Tianjin Daxue Xuebao (Ziran Kexue yu Gongcheng Jishu Ban)/
Journal of Tianjin University Science and Technology
ISSN (Online):0493-2137
E-Publication: Online Open Access
Vol: 58 Issue: 04:2025
DOI: 10.5281/zenodo.18885496

FROM MONOLITHIC APPS TO LIVING SYSTEMS: SOFTWARE DEVELOPMENT MODELS FOR PERSISTENT AI-AGENT ECOSYSTEMS

UMUT GUMELI
Luyani Inc – Founder & CEO, San Francisco, CA, USA.

Abstract

For decades, software development has been dominated by application-centric paradigms in which systems are designed as bounded artifacts with clearly defined lifecycles. Even as architectures evolved from monolithic applications to distributed and service-oriented systems, the fundamental assumption remained unchanged: software is deployed, executed, and eventually replaced, rather than continuously existing as an adaptive entity. The recent emergence of persistent AI agents challenges this assumption by introducing software components that maintain identity, memory, and behavior across extended periods of operation. This paper proposes a shift from traditional application models toward **living software systems**, defined as persistent ecosystems of AI agents that continuously interact, evolve, and coordinate within a shared software environment. Unlike stateless services or short-lived processes, persistent AI agents retain contextual knowledge over time, enabling systems to exhibit long-term behavior, adaptation, and emergent dynamics. This transformation has profound implications for software development models, which must move beyond code-centric abstractions toward behavior-centric design and lifecycle-aware architectures. The study examines how persistent AI-agent ecosystems differ fundamentally from monolithic and conventional distributed systems, both conceptually and architecturally. It introduces a software development perspective that treats persistence, coordination, and behavioral evolution as first-class concerns. Rather than focusing on individual algorithms or agent implementations, the paper analyzes how development models, architectural structures, and lifecycle practices must change to support living systems at scale. The contributions of this work are threefold. First, it provides a clear conceptual distinction between traditional software systems and living systems composed of persistent AI agents. Second, it outlines software development models that enable the design, implementation, and evolution of such systems. Third, it discusses the broader implications of persistent AI-agent ecosystems for software architecture, orchestration, and long-term system governance. By reframing software as a continuously evolving ecosystem rather than a static product, this paper offers a new foundation for AI-native software development.

Keywords: Persistent AI Agents; Living Software Systems; Software Development Models; AI-Agent Ecosystems; Autonomous Software; Behavior-Centric Development.

1. INTRODUCTION

Software development has traditionally conceptualized applications as bounded artifacts with finite lifecycles. In the monolithic paradigm, systems are designed, deployed, executed, and eventually decommissioned, with clear boundaries between development time and runtime. Even as software architectures transitioned toward distributed, service-oriented, and microservice-based systems, this foundational assumption largely persisted. Applications became more modular and scalable, but they remained fundamentally episodic in nature—instantiated to perform predefined functions rather than to exist as continuously evolving entities.

This application-centric worldview increasingly struggles to accommodate modern software requirements. Contemporary systems are expected to operate continuously,

Tianjin Daxue Xuebao (Ziran Kexue yu Gongcheng Jishu Ban)/
Journal of Tianjin University Science and Technology
ISSN (Online):0493-2137
E-Publication: Online Open Access
Vol: 58 Issue: 04:2025
DOI: 10.5281/zenodo.18885496

adapt to changing conditions, and interact dynamically with users, data, and other systems over extended periods. While cloud infrastructure and automation tooling have improved scalability and operational efficiency, they have not fundamentally altered the underlying development model. Software is still treated as something that runs, not something that lives.

The emergence of AI agents introduces a qualitatively different class of software components. Unlike traditional services or functions, AI agents can maintain internal state, accumulate memory, and exhibit behavior that evolves over time. When deployed persistently, these agents form long-lived software entities whose behavior cannot be fully specified at design time. Their actions are shaped not only by input data, but also by historical interactions, learned patterns, and ongoing coordination with other agents. This persistence challenges core assumptions embedded in existing software development models.

Current architectural paradigms provide limited guidance for designing systems composed of such entities. Monolithic applications encapsulate logic within tightly coupled structures, making continuous behavioral evolution difficult. Distributed systems decompose functionality across services, but often rely on stateless interactions and external coordination mechanisms. Even agent-based systems described in prior literature frequently assume short-lived or task-specific agents rather than persistent participants in an ongoing software ecosystem. As a result, software development practices lag behind the capabilities introduced by persistent AI agents.

This gap becomes especially pronounced when multiple persistent agents coexist within the same system. In such environments, behavior emerges not from isolated components but from interactions across the ecosystem. Coordination patterns, memory sharing, and long-term adaptation play a central role in system outcomes. Traditional software development models, which emphasize deterministic execution paths and predefined workflows, offer limited support for reasoning about these emergent dynamics. Without new conceptual and architectural frameworks, systems risk becoming opaque, fragile, and difficult to govern.

In response to these challenges, this paper introduces the concept of **living software systems**—software environments composed of persistent AI-agent ecosystems that continuously operate, adapt, and evolve. Living systems differ from traditional applications not merely in scale or complexity, but in their temporal orientation. They are designed to persist indefinitely, accumulating context and adapting behavior over time. Software development for such systems therefore requires models that explicitly address persistence, behavioral evolution, and ecosystem-level coordination.

The central argument of this study is that supporting persistent AI-agent ecosystems demands a rethinking of software development models. Rather than focusing exclusively on code structure and deployment mechanics, development practices must incorporate behavior design, lifecycle continuity, and long-term system stewardship. Developers shift from defining exact execution paths to shaping behavioral constraints, interaction

Tianjin Daxue Xuebao (Ziran Kexue yu Gongcheng Jishu Ban)/
Journal of Tianjin University Science and Technology
ISSN (Online):0493-2137
E-Publication: Online Open Access
Vol: 58 Issue: 04:2025
DOI: 10.5281/zenodo.18885496

patterns, and evolutionary boundaries. This shift has implications not only for architecture, but also for testing, deployment, maintenance, and governance.

The primary objective of this paper is to examine how software development models must evolve to support living systems built on persistent AI-agent ecosystems. The study does not propose a specific agent implementation or algorithmic technique. Instead, it focuses on structural and conceptual changes required at the software development level. By analyzing the transition from monolithic applications to living systems, the paper seeks to provide a foundation for designing, building, and maintaining AI-native software that operates as a continuously evolving ecosystem.

The remainder of the paper is organized as follows. Section 2 traces the historical evolution from monolithic applications to distributed software systems, highlighting the limitations of existing paradigms. Section 3 introduces persistent AI-agent ecosystems as a new software paradigm. Section 4 examines the defining characteristics of living software systems. Sections 5 through 7 explore development models, architectural structures, and coordination mechanisms for persistent agents. Section 8 discusses implications for the software development lifecycle, followed by a discussion of opportunities and limitations in Section 9. The paper concludes with directions for future research in Section 10.

2. BACKGROUND: FROM MONOLITHIC APPLICATIONS TO DISTRIBUTED SOFTWARE SYSTEMS

The monolithic application model dominated early software development due to its conceptual simplicity and operational predictability. In this paradigm, all application logic, data access, and execution control are encapsulated within a single deployable unit. Development models aligned naturally with this structure: systems were designed as cohesive artifacts, deployed as discrete versions, and managed through clearly defined release cycles. For many years, this approach provided sufficient structure for reasoning about system behavior and correctness.

As software systems grew in scale and scope, the limitations of monolithic architectures became increasingly apparent. Tight coupling between components constrained independent evolution, while centralized execution paths limited scalability and fault isolation. In response, the industry shifted toward distributed architectures, including service-oriented architectures and, later, microservice-based systems. These paradigms decomposed applications into smaller units with well-defined interfaces, enabling parallel development, independent deployment, and improved scalability.

Despite these architectural advances, the underlying software development model remained largely unchanged. Distributed systems continued to treat services as bounded executables with finite lifecycles. Coordination logic was externalized into orchestration layers, configuration management systems, and operational tooling, but decision-making authority remained human-defined and static. Services executed predefined logic in response to requests, and adaptation was primarily reactive rather than proactive.

Tianjin Daxue Xuebao (Ziran Kexue yu Gongcheng Jishu Ban)/
Journal of Tianjin University Science and Technology
ISSN (Online):0493-2137
E-Publication: Online Open Access
Vol: 58 Issue: 04:2025
DOI: 10.5281/zenodo.18885496

The rise of cloud-native infrastructure further accelerated this trend. Elastic scaling, automated deployment pipelines, and infrastructure-as-code practices significantly reduced operational friction. However, these capabilities focused on managing resources rather than reshaping software behavior. Systems became easier to operate, but not fundamentally more adaptive. Automation replaced manual steps, yet the automation itself was rigid, encoding assumptions that required continuous human maintenance as conditions evolved.

In parallel, distributed systems research emphasized statelessness as a core design principle. Stateless services simplified replication, load balancing, and fault tolerance, making them well-suited for large-scale environments. While effective for transactional workloads, statelessness constrained the ability of systems to retain long-term context or exhibit continuity of behavior. State was externalized into databases or caches, reinforcing a separation between computation and memory that limited system-level reasoning. These constraints became more pronounced as applications increasingly relied on data-driven and intelligent behavior. Machine learning models were introduced to enhance specific functions, such as personalization, prediction, or anomaly detection. Architecturally, these models were typically embedded as services or libraries invoked by application logic. While this integration improved functional outcomes, it did not alter the fundamental execution model: intelligence was applied episodically, not persistently.

Agent-based approaches emerged as an alternative abstraction for modeling autonomous behavior in software systems. Early agent frameworks emphasized local decision-making, interaction, and negotiation among software entities. However, many implementations treated agents as short-lived processes or task-specific actors, limiting their ability to accumulate memory or influence long-term system behavior. As a result, agent-based systems often operated alongside, rather than within, mainstream software development models. Taken together, these developments reveal a pattern of incremental architectural change without a corresponding evolution in software development paradigms. Systems became more distributed, scalable, and automated, but they did not fundamentally transition toward continuous, self-directed operation. Coordination remained external, behavior remained predefined, and system evolution remained human-driven.

This historical trajectory highlights a conceptual gap that persistent AI agents begin to fill. By maintaining identity, memory, and behavior across time, persistent agents challenge the assumption that software components should be ephemeral and stateless. They introduce the possibility of systems that not only respond to inputs, but also develop behavior through sustained interaction. Understanding this shift requires moving beyond architectural decomposition toward a new model of software development—one that treats persistence and behavioral continuity as foundational design concerns.

This background sets the stage for introducing persistent AI-agent ecosystems as a distinct software paradigm. The next section formalizes this concept and examines how it differs from both traditional distributed systems and prior agent-based approaches.

Tianjin Daxue Xuebao (Ziran Kexue yu Gongcheng Jishu Ban)/
Journal of Tianjin University Science and Technology
ISSN (Online):0493-2137
E-Publication: Online Open Access
Vol: 58 Issue: 04:2025
DOI: 10.5281/zenodo.18885496

3. PERSISTENT AI-AGENT ECOSYSTEMS AS A NEW SOFTWARE PARADIGM

The concept of persistent AI-agent ecosystems represents a departure from conventional software abstractions that model systems as collections of transient processes. In this paradigm, agents are not instantiated solely to execute isolated tasks; they persist over time as identifiable entities with memory, context, and evolving behavior. Persistence transforms agents from functional components into long-lived participants within a shared software environment, fundamentally altering how systems are designed and developed.

A persistent AI agent can be understood as a software entity that maintains continuity of identity and internal state across interactions and execution cycles. Unlike stateless services that derive behavior exclusively from current inputs, persistent agents incorporate historical context into decision-making. This continuity enables agents to learn from prior outcomes, adapt strategies, and develop patterns of behavior that unfold over extended periods. From a software development perspective, persistence introduces temporal depth as a core design dimension.

When multiple persistent agents coexist, they form an ecosystem rather than a collection of independent components. In such ecosystems, system behavior emerges from interactions among agents, shared context, and environmental constraints. Coordination is not imposed solely through centralized orchestration, but arises through ongoing negotiation, adaptation, and mutual influence. This emergent quality distinguishes persistent AI-agent ecosystems from both monolithic systems and conventional distributed architectures, which rely on predefined interaction patterns.

Existing agent-based systems often fall short of this paradigm due to limited persistence or scope. Many agent frameworks assume agents that are created, execute tasks, and terminate, mirroring job-oriented execution models. While suitable for discrete problem-solving, these approaches do not support long-term behavioral evolution or ecosystem-level dynamics. Persistent AI-agent ecosystems, by contrast, are designed to operate indefinitely, accumulating context and shaping system behavior over time.

From a software development standpoint, this shift challenges established assumptions about system boundaries and responsibilities. In traditional models, application logic encapsulates behavior, while infrastructure and orchestration manage execution. Persistent agent ecosystems blur these distinctions by embedding behavior, memory, and coordination within the agents themselves. Development models must therefore account for how agent behavior is specified, constrained, and evolved, rather than treating behavior as static code paths.

Another defining characteristic of persistent AI-agent ecosystems is their relationship to environment and context. Agents continuously perceive and influence their environment, which includes not only external inputs but also the state and behavior of other agents. This bidirectional interaction creates feedback loops that shape collective dynamics. Software development models must accommodate these feedback mechanisms, ensuring that emergent behavior remains intelligible and aligned with system objectives.

Tianjin Daxue Xuebao (Ziran Kexue yu Gongcheng Jishu Ban)/
Journal of Tianjin University Science and Technology
ISSN (Online):0493-2137
E-Publication: Online Open Access
Vol: 58 Issue: 04:2025
DOI: 10.5281/zenodo.18885496

Importantly, persistence does not imply unchecked autonomy. Persistent agents operate within architectural and governance constraints that define acceptable behavior and interaction patterns. These constraints shape the ecosystem's evolution, balancing flexibility with control. Designing such constraints becomes a central concern of software development, replacing traditional assumptions of determinism with bounded adaptability.

By framing persistent AI-agent ecosystems as a distinct software paradigm, this section establishes a conceptual foundation for the notion of living systems introduced in this paper. Persistence, interaction, and behavioral evolution are not incidental features; they are defining properties that require new development models and architectural thinking. The following section examines the characteristics that distinguish living software systems from traditional applications and distributed systems.

4. CHARACTERISTICS OF LIVING SOFTWARE SYSTEMS

Living software systems differ from traditional applications not merely in their use of artificial intelligence, but in how they exist, behave, and evolve over time. Whereas conventional software is designed to execute predefined logic in response to inputs, living systems are structured to persist, adapt, and exhibit behavior that unfolds continuously. These characteristics redefine foundational assumptions in software development, particularly around system identity, temporality, and control.

A defining characteristic of living software systems is **persistence beyond execution cycles**. Traditional applications are instantiated, run, and terminated, with continuity maintained externally through data stores or configuration state. In living systems, persistence is intrinsic to the system's components—particularly persistent AI agents—which retain identity and internal state across interactions. This persistence enables continuity of behavior and allows the system to accumulate experience rather than merely process transactions.

Another core characteristic is **contextual continuity**. Living systems maintain an integrated understanding of their operational context over time, incorporating signals from past interactions, environmental changes, and internal dynamics. Context is not reconstructed anew for each execution, but evolves alongside the system. From a software development perspective, this requires architectures that support long-lived context propagation and mechanisms for reconciling short-term events with long-term patterns.

Living systems are also distinguished by **behavioral evolution**. In conventional software, behavior changes primarily through redeployment or configuration updates initiated by humans. In living systems, behavior can evolve incrementally through interaction and learning, even in the absence of explicit redeployment. This evolution does not imply unrestricted self-modification; rather, it occurs within boundaries defined by development-time constraints and architectural governance. Software development models must therefore account for behavior that is neither static nor entirely predetermined.

Tianjin Daxue Xuebao (Ziran Kexue yu Gongcheng Jishu Ban)/
Journal of Tianjin University Science and Technology
ISSN (Online):0493-2137
E-Publication: Online Open Access
Vol: 58 Issue: 04:2025
DOI: 10.5281/zenodo.18885496

Interaction-driven dynamics further differentiate living systems from traditional architectures. In ecosystems of persistent AI agents, system behavior emerges from interactions among agents rather than from centralized control logic alone. These interactions may be cooperative, competitive, or complementary, producing outcomes that cannot be fully predicted from individual agent behavior. Designing for such dynamics requires development models that emphasize interaction patterns, coordination mechanisms, and systemic observability.

Another characteristic is **temporal openness**. Living systems are not designed with a fixed endpoint or completion condition. They are intended to operate indefinitely, adapting to new requirements, environments, and constraints. This openness challenges traditional lifecycle assumptions in software development, which often emphasize versioned releases and finite maintenance phases. For living systems, evolution is continuous, and development becomes an ongoing process of shaping system behavior rather than delivering discrete artifacts.

Human–system relationships also change in living software systems. Developers and operators no longer act as primary controllers of system behavior, but as stewards who define goals, constraints, and acceptable behavior ranges. The system assumes responsibility for navigating the space of possible actions within those boundaries. This shift elevates the importance of transparency, interpretability, and governance mechanisms that allow humans to understand and influence system evolution without micromanagement.

Finally, living software systems exhibit **resilience through adaptation** rather than through static redundancy alone. Traditional resilience strategies emphasize replication, failover, and predefined recovery procedures. Living systems complement these mechanisms with adaptive behavior that adjusts strategies based on observed outcomes. Software development models must therefore integrate resilience considerations into behavior design, rather than treating them solely as operational concerns.

Together, these characteristics illustrate that living software systems represent a qualitative shift in how software is conceived and built. Persistence, contextual continuity, behavioral evolution, and interaction-driven dynamics are not incremental enhancements; they redefine the role of software as a continuously evolving entity. Understanding these characteristics provides the foundation for examining how software development models must change to support persistent AI-agent ecosystems, a topic addressed in the following section.

5. SOFTWARE DEVELOPMENT MODELS FOR AI-AGENT-BASED SYSTEMS

Persistent AI-agent ecosystems require software development models that differ fundamentally from those designed for monolithic or service-based applications. Traditional models assume that system behavior can be fully specified through code and configuration at development time. In contrast, AI-agent-based systems exhibit behavior that emerges through interaction, learning, and long-term context accumulation. As a

Tianjin Daxue Xuebao (Ziran Kexue yu Gongcheng Jishu Ban)/
Journal of Tianjin University Science and Technology
ISSN (Online):0493-2137
E-Publication: Online Open Access
Vol: 58 Issue: 04:2025
DOI: 10.5281/zenodo.18885496

result, software development shifts from defining exact execution logic toward shaping behavioral spaces within which agents operate.

One of the most significant changes is the transition from **code-centric development** to **behavior-centric design**. In conventional software projects, correctness is achieved by encoding explicit rules and control flows. For persistent AI agents, behavior is influenced by internal state, memory, and interaction history, making it impractical to enumerate all possible execution paths. Developers instead define objectives, constraints, and evaluation criteria that guide agent behavior over time. Software artifacts thus increasingly represent intent and boundaries rather than exhaustive procedural logic.

This shift alters how requirements are specified and validated. Traditional requirements focus on functional outputs and deterministic responses. In agent-based systems, requirements must capture acceptable behavioral ranges, long-term tendencies, and interaction outcomes. Validation becomes probabilistic and longitudinal, assessing whether system behavior remains aligned with intended goals across time rather than whether specific outputs are produced in isolation. Software development models must therefore incorporate continuous evaluation as a core activity rather than a post-development phase.

The design of agent behavior introduces new abstractions into the development process. Developers must reason about state representation, memory persistence, and learning boundaries as first-class concerns. Decisions about what agents should remember, forget, or generalize from experience directly influence system behavior. These considerations blur the traditional separation between application logic and data management, requiring integrated development practices that account for both simultaneously.

Collaboration patterns among development teams also evolve. In monolithic systems, teams often organize around functional modules or services. In AI-agent ecosystems, teams may organize around behavioral domains, agent roles, or interaction protocols. This reorganization reflects the fact that system behavior emerges from coordinated agent activity rather than from isolated components. Development models must support clear ownership of behavioral responsibilities while enabling cross-cutting coordination.

Testing strategies further illustrate the divergence from traditional development models. Deterministic unit tests and integration tests remain useful for validating infrastructure and interaction contracts, but they are insufficient for assessing agent behavior over time. Development models for AI-agent-based systems emphasize scenario-based testing, simulation, and continuous monitoring. Testing becomes an ongoing process that accompanies system operation, enabling developers to detect behavioral drift and unintended interactions as they arise.

Versioning and deployment practices are similarly affected. In conventional systems, new behavior is introduced through discrete releases. Persistent AI agents may evolve behavior continuously through learning and interaction, reducing the significance of traditional version boundaries. Software development models must therefore reconcile

Tianjin Daxue Xuebao (Ziran Kexue yu Gongcheng Jishu Ban)/
Journal of Tianjin University Science and Technology
ISSN (Online):0493-2137
E-Publication: Online Open Access
Vol: 58 Issue: 04:2025
DOI: 10.5281/zenodo.18885496

continuous behavioral change with the need for traceability and rollback. This often involves separating agent capabilities, which change infrequently, from agent behavior, which evolves more fluidly within defined constraints.

Finally, development models for persistent AI-agent ecosystems emphasize **stewardship over control**. Developers act as designers of environments and guardians of system behavior rather than as direct authors of every action. This role requires tools and practices that support observability, interpretability, and governance, ensuring that autonomous behavior remains understandable and aligned with organizational intent.

In summary, software development models for AI-agent-based systems depart from traditional application development by prioritizing behavior design, long-term evaluation, and ecosystem-level coordination. These models reflect the realities of living software systems, where persistence and interaction redefine what it means to build, test, and evolve software. The next section examines how these models are supported and constrained by the underlying architecture of persistent AI-agent ecosystems.

6. ARCHITECTURE OF PERSISTENT AI-AGENT ECOSYSTEMS

The architecture of persistent AI-agent ecosystems departs from conventional application and service-based architectures by treating agents as long-lived architectural entities rather than ephemeral execution units. In traditional systems, architectural concerns focus on request handling, data storage, and service coordination. In persistent agent ecosystems, architecture must additionally support identity continuity, memory management, and sustained interaction across time. These requirements introduce new structural considerations that reshape how software systems are composed.

At the core of such architectures lies the concept of an **agent lifecycle** that extends beyond individual tasks or sessions. Persistent agents are created with the expectation that they will remain active participants in the system for extended periods, accumulating context and adapting behavior. Architectural support for this lifecycle includes mechanisms for agent initialization, state persistence, recovery, and graceful evolution. Unlike stateless services that can be freely restarted without semantic impact, persistent agents require careful handling to preserve continuity and behavioral integrity.

Coordination among agents represents another foundational architectural concern. In monolithic systems, coordination is implicit, enforced through shared memory and centralized control flow. In distributed systems, coordination is externalized through orchestration layers and messaging infrastructure. Persistent AI-agent ecosystems adopt a hybrid approach in which coordination emerges from structured interaction protocols and shared contextual understanding. Architecture must therefore provide communication primitives that support both direct interaction and indirect influence through shared state or environmental signals.

System memory plays a central role in enabling persistence and continuity. Architectural designs must distinguish between **agent-local memory**, which captures individual experience and internal state, and **ecosystem-level memory**, which represents shared

Tianjin Daxue Xuebao (Ziran Kexue yu Gongcheng Jishu Ban)/
Journal of Tianjin University Science and Technology
ISSN (Online):0493-2137
E-Publication: Online Open Access
Vol: 58 Issue: 04:2025
DOI: 10.5281/zenodo.18885496

knowledge, norms, or environmental context. Effective architectures balance these layers to prevent excessive coupling while enabling meaningful coordination. Decisions about memory scope and accessibility directly influence system dynamics and emergent behavior.

Another architectural consideration is **identity management**. Persistent agents require stable identities that persist across execution contexts and system changes. These identities enable agents to form long-term relationships, maintain reputational context, and reason about past interactions. From a software development perspective, identity becomes an architectural primitive rather than a peripheral concern. Architectures must support identity continuity even as underlying infrastructure evolves.

The interaction between agents and their environment further shapes architectural design. In living systems, the environment is not a passive backdrop but an active participant that influences and is influenced by agent behavior. Architectural support for environment representation includes event streams, shared state abstractions, and feedback channels that convey the consequences of actions. This interaction layer enables agents to perceive system-wide effects and adjust behavior accordingly.

Scalability in persistent AI-agent ecosystems presents unique challenges. While distributed systems scale by replicating stateless components, persistent agents cannot be duplicated arbitrarily without violating continuity assumptions. Architecture must therefore support scaling strategies that preserve agent identity and memory while accommodating growth in system complexity. Techniques such as sharding agent populations, partitioning interaction spaces, and isolating behavioral domains become central architectural tools.

Finally, architectural governance mechanisms are essential to maintaining system coherence over time. As agents adapt and interactions evolve, the risk of unintended emergent behavior increases. Persistent agent ecosystems must incorporate architectural constraints that define acceptable interaction patterns, resource usage, and behavioral boundaries. These constraints act as stabilizing forces, enabling autonomy and evolution without sacrificing predictability or control.

In sum, the architecture of persistent AI-agent ecosystems is shaped by the need to support continuity, interaction, and evolution at scale. By elevating agents to first-class architectural entities, these systems move beyond application-centric design toward structures that sustain living software. The following section examines how orchestration and coordination operate within these ecosystems, and how emergent behavior can be guided without imposing rigid control.

7. ORCHESTRATION, COORDINATION, AND EMERGENT SYSTEM BEHAVIOR

Orchestration in persistent AI-agent ecosystems differs fundamentally from coordination mechanisms used in traditional software systems. Conventional orchestration models are built around predefined workflows, explicit control flow, and deterministic execution sequences. These models assume that the order of actions and their dependencies can

Tianjin Daxue Xuebao (Ziran Kexue yu Gongcheng Jishu Ban)/
Journal of Tianjin University Science and Technology
ISSN (Online):0493-2137
E-Publication: Online Open Access
Vol: 58 Issue: 04:2025
DOI: 10.5281/zenodo.18885496

be fully specified at design time. In ecosystems composed of persistent agents, this assumption no longer holds. System behavior arises from ongoing interactions among agents whose internal states evolve continuously.

In such environments, orchestration shifts from enforcing sequences to shaping interaction conditions. Rather than dictating exactly how agents should act, the system defines constraints, incentives, and communication channels that influence agent behavior. Coordination becomes a dynamic process in which agents negotiate, adapt, and respond to one another over time. Software development models must therefore account for coordination as an emergent property rather than a centrally imposed structure. Emergent behavior is a defining characteristic of persistent AI-agent ecosystems. Emergence refers to system-level patterns that arise from local interactions without being explicitly programmed. In living software systems, emergent behavior may manifest as stable interaction norms, adaptive load distribution, or collective problem-solving strategies. While emergence can produce beneficial outcomes, it also introduces unpredictability. Software development must balance the creative potential of emergence with the need for reliability and control.

Architