can be traced to the presence or absence of price-signal information rather than unrelated control-flow changes.

B.6 Reproducibility Protocol

This section enumerates the steps needed to regenerate all Simulation B artifacts, including runs, economic and network metrics, the paired economic trial card, and the Simulation B-specific bid-landscape figure. All paths are relative to the repository root.

Phase 1 — Build Binaries

```
go build -o bin/sdge           ./src/sdge.go
go build -o bin/trial          ./src/trial.go
go build -o bin/swechain_agent ./src/swechain_agent.go
go build -o bin/trial_card     ./src/trial_card.go
go build -o bin/simB_results   ./simB/src/simB_results.go
```

This produces the deterministic data generator (`sdge`), the trial orchestrator (`trial`), the agent binary (`swechain_agent`), the canonical metrics tool (`trial_card`), and the Simulation B-specific bid-landscape tool (`simB_results`).

Phase 2 — Generate Paired Trials

```
./bin/sdge --paired \
  -problems-file ./data/problems.json \
  -trials 1 -problems 200 -agents 20 -seed 1111 \
  -output-dir ./simB/data
```

This produces:

- ❏ `simB/data/baseline/trial_001.json`
- ❏ `simB/data/intervention/trial_001.json`

plus manifests that record which trial files belong to each condition. The baseline and intervention trial files share task order and agent lists by construction.

Phase 3 — Initialize SWEChain for Baseline

```
./start_swechain.sh 20
swechaind query auction params -o json \
  > simB/baseline_params.json
```

The `start_swechain.sh` script starts a local SWEChain node with 20 validators and a fixed configuration. The exported parameters in `simB/baseline_params.json` are used later to confirm that the intervention run uses the same mechanism.

Phase 4 — Run the Baseline Trial

```
./bin/trial -minutes 20 \
  -agent-bin ./bin/swechain_agent \
  -spec ./simB/config/agent_B_baseline.json \
  -trial ./simB/trials/baseline \
  -dbdir ./simB/trials/baseline_db \
  ./simB/data/baseline/trial_001.json
```

This runs the baseline configuration for 20 minutes of wall-clock time, producing per-agent databases and journals under `simB/trials/baseline_db/` and all on-chain auction and bid events for the baseline condition.

Phase 5 — Baseline Economic and Network Metrics (`trial_card` Compute Mode)

While the baseline chain state is still present, compute the canonical economic and network metrics and microdata:

```
./bin/trial_card \
  --bin swechaind \
  --node tcp://localhost:26657 \
  --home "${HOME}/.swechain" \
  --outdir ./simB/artifact/baseline \
  --output text
```

This writes:

❏ `simB/artifact/baseline/trial_card.json`

❏ `simB/artifact/baseline/trial_card.svg`

❏ `simB/artifact/baseline/trial_card.pdf`

The single-run baseline card is copied as `figs/simB_baseline_trial_card.pdf` for use in the dissertation figures. The JSON file `trial_card.json` provides the canonical baseline metrics and bid-level microdata for all downstream paired analyses.

Phase 6 — Reset Chain for Intervention

Before running the intervention condition, restart the chain and verify that the mechanism parameters match the baseline:

```
./start_swechain.sh 20
swechaind query auction params -o json \
  > simB/intervention_params.json
diff -u simB/baseline_params.json \
       simB/intervention_params.json
```

The `diff` command should report no differences. Any discrepancy indicates a configuration drift that must be fixed before proceeding.

Phase 7 — Run the Intervention Trial

```
./bin/trial -minutes 20 \
  -agent-bin ./bin/swechain_agent \
  -spec ./simB/config/agent_B_intervention.json \
  -trial ./simB/trials/intervention \
  -dbdir ./simB/trials/intervention_db \
  ./simB/data/intervention/trial_001.json
```

This uses the same dataset, agents, random seeds, and timing as the baseline run, but with the price-signal prompt block enabled in `agent_B_intervention.json`. The resulting on-chain history is the intervention counterpart of the baseline history.

Phase 8 — Intervention Economic and Network Metrics (`trial_card` Compute Mode)

While the intervention chain state is still present, compute the canonical metrics and microdata:

```
./bin/trial_card \
  --bin swechaind \
  --node tcp://localhost:26657 \
  --home "${HOME}/.swechain" \
  --outdir ./simB/artifact/intervention \
  --output text
```

This writes:

❏ `simB/artifact/intervention/trial_card.json`

❏ `simB/artifact/intervention/trial_card.svg`

❏ `simB/artifact/intervention/trial_card.pdf`

The single-run intervention card is copied as `figs/simB_intervention_trial_card.pdf`. At this point, both conditions have their own `trial_card.json` files, generated under identical chain parameters.

Phase 9 — Paired Economic Comparison (`trial_card` Compare Mode)

With both `trial_card.json` files in place, the paired baseline–intervention economic trial card is generated offline, without accessing the chain:

```
./bin/trial_card --compare \
  --baseline ./simB/artifact/baseline/trial_card.json \
  --intervention ./simB/artifact/intervention/trial_card.json \
  --outdir ./simB/artifact/compare \
  --label "Simulation B - Price-signal utilization" \
  --output text
```

This writes:

❏ `simB/artifact/compare/trial_pair.json`

❏ `simB/artifact/compare/trial_pair.svg`

❏ `simB/artifact/compare/trial_pair.pdf`

The paired economic trial card is copied as `figs/simB_trial_pair.pdf` for use in the dissertation figures.

Phase 10 — Paired Bid-Landscape Comparison (`simB_results`)

Finally, the Simulation B-specific paired bid-landscape figure is generated offline from the same canonical metrics files. The `simB_results` program reads the baseline and intervention `trial_card.json` files under `simB/artifact/` and emits an SVG that is converted to PDF via `rsvg-convert`:

```
./bin/simB_results \
  -artifact-dir ./simB/artifact \
  -out-pdf      ./simB/artifact/compare/simB_results.pdf
```

This writes:

- ❏ `simB/artifact/compare/simB_results.svg`
- ❏ `simB/artifact/compare/simB_results.pdf`

The PDF is copied as `figs/simB_results.pdf` for use in the dissertation figures. Because `simB_results` operates purely on the `trial_card.json` microdata, this phase does not depend on any particular chain height being kept alive. In the resulting figure, the horizontal axis represents bid or winning price in SWEChain tokens per task; the two bands distinguish baseline and intervention; and within each band non-winning bids are jittered vertically for visibility while winning bids lie on a fixed horizontal line.

B.7 Limits, Assumptions, and Validation

Simulation B enforces that the only policy change between conditions is the presence of price signals in the bidding prompts. All other parameters, including dataset, seeds, agent pool, prompts (apart from the price-signal block), auction rules, timing, and chain parameters, are held fixed.

Language model stochasticity means individual token sequences may differ across reruns, even with the same seeds and configuration. However, the recorded configuration and protocol are strong enough to reproduce the experimental setup and to check whether the qualitative and quantitative patterns in task completion, outsourcing cost, competition, and revenue concentration persist.

Validation consists of:

1. checking that baseline and intervention trial files share identical task order and agent lists, and that the manifests show the expected pairing;

2. verifying that `simB/baseline_params.json` and `simB/intervention_params.json` are identical, confirming that the blockchain mechanism did not change across conditions;

3. recomputing economic and network metrics via `trial_card` for both conditions and confirming that the regenerated single-run cards match the reported values up to ordinary numeric and rendering tolerances;

4. recomputing the paired economic trial card via `trial_card` in compare mode and the bid-landscape figure via `simB_results` from the stored `trial_card.json` files, and confirming that the observed patterns in winning prices, bid dispersion, competition, and revenue concentration are consistent with those in the dissertation figures;

5. for the bid-landscape figure specifically, checking that the counts in the summary box (numbers of auctions, auctions with bids, bids, and paired problems) match those implied by the underlying `trial_card.json` microdata, and that all winning-bid markers and mean-price reference lines agree with the recorded winning prices and mean cost per completed task.

Under this protocol, a reader with access to the public repository, the same model identifier, and a similar machine can rerun Simulation B, regenerate both the economic trial cards and the Simulation B bid landscape from the canonical JSON artifacts, and verify the main economic and network findings associated with enabling bidders' price-signal utilization.

APPENDIX C

Simulation Experiment C

C.1 Overview

This appendix documents all technical steps required to reproduce Simulation C, including data generation, chain initialization, agent execution, economic and network metric extraction, and Simulation C-specific price–quality analysis. All commands and artifacts correspond to the public repository.

Simulation C varies one policy dimension: how the principal closes software outsourcing auctions once bids have been placed. The baseline configuration uses a price-only rule that selects the lowest admissible bid subject to reserve and eligibility conditions. The intervention configuration replaces this with a transparent price–quality rule in which quality is proxied by the match between a bidder's specialization tags and the problem's tags. Both configurations encourage bidders to surface their specialization explicitly in their prompts so that it can be used as a quality signal. All other elements, including dataset, agent pool, random seeds, auction rules, and execution timing, are held fixed. Thus, observed differences reflect the marginal effect of principal-weighted quality-signal utilization under otherwise identical conditions.

Simulation C uses two layers of results:

1. an experiment-independent layer of economic and network metrics, extracted by the tool `trial_card` (completion, cost, inequality, and participation structure) from each run's on-chain auctions and bids; and

2. a Simulation C-specific price–quality comparison, extracted by the tool `simC_results` from the canonical `trial_card.json` files and trial databases for baseline and intervention.

As in Simulation B, the dependency is one-way. In compute mode, `trial_card` talks to the local `swechaind` node and writes canonical, simulation-agnostic `trial_card.json` files that contain aggregate metrics and microdata. The compare mode of `trial_card` and the `simC_results` tool are pure offline consumers of these JSON artifacts and the per-agent trial databases; neither accesses the chain.

C.2 Data Engineering

Simulation C uses a deterministic dataset snapshot `data/problems.json` and a single synthetic trial that is reused for both baseline and intervention. The trial is generated once with specialized agents so that both arms share the same problems, order, and specialization tags.

Synthetic data are produced with the `sdge` tool. Assuming the binary has been built as `bin/sdge` and the problem cache is available at `./data/problems.json`, the Simulation C trial is generated via:

```
./bin/sdge \
  -problems-file ./data/problems.json \
  -trials 1 -problems 200 -agents 20 \
  -seed 1111 \
  -specialist -spec-tags 2 \
  -output-dir ./simC/data
```

This emits:

- ❏ `simC/data/trial_001.json`
- ❏ `simC/data/manifest.json`

where `trial_001.json` encodes 200 problems, 20 agents, per-agent specialization tags, cost profiles, and a fixed problem order. The manifest records file hashes and basic metadata.

To ensure that baseline and intervention runs are perfectly paired, this single trial is duplicated into separate baseline and intervention directories:

```
mkdir -p ./simC/data/baseline ./simC/data/intervention

cp ./simC/data/trial_001.json ./simC/data/baseline/
cp ./simC/data/trial_001.json ./simC/data/intervention/

cp ./simC/data/manifest.json ./simC/data/baseline/
cp ./simC/data/manifest.json ./simC/data/intervention/
```

After these steps, both:

- `simC/data/baseline/trial_001.json` and
- `simC/data/intervention/trial_001.json`

refer to the same synthetic trial, with identical problems, ordering, agent pool, and specialization tags. Comparative advantage is therefore present in both arms; the only intended difference is the principal's winner-selection policy.

C.3 Host Execution Environment

Simulation C was executed on the same local workstation used for Simulation B:

- Debian GNU/Linux 12 (x86_64)
- AMD Ryzen 7 5800H, 27 GiB RAM
- Go 1.24.7, Git 2.39.5
- Ollama 0.12.9 running model `gpt-oss:120b-cloud`

Language model sampling configuration (temperature, top-p, maximum tokens) is pinned in the agent specification files and used for both conditions. Any reproduction should use the same model identifier and similar hardware, or at least document deviations.

C.4 Blockchain Configuration

Simulation C uses a dedicated SWEChain instance with fixed:

- genesis file and chain identifier;
- account balances and initial token distribution;
- software outsourcing auction parameters (duration, grace periods, tick size);
- consensus and gas settings.

As in Simulation B, the experiment requires that baseline and intervention runs use identical chain parameters, apart from the winner-selection rule encoded in the auction module. In the baseline configuration, winners are chosen by lowest admissible bid (price-only). In the intervention configuration, winners are chosen by a deterministic price–quality scoring rule in which quality is proxied by specialization tags surfaced by bidders.

Consistency of all other parameters is enforced by exporting and comparing parameter snapshots before each run. Any difference detected in the exported JSON files must be resolved before using the results in the paired comparison.

C.5 Agent Configuration

Two agent specifications are used:

- `simC/config/agent_C_baseline.json`
- `simC/config/agent_C_intervention.json`

Both files configure the same finite-state machine, retry policy, memory filters, and language model settings. Agents discover their wallet, open auctions for their own assignments, place bids on other agents' auctions, accept and announce winners, code solutions, and periodically update a simple profit-and-loss summary.

In both baseline and intervention:

- agents read their specialization tags from memory (as injected by the synthetic trial);
- bidders are explicitly instructed to prioritize auctions whose problem tags overlap with their specialization; and
- bidders are encouraged to surface a compact summary of their specialization as part of their bidding behavior, so that it can act as a quality signal.

The only conceptual difference between the two specifications lies in the principal's accepting logic. In the baseline configuration, the accepting prompts describe and reinforce price-only selection: the principal closes auctions by choosing the lowest admissible bid, subject to reserve and eligibility checks. In the intervention configuration, the accepting prompts describe and reinforce the chain's price–quality scoring rule, which weighs both price and specialization-based quality when closing auctions.

These finite-state machines make the agent behavior explicit and ensure that any differences in outcomes can be traced to the change in winner-selection policy rather than unrelated control-flow changes.

C.6 Reproducibility Protocol

This section enumerates the steps needed to regenerate all Simulation C artifacts, including runs, economic and network metrics, the paired economic trial card, and the Simulation C-specific price–quality comparison. All paths are relative to the repository root.

Phase 1 — Build Binaries

```
go build -o bin/sdge          ./src/sdge.go
go build -o bin/trial         ./src/trial.go
go build -o bin/swechain_agent ./src/swechain_agent.go
go build -o bin/trial_card    ./src/trial_card.go
go build -o bin/simC_results  ./simC/src/simC_results.go
```

This produces the deterministic data generator, trial orchestrator, agent binary, canonical metrics tool, and the Simulation C analysis tool.

Phase 2 — Generate the Synthetic Trial

```
./bin/sdge \
  -problems-file ./data/problems.json \
  -trials 1 -problems 200 -agents 20 \
  -seed 1111 \
  -specialist -spec-tags 2 \
  -output-dir ./simC/data
```

This writes `simC/data/trial_001.json` and `simC/data/manifest.json`. The trial includes 200 problems, 20 agents, specialization tags, and a fixed problem order.

Phase 3 — Prepare Baseline and Intervention Data

```
mkdir -p ./simC/data/baseline ./simC/data/intervention

cp ./simC/data/trial_001.json ./simC/data/baseline/
cp ./simC/data/trial_001.json ./simC/data/intervention/

cp ./simC/data/manifest.json ./simC/data/baseline/
cp ./simC/data/manifest.json ./simC/data/intervention/
```

This ensures that baseline and intervention use the same trial file and manifest, making the pairing explicit.

Phase 4 — Initialize SWEChain for Baseline

```
./start_swechain.sh 20
swechaind query auction params -o json \
  > simC/baseline_params.json
```

The `start_swechain.sh` script starts a local SWEChain node with a fixed configuration. The exported parameters are saved for later comparison.

Phase 5 — Run the Baseline Trial

```
./bin/trial -minutes 20 \
  -agent-bin ./bin/swechain_agent \
  -spec ./simC/config/agent_C_baseline.json \
  -trial ./simC/trials/baseline \
  -dbdir ./simC/trials/baseline_db \
  ./simC/data/baseline/trial_001.json
```

This runs the baseline configuration for 20 minutes of wall-clock time, producing per-agent databases and journals under `simC/trials/baseline_db/` and all on-chain auction and bid events for the baseline condition.

Phase 6 — Baseline Economic and Network Metrics (`trial_card` Compute Mode)

While the baseline chain state is still present, compute the canonical economic and network metrics and microdata:

```
./bin/trial_card \
  --bin swechaind \
  --node tcp://localhost:26657 \
  --home "${HOME}/.swechain" \
  --outdir ./simC/artifact/baseline \
  --output text
```

This writes:

- ❏ `simC/artifact/baseline/trial_card.json`
- ❏ `simC/artifact/baseline/trial_card.svg`
- ❏ `simC/artifact/baseline/trial_card.pdf`

The single-run baseline card is copied as `figs/simC_baseline_trial_card.pdf` for use in the dissertation figures.

Phase 7 — Reset Chain for Intervention

Before running the intervention condition, restart the chain and verify that the mechanism parameters match the baseline:

```
./start_swechain.sh 20
swechaind query auction params -o json \
  > simC/intervention_params.json
diff -u simC/baseline_params.json \
       simC/intervention_params.json
```

The `diff` command should report no differences. Any discrepancy indicates configuration drift that must be corrected before continuing.

Phase 8 — Run the Intervention Trial

```
./bin/trial -minutes 20 \
  -agent-bin ./bin/swechain_agent \
  -spec ./simC/config/agent_C_intervention.json \
  -trial ./simC/trials/intervention \
  -dbdir ./simC/trials/intervention_db \
  ./simC/data/intervention/trial_001.json
```

This uses the same dataset, agents, random seeds, and timing as the baseline run, but with the price–quality winner-selection rule enabled in the chain configuration and reinforced in `agent_C_intervention.json`. The resulting on-chain history is the intervention counterpart of the baseline history.

Phase 9 — Intervention Economic and Network Metrics (`trial_card` Compute Mode)

While the intervention chain state is still present, compute the canonical metrics and microdata:

```
./bin/trial_card \
  --bin swechaind \
  --node tcp://localhost:26657 \
  --home "${HOME}/.swechain" \
  --outdir ./simC/artifact/intervention \
  --output text
```

This writes:

- ❏ `simC/artifact/intervention/trial_card.json`
- ❏ `simC/artifact/intervention/trial_card.svg`
- ❏ `simC/artifact/intervention/trial_card.pdf`

The single-run intervention card is copied as `figs/simC_intervention_trial_card.pdf` for use in the dissertation figures.

Phase 10 — Paired Economic Comparison (`trial_card` Compare Mode)

With both `trial_card.json` files in place, the paired baseline–intervention economic trial card is generated offline, without accessing the chain:

```
./bin/trial_card --compare \
  --baseline ./simC/artifact/baseline/trial_card.json \
  --intervention ./simC/artifact/intervention/trial_card.json \
  --outdir ./simC/artifact/compare \
  --label "Simulation C - Price-quality selection" \
  --output text
```

This writes:

- ❏ `simC/artifact/compare/trial_pair.json`
- ❏ `simC/artifact/compare/trial_pair.svg`
- ❏ `simC/artifact/compare/trial_pair.pdf`

The paired economic trial card is copied as `figs/simC_trial_pair.pdf` for use in the dissertation figures.

Phase 11 — Price–Quality Comparison (`simC_results`)

Finally, the Simulation C-specific analysis is generated offline from the same canonical metrics files and the per-agent trial databases. The `simC_results` program reads the baseline and intervention artifacts under `simC/artifact/` and emits an SVG that is converted to PDF:

```
./bin/simC_results \
  -baseline-dbdir     ./simC/trials/baseline_db \
  -intervention-dbdir ./simC/trials/intervention_db \
  -artifact-dir       ./simC/artifact \
  -out-pdf            ./simC/artifact/compare/simC_results.pdf
```

This writes:

- ❏ `simC/artifact/compare/simC_results.svg`
- ❏ `simC/artifact/compare/simC_results.pdf`

The PDF is copied as `figs/simC_results.pdf` for use in the dissertation figures. Because `simC_results` operates purely on the stored JSON artifacts and trial databases, this phase does not depend on any particular chain height being kept alive.

C.7 Limits, Assumptions, and Validation

Simulation C enforces that the only policy change between conditions is the winner-selection rule applied by the principal. Both arms use the same synthetic trial (problems, order, agents, specialization tags), and both encourage bidders to expose their specialization as a quality signal. All other parameters, including auction rules, timing, chain configuration, and agent finite-state machines, are held fixed.

Language model stochasticity implies that individual token sequences may differ across reruns, even with the same configuration. However, the recorded configuration and protocol are strong enough to reproduce the experimental setup and to check whether the qualitative and quantitative patterns in task completion, outsourcing cost, price dispersion, and revenue concentration persist.

Validation consists of:

1. checking that `simC/data/baseline/trial_001.json` and `simC/data/intervention/trial_001.js` are identical (or share the same hash recorded in the manifest), confirming that both arms use the same synthetic trial;

2. verifying that `simC/baseline_params.json` and `simC/intervention_params.json` are identical, confirming that the blockchain mechanism did not change across conditions apart from the intended winner-selection policy;

3. recomputing economic and network metrics via `trial_card` for both conditions and confirming that the regenerated single-run cards match the reported values up to ordinary numeric and rendering tolerances;

4. recomputing the paired economic trial card via `trial_card` in compare mode and the Simulation C figure via `simC_results` from the stored artifacts, and confirming that the observed patterns in allocations, prices, and revenue concentration are consistent with those in the dissertation figures; and

5. checking that counts and summary statistics reported in the Simulation C figure (for example, numbers of auctions, completed tasks, and the distribution of revenue shares) match those implied by the underlying `trial_card.json` microdata and trial databases.

Under this protocol, a reader with access to the public repository, the same model identifier, and a similar machine can rerun Simulation C, regenerate both the economic trial cards and the Simulation C price–quality analysis from the canonical artifacts, and verify the main findings associated with principal-weighted quality-signal utilization.

APPENDIX D

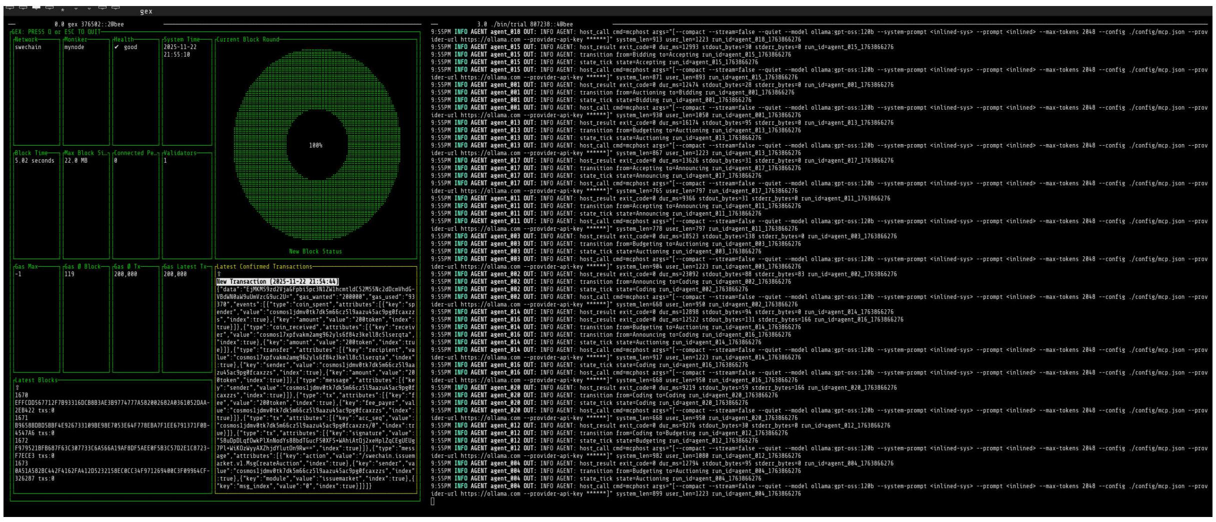
Tooling and Inspection

This appendix documents three practical inspection tools used during the development and evaluation of *SWEChain-SDK*: a terminal block explorer for the SWEChain appchain, a live dashboard for monitoring paired trials, and a TUI browser over the native agent runtime's bbolt-backed state files. These tools are conveniences rather than requirements for reproducibility: they operate on the same on-chain events and agent-level logs that drive the metrics and figures in the simulation chapters.

D.1 Terminal Block Explorer for SWEChain

During runs, the SWEChain node exposes its RPC endpoint on the local host. To inspect blocks, transactions, and events without leaving the terminal, we use a command-line block explorer that attaches directly to this RPC interface. For a given trial, the explorer is pointed at the same node directory and RPC address as the simulation code.

Figure 24 shows an example of this explorer attached to a SWEChain instance during a trial. The screenshot illustrates the main views used in practice: a list of recent blocks with heights and hashes, details for the selected block (including transactions and their type tags), and decoded events with their key–value attributes. This view makes it straightforward to confirm that auctions are being created with the expected parameters, that bids are being submitted at the expected heights, and that settlement transfers occur when auctions close.



```
gex
──            0.0 gex 376502::2@bee
GEX: PRESS Q or ESC TO QUIT
Network          Moniker          Health           System Time      Current Block Round
swechain         mynode           ✔ good           2025-11-22
                                                   21:55:10
                                                                         100%
Block Time       Max Block Si…    Connected Pe…    Validators
5.02 seconds     22.0 MB          0                1
                                                                    New Block Status
Gas Max          Gas @ Block      Gas @ Tx         Gas Latest Tx    Latest Confirmed Transactions
-1               119              200,000          200,000          ⇧
                                                                    New Transaction (2025-11-22 21:54:44)
                                                                    {"data":"EjMKMS9zd2VjaGFpbi5pc3N1ZW1hcmtldC52MS5Nc2dDcmVhdG-
                                                                    VBdWN0aW9uUmVzcG9uc2U=","gas_wanted":"200000","gas_used":"93
                                                                    370","events":[{"type":"coin_spent","attributes":[{"key":"sp
                                                                    ender","value":"cosmos1jdmv0tk7dk5m66cz5l9aazu45ac9pg0fcaxzz
                                                                    s","index":true},{"key":"amount","value":"200token","index":
                                                                    true}]},{"type":"coin_received","attributes":[{"key":"receiv
                                                                    er","value":"cosmos17xpfvakm2amg962yls6f84z3kell8c5lserqta",
                                                                    "index":true},{"key":"amount","value":"200token","index":tru
                                                                    e}]},{"type":"transfer","attributes":[{"key":"recipient","va
                                                                    lue":"cosmos17xpfvakm2amg962yls6f84z3kell8c5lserqta","index"
                                                                    :true},{"key":"sender","value":"cosmos1jdmv0tk7dk5m66cz5l9aa
                                                                    zu45ac9pg0fcaxzzs","index":true},{"key":"amount","value":"20
Latest Blocks                                                       0token","index":true}]},{"type":"message","attributes":[{"ke
⇧                                                                   y":"sender","value":"cosmos1jdmv0tk7dk5m66cz5l9aazu45ac9pg0f
1670                                                                caxzzs","index":true}]},{"type":"tx","attributes":[{"key":"f
EFFCDD567712F7B93316DCB8B3AE3B9774777A582002682A0361052DAA-         ee","value":"200token","index":true},{"key":"fee_payer","val
2EB422 txs:0                                                        ue":"cosmos1jdmv0tk7dk5m66cz5l9aazu45ac9pg0fcaxzzs","index":
1671                                                                true}]},{"type":"tx","attributes":[{"key":"acc_seq","value":
B9658BDBD5BBF4E926733109BE98E7053E64F778EBA7F1EE6791371F0B-         "cosmos1jdmv0tk7dk5m66cz5l9aazu45ac9pg0fcaxzzs/0","index":tr
4547A6 txs:0                                                        ue}]},{"type":"tx","attributes":[{"key":"signature","value":
1672                                                                "58uDpDLqfOwkPlXnNodYs88bdTGucFS0XFS+WAhiAtQj2xeHplZqCEgUEUg
F979521BFB687F63C307733C6A566A19AF8DF5AEE0F5B3C57D2E1C8723-         7Pl+WiKOzWyyAXZhjdYlutOn9Rw==","index":true}]},{"type":"mess
F7ECE3 txs:0                                                        age","attributes":[{"key":"action","value":"/swechain.issuem
1673                                                                arket.v1.MsgCreateAuction","index":true},{"key":"sender","va
0A51A582BC442F4162FA412D5232158EC0CC34F971269400C3F09964CF-         lue":"cosmos1jdmv0tk7dk5m66cz5l9aazu45ac9pg0fcaxzzs","index"
326287 txs:0                                                        :true},{"key":"module","value":"issuemarket","index":true},{
                                                                    "key":"msg_index","value":"0","index":true}]}]}
```

```
──            3.0 ./bin/trial 807238::4@bee
9:55PM INFO AGENT agent_018 OUT: INFO AGENT: host_call cmd=mcphost args="[--compact --stream=false --quiet --model ollama:gpt-oss:120b --system-prompt <inlined-sys> --prompt <inlined> --max-tokens 2048 --config ./config/mcp.json --prov
ider-url https://ollama.com --provider-api-key ******]" system_len=913 user_len=1223 run_id=agent_018_1763866276
9:55PM INFO AGENT agent_015 OUT: INFO AGENT: host_result exit_code=0 dur_ms=12993 stdout_bytes=30 stderr_bytes=0 run_id=agent_015_1763866276
9:55PM INFO AGENT agent_015 OUT: INFO AGENT: transition from=Bidding to=Accepting run_id=agent_015_1763866276
9:55PM INFO AGENT agent_015 OUT: INFO AGENT: state_tick state=Accepting run_id=agent_015_1763866276
9:55PM INFO AGENT agent_015 OUT: INFO AGENT: host_call cmd=mcphost args="[--compact --stream=false --quiet --model ollama:gpt-oss:120b --system-prompt <inlined-sys> --prompt <inlined> --max-tokens 2048 --config ./config/mcp.json --prov
ider-url https://ollama.com --provider-api-key ******]" system_len=871 user_len=893 run_id=agent_015_1763866276
9:55PM INFO AGENT agent_001 OUT: INFO AGENT: host_result exit_code=0 dur_ms=12474 stdout_bytes=28 stderr_bytes=0 run_id=agent_001_1763866276
9:55PM INFO AGENT agent_001 OUT: INFO AGENT: transition from=Auctioning to=Bidding run_id=agent_001_1763866276
9:55PM INFO AGENT agent_001 OUT: INFO AGENT: state_tick state=Bidding run_id=agent_001_1763866276
9:55PM INFO AGENT agent_001 OUT: INFO AGENT: host_call cmd=mcphost args="[--compact --stream=false --quiet --model ollama:gpt-oss:120b --system-prompt <inlined-sys> --prompt <inlined> --max-tokens 2048 --config ./config/mcp.json --prov
ider-url https://ollama.com --provider-api-key ******]" system_len=930 user_len=1050 run_id=agent_001_1763866276
9:55PM INFO AGENT agent_013 OUT: INFO AGENT: host_result exit_code=0 dur_ms=16174 stdout_bytes=95 stderr_bytes=0 run_id=agent_013_1763866276
9:55PM INFO AGENT agent_013 OUT: INFO AGENT: transition from=Budgeting to=Auctioning run_id=agent_013_1763866276
9:55PM INFO AGENT agent_013 OUT: INFO AGENT: state_tick state=Auctioning run_id=agent_013_1763866276
9:55PM INFO AGENT agent_013 OUT: INFO AGENT: host_call cmd=mcphost args="[--compact --stream=false --quiet --model ollama:gpt-oss:120b --system-prompt <inlined-sys> --prompt <inlined> --max-tokens 2048 --config ./config/mcp.json --prov
ider-url https://ollama.com --provider-api-key ******]" system_len=867 user_len=1223 run_id=agent_013_1763866276
9:55PM INFO AGENT agent_017 OUT: INFO AGENT: host_result exit_code=0 dur_ms=13626 stdout_bytes=31 stderr_bytes=0 run_id=agent_017_1763866276
9:55PM INFO AGENT agent_017 OUT: INFO AGENT: transition from=Accepting to=Announcing run_id=agent_017_1763866276
9:55PM INFO AGENT agent_017 OUT: INFO AGENT: state_tick state=Announcing run_id=agent_017_1763866276
9:55PM INFO AGENT agent_017 OUT: INFO AGENT: host_call cmd=mcphost args="[--compact --stream=false --quiet --model ollama:gpt-oss:120b --system-prompt <inlined-sys> --prompt <inlined> --max-tokens 2048 --config ./config/mcp.json --prov
ider-url https://ollama.com --provider-api-key ******]" system_len=765 user_len=797 run_id=agent_017_1763866276
9:55PM INFO AGENT agent_011 OUT: INFO AGENT: host_result exit_code=0 dur_ms=9366 stdout_bytes=31 stderr_bytes=0 run_id=agent_011_1763866276
9:55PM INFO AGENT agent_011 OUT: INFO AGENT: transition from=Accepting to=Announcing run_id=agent_011_1763866276
9:55PM INFO AGENT agent_011 OUT: INFO AGENT: state_tick state=Announcing run_id=agent_011_1763866276
9:55PM INFO AGENT agent_011 OUT: INFO AGENT: host_call cmd=mcphost args="[--compact --stream=false --quiet --model ollama:gpt-oss:120b --system-prompt <inlined-sys> --prompt <inlined> --max-tokens 2048 --config ./config/mcp.json --prov
ider-url https://ollama.com --provider-api-key ******]" system_len=778 user_len=797 run_id=agent_011_1763866276
9:55PM INFO AGENT agent_003 OUT: INFO AGENT: host_result exit_code=0 dur_ms=18523 stdout_bytes=138 stderr_bytes=0 run_id=agent_003_1763866276
9:55PM INFO AGENT agent_003 OUT: INFO AGENT: transition from=Budgeting to=Auctioning run_id=agent_003_1763866276
9:55PM INFO AGENT agent_003 OUT: INFO AGENT: state_tick state=Auctioning run_id=agent_003_1763866276
9:55PM INFO AGENT agent_003 OUT: INFO AGENT: host_call cmd=mcphost args="[--compact --stream=false --quiet --model ollama:gpt-oss:120b --system-prompt <inlined-sys> --prompt <inlined> --max-tokens 2048 --config ./config/mcp.json --prov
ider-url https://ollama.com --provider-api-key ******]" system_len=904 user_len=1223 run_id=agent_003_1763866276
9:55PM INFO AGENT agent_002 OUT: INFO AGENT: host_result exit_code=0 dur_ms=23092 stdout_bytes=88 stderr_bytes=83 run_id=agent_002_1763866276
9:55PM INFO AGENT agent_002 OUT: INFO AGENT: transition from=Announcing to=Coding run_id=agent_002_1763866276
9:55PM INFO AGENT agent_002 OUT: INFO AGENT: state_tick state=Coding run_id=agent_002_1763866276
9:55PM INFO AGENT agent_002 OUT: INFO AGENT: host_call cmd=mcphost args="[--compact --stream=false --quiet --model ollama:gpt-oss:120b --system-prompt <inlined-sys> --prompt <inlined> --max-tokens 2048 --config ./config/mcp.json --prov
ider-url https://ollama.com --provider-api-key ******]" system_len=668 user_len=950 run_id=agent_002_1763866276
9:55PM INFO AGENT agent_014 OUT: INFO AGENT: host_result exit_code=0 dur_ms=12898 stdout_bytes=94 stderr_bytes=0 run_id=agent_014_1763866276
9:55PM INFO AGENT agent_016 OUT: INFO AGENT: host_result exit_code=0 dur_ms=12522 stdout_bytes=131 stderr_bytes=166 run_id=agent_016_1763866276
9:55PM INFO AGENT agent_014 OUT: INFO AGENT: transition from=Budgeting to=Auctioning run_id=agent_014_1763866276
9:55PM INFO AGENT agent_016 OUT: INFO AGENT: transition from=Announcing to=Coding run_id=agent_016_1763866276
9:55PM INFO AGENT agent_014 OUT: INFO AGENT: state_tick state=Auctioning run_id=agent_014_1763866276
9:55PM INFO AGENT agent_014 OUT: INFO AGENT: host_call cmd=mcphost args="[--compact --stream=false --quiet --model ollama:gpt-oss:120b --system-prompt <inlined-sys> --prompt <inlined> --max-tokens 2048 --config ./config/mcp.json --prov
ider-url https://ollama.com --provider-api-key ******]" system_len=917 user_len=1223 run_id=agent_014_1763866276
9:55PM INFO AGENT agent_016 OUT: INFO AGENT: state_tick state=Coding run_id=agent_016_1763866276
9:55PM INFO AGENT agent_016 OUT: INFO AGENT: host_call cmd=mcphost args="[--compact --stream=false --quiet --model ollama:gpt-oss:120b --system-prompt <inlined-sys> --prompt <inlined> --max-tokens 2048 --config ./config/mcp.json --prov
ider-url https://ollama.com --provider-api-key ******]" system_len=668 user_len=950 run_id=agent_016_1763866276
9:55PM INFO AGENT agent_020 OUT: INFO AGENT: host_result exit_code=0 dur_ms=9219 stdout_bytes=59 stderr_bytes=166 run_id=agent_020_1763866276
9:55PM INFO AGENT agent_020 OUT: INFO AGENT: transition from=Coding to=Coding run_id=agent_020_1763866276
9:55PM INFO AGENT agent_020 OUT: INFO AGENT: state_tick state=Coding run_id=agent_020_1763866276
9:55PM INFO AGENT agent_020 OUT: INFO AGENT: host_call cmd=mcphost args="[--compact --stream=false --quiet --model ollama:gpt-oss:120b --system-prompt <inlined-sys> --prompt <inlined> --max-tokens 2048 --config ./config/mcp.json --prov
ider-url https://ollama.com --provider-api-key ******]" system_len=668 user_len=950 run_id=agent_020_1763866276
9:55PM INFO AGENT agent_012 OUT: INFO AGENT: host_result exit_code=0 dur_ms=9276 stdout_bytes=30 stderr_bytes=0 run_id=agent_012_1763866276
9:55PM INFO AGENT agent_012 OUT: INFO AGENT: transition from=Coding to=Budgeting run_id=agent_012_1763866276
9:55PM INFO AGENT agent_012 OUT: INFO AGENT: state_tick state=Budgeting run_id=agent_012_1763866276
9:55PM INFO AGENT agent_012 OUT: INFO AGENT: host_call cmd=mcphost args="[--compact --stream=false --quiet --model ollama:gpt-oss:120b --system-prompt <inlined-sys> --prompt <inlined> --max-tokens 2048 --config ./config/mcp.json --prov
ider-url https://ollama.com --provider-api-key ******]" system_len=982 user_len=1080 run_id=agent_012_1763866276
9:55PM INFO AGENT agent_004 OUT: INFO AGENT: host_result exit_code=0 dur_ms=12794 stdout_bytes=95 stderr_bytes=0 run_id=agent_004_1763866276
9:55PM INFO AGENT agent_004 OUT: INFO AGENT: transition from=Budgeting to=Auctioning run_id=agent_004_1763866276
9:55PM INFO AGENT agent_004 OUT: INFO AGENT: state_tick state=Auctioning run_id=agent_004_1763866276
9:55PM INFO AGENT agent_004 OUT: INFO AGENT: host_call cmd=mcphost args="[--compact --stream=false --quiet --model ollama:gpt-oss:120b --system-prompt <inlined-sys> --prompt <inlined> --max-tokens 2048 --config ./config/mcp.json --prov
ider-url https://ollama.com --provider-api-key ******]" system_len=899 user_len=1223 run_id=agent_004_1763866276
▯
```

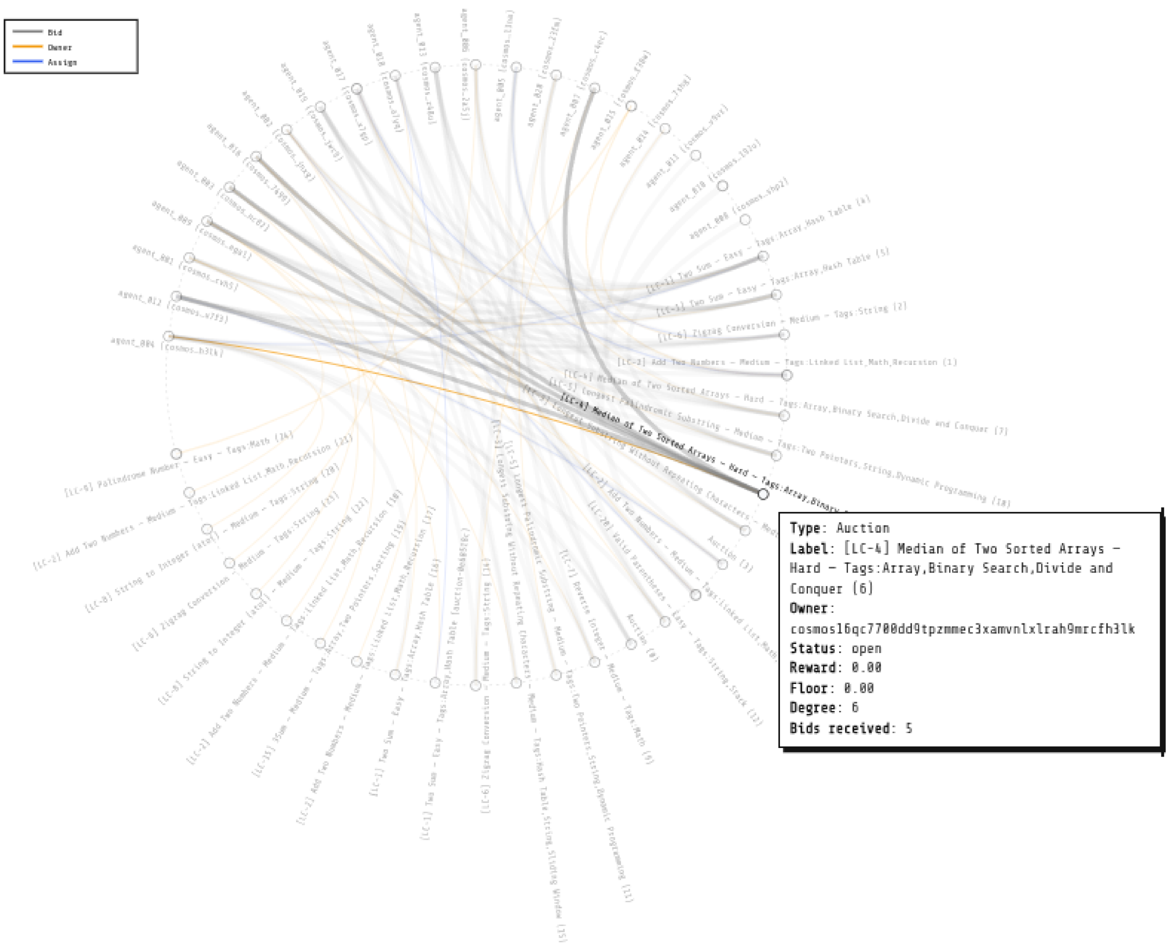
Figure 24 – Terminal block explorer attached to a SWEChain node for a simulation run. The explorer renders blocks, transactions, and their events directly from the node's RPC endpoint, making it easy to inspect auction creation, bidding, and settlement at the block level.

The explorer is not used to generate any of the quantitative results in this dissertation. Instead, it serves as a sanity-check and inspection tool during development and debugging, complementing the automated extraction of event streams into `chain_events.jsonl` and related artifacts described in the main text.

D.2 Live Dashboard for Paired Trials

The live dashboard is a visualization layer over the same structured data that supports the offline analytics. It consumes exported chain events and agent journals, aggregates them into high-level indicators, and presents them as panels and time series.

Figure 25 shows a screenshot of the dashboard during a paired trial. At a glance, the dashboard surfaces the current chain height, the number of open and closed auctions, bid counts, task completion over time, and per-agent balances or revenues. This helps the researcher monitor progress, detect stalled agents or unusual bidding patterns, and verify that the baseline and intervention runs are proceeding under the intended workloads.



```
Type: Auction
Label: [LC-4] Median of Two Sorted Arrays -
Hard - Tags:Array,Binary Search,Divide and
Conquer (6)
Owner:
cosmos16qc7700dd9tpzmmec3xamvnlxlrah9mrcfh3lk
Status: open
Reward: 0.00
Floor: 0.00
Degree: 6
Bids received: 5
```

Figure 25 – Live dashboard for a paired trial. The dashboard presents near-real-time views of chain height, auctions, bids, and agent status using the same event streams and journals that later feed the offline analytics.

Importantly, the dashboard is strictly read-only with respect to the chain and agent runtimes. It does not send transactions or control signals; it only reads from the event and metrics streams written to the artifact store. As such, it does not change the semantics of a run and does not affect any of the experimental results, but it provides a useful "glass box" view over the architecture described in Chapter 3.

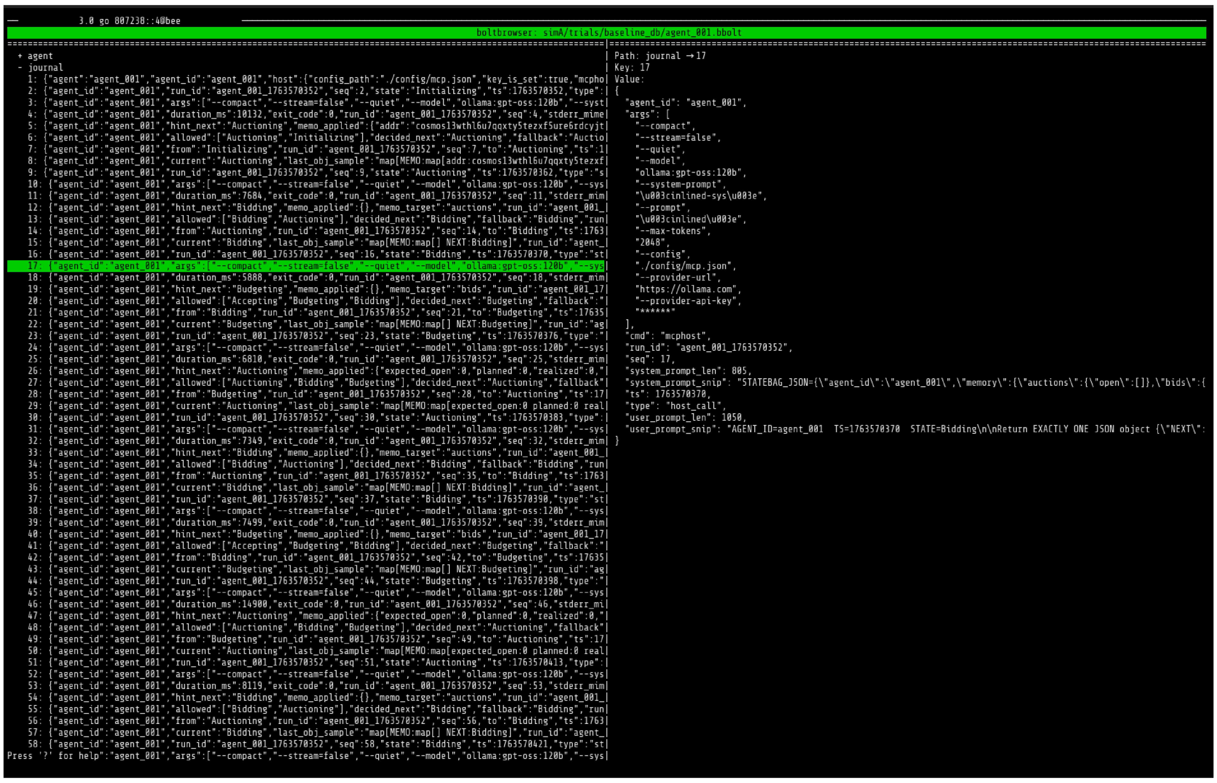
D.3 Browsing Agent State with `boltbrowser`

The native agent runtime is built on `mcphost` and uses bbolt as an embedded key–value store for per-agent state. For each agent, the runtime maintains a bbolt database file that stores buckets such as `meta`, `memory`, and `journal`, alongside any additional structures used by the controller. These files live under trial-specific directories, for example:

```
simA/trials/baseline_db/agent_001.bbolt
```

`simA/trials/intervention_db/agent_001.bbolt`

To inspect the contents of these databases without writing custom code, we use a terminal UI browser for bbolt files. Figure 26 shows a screenshot of this browser opened on an agent's database file for a baseline run. The left pane lists buckets, and the right pane shows key–value pairs within the selected bucket, with values decoded as UTF-8 where appropriate.



```
—          3.0 go 807238::4@bee          ————————————————————————————————————————
boltbrowser: simA/trials/baseline_db/agent_001.bbolt
========================================================================================
 + agent
 - journal
   1: {"agent":"agent_001","agent_id":"agent_001","host":{"config_path":"./config/mcp.json","key_is_set":true,"mcpho|
   2: {"agent_id":"agent_001","run_id":"agent_001_1763570352","seq":2,"state":"Initializing","ts":1763570352,"type":|
   3: {"agent_id":"agent_001","args":["--compact","--stream=false","--quiet","--model","ollama:gpt-oss:120b","--syst|
   4: {"agent_id":"agent_001","duration_ms":10132,"exit_code":0,"run_id":"agent_001_1763570352","seq":4,"stderr_mime|
   5: {"agent_id":"agent_001","hint_next":"Auctioning","memo_applied":{"addr":"cosmos13wthl6u7qqxty5tezxf5ure6rdcyjt|
   6: {"agent_id":"agent_001","allowed":["Auctioning","Initializing"],"decided_next":"Auctioning","fallback":"Auctio|
   7: {"agent_id":"agent_001","from":"Initializing","run_id":"agent_001_1763570352","seq":7,"to":"Auctioning","ts":1|
   8: {"agent_id":"agent_001","current":"Auctioning","last_obj_sample":"map[MEMO:map[addr:cosmos13wthl6u7qqxty5tezxf|
   9: {"agent_id":"agent_001","run_id":"agent_001_1763570352","seq":9,"state":"Auctioning","ts":1763570362,"type":"s|
   10: {"agent_id":"agent_001","args":["--compact","--stream=false","--quiet","--model","ollama:gpt-oss:120b","--sys|
   11: {"agent_id":"agent_001","duration_ms":7684,"exit_code":0,"run_id":"agent_001_1763570352","seq":11,"stderr_mim|
   12: {"agent_id":"agent_001","hint_next":"Bidding","memo_applied":{},"memo_target":"auctions","run_id":"agent_001_|
   13: {"agent_id":"agent_001","allowed":["Bidding","Auctioning"],"decided_next":"Bidding","fallback":"Bidding","run|
   14: {"agent_id":"agent_001","from":"Auctioning","run_id":"agent_001_1763570352","seq":14,"to":"Bidding","ts":1763|
   15: {"agent_id":"agent_001","current":"Bidding","last_obj_sample":"map[MEMO:map[] NEXT:Bidding]","run_id":"agent_|
   16: {"agent_id":"agent_001","run_id":"agent_001_1763570352","seq":16,"state":"Bidding","ts":1763570370,"type":"st|
   17: {"agent_id":"agent_001","args":["--compact","--stream=false","--quiet","--model","ollama:gpt-oss:120b","--sys|
   18: {"agent_id":"agent_001","duration_ms":5888,"exit_code":0,"run_id":"agent_001_1763570352","seq":18,"stderr_mim|
   19: {"agent_id":"agent_001","hint_next":"Budgeting","memo_applied":{},"memo_target":"bids","run_id":"agent_001_17|
   20: {"agent_id":"agent_001","allowed":["Accepting","Budgeting","Bidding"],"decided_next":"Budgeting","fallback":"|
   21: {"agent_id":"agent_001","from":"Bidding","run_id":"agent_001_1763570352","seq":21,"to":"Budgeting","ts":17635|
   22: {"agent_id":"agent_001","current":"Budgeting","last_obj_sample":"map[MEMO:map[] NEXT:Budgeting]","run_id":"ag|
   23: {"agent_id":"agent_001","run_id":"agent_001_1763570352","seq":23,"state":"Budgeting","ts":1763570376,"type":"|
   24: {"agent_id":"agent_001","args":["--compact","--stream=false","--quiet","--model","ollama:gpt-oss:120b","--sys|
   25: {"agent_id":"agent_001","duration_ms":6810,"exit_code":0,"run_id":"agent_001_1763570352","seq":25,"stderr_mim|
   26: {"agent_id":"agent_001","hint_next":"Auctioning","memo_applied":{"expected_open":0,"planned":0,"realized":0,"|
   27: {"agent_id":"agent_001","allowed":["Auctioning","Bidding","Budgeting"],"decided_next":"Auctioning","fallback"|
   28: {"agent_id":"agent_001","from":"Budgeting","run_id":"agent_001_1763570352","seq":28,"to":"Auctioning","ts":17|
   29: {"agent_id":"agent_001","current":"Auctioning","last_obj_sample":"map[MEMO:map[expected_open:0 planned:0 real|
   30: {"agent_id":"agent_001","run_id":"agent_001_1763570352","seq":30,"state":"Auctioning","ts":1763570383,"type":|
   31: {"agent_id":"agent_001","args":["--compact","--stream=false","--quiet","--model","ollama:gpt-oss:120b","--sys|
   32: {"agent_id":"agent_001","duration_ms":7349,"exit_code":0,"run_id":"agent_001_1763570352","seq":32,"stderr_mim|
   33: {"agent_id":"agent_001","hint_next":"Bidding","memo_applied":{},"memo_target":"auctions","run_id":"agent_001_|
   34: {"agent_id":"agent_001","allowed":["Bidding","Auctioning"],"decided_next":"Bidding","fallback":"Bidding","run|
   35: {"agent_id":"agent_001","from":"Auctioning","run_id":"agent_001_1763570352","seq":35,"to":"Bidding","ts":1763|
   36: {"agent_id":"agent_001","current":"Bidding","last_obj_sample":"map[MEMO:map[] NEXT:Bidding]","run_id":"agent_|
   37: {"agent_id":"agent_001","run_id":"agent_001_1763570352","seq":37,"state":"Bidding","ts":1763570390,"type":"st|
   38: {"agent_id":"agent_001","args":["--compact","--stream=false","--quiet","--model","ollama:gpt-oss:120b","--sys|
   39: {"agent_id":"agent_001","duration_ms":7499,"exit_code":0,"run_id":"agent_001_1763570352","seq":39,"stderr_mim|
   40: {"agent_id":"agent_001","hint_next":"Budgeting","memo_applied":{},"memo_target":"bids","run_id":"agent_001_17|
   41: {"agent_id":"agent_001","allowed":["Accepting","Budgeting","Bidding"],"decided_next":"Budgeting","fallback":"|
   42: {"agent_id":"agent_001","from":"Bidding","run_id":"agent_001_1763570352","seq":42,"to":"Budgeting","ts":17635|
   43: {"agent_id":"agent_001","current":"Budgeting","last_obj_sample":"map[MEMO:map[] NEXT:Budgeting]","run_id":"ag|
   44: {"agent_id":"agent_001","run_id":"agent_001_1763570352","seq":44,"state":"Budgeting","ts":1763570398,"type":"|
   45: {"agent_id":"agent_001","args":["--compact","--stream=false","--quiet","--model","ollama:gpt-oss:120b","--sys|
   46: {"agent_id":"agent_001","duration_ms":14900,"exit_code":0,"run_id":"agent_001_1763570352","seq":46,"stderr_mi|
   47: {"agent_id":"agent_001","hint_next":"Auctioning","memo_applied":{"expected_open":0,"planned":0,"realized":0,"|
   48: {"agent_id":"agent_001","allowed":["Auctioning","Bidding","Budgeting"],"decided_next":"Auctioning","fallback"|
   49: {"agent_id":"agent_001","from":"Budgeting","run_id":"agent_001_1763570352","seq":49,"to":"Auctioning","ts":17|
   50: {"agent_id":"agent_001","current":"Auctioning","last_obj_sample":"map[MEMO:map[expected_open:0 planned:0 real|
   51: {"agent_id":"agent_001","run_id":"agent_001_1763570352","seq":51,"state":"Auctioning","ts":1763570413,"type":|
   52: {"agent_id":"agent_001","args":["--compact","--stream=false","--quiet","--model","ollama:gpt-oss:120b","--sys|
   53: {"agent_id":"agent_001","duration_ms":8119,"exit_code":0,"run_id":"agent_001_1763570352","seq":53,"stderr_mim|
   54: {"agent_id":"agent_001","hint_next":"Bidding","memo_applied":{},"memo_target":"auctions","run_id":"agent_001_|
   55: {"agent_id":"agent_001","allowed":["Bidding","Auctioning"],"decided_next":"Bidding","fallback":"Bidding","run|
   56: {"agent_id":"agent_001","from":"Auctioning","run_id":"agent_001_1763570352","seq":56,"to":"Bidding","ts":1763|
   57: {"agent_id":"agent_001","current":"Bidding","last_obj_sample":"map[MEMO:map[] NEXT:Bidding]","run_id":"agent_|
   58: {"agent_id":"agent_001","run_id":"agent_001_1763570352","seq":58,"state":"Bidding","ts":1763570421,"type":"st|
Press '?' for help":"agent_001","args":["--compact","--stream=false","--quiet","--model","ollama:gpt-oss:120b","--sys|
```

```
| Path: journal →17
| Key: 17
| Value:
| {
|   "agent_id": "agent_001",
|   "args": [
|     "--compact",
|     "--stream=false",
|     "--quiet",
|     "--model",
|     "ollama:gpt-oss:120b",
|     "--system-prompt",
|     "\u003cinlined-sys\u003e",
|     "--prompt",
|     "\u003cinlined\u003e",
|     "--max-tokens",
|     "2048",
|     "--config",
|     "./config/mcp.json",
|     "--provider-url",
|     "https://ollama.com",
|     "--provider-api-key",
|     "******"
|   ],
|   "cmd": "mcphost",
|   "run_id": "agent_001_1763570352",
|   "seq": 17,
|   "system_prompt_len": 805,
|   "system_prompt_snip": "STATEBAG_JSON={\"agent_id\":\"agent_001\",\"memory\":{\"auctions\":{\"open\":[]},\"bids\":{
|   "ts": 1763570370,
|   "type": "host_call",
|   "user_prompt_len": 1050,
|   "user_prompt_snip": "AGENT_ID=agent_001  TS=1763570370  STATE=Bidding\n\nReturn EXACTLY ONE JSON object {\"NEXT\":
| }
```

Figure 26 – Browsing a native agent's bbolt-backed state file with a TUI bbolt browser. Buckets such as `meta`, `memory`, and `journal` allow inspection of controller configuration, in-run state, and append-only decision logs for a particular agent.

The bbolt browser is especially useful for cross-checking that controller configurations, step transitions, and memo fields match the JSON journals exported for analysis. For example, it can be used to confirm that a given agent's private specialization tags and cost multipliers match the policy expected for Experiment A, or that its bidding-related state evolves as intended under the price-signal and quality-signal policies in Experiments B and C.

As with the block explorer and live dashboard, the bbolt browser is not part of the core experimental pipeline. The browser serves only as a low-level inspection tool that provides additional confidence that the native agent runtime's internal state is consistent with the higher-level metrics and figures reported in the simulation chapters.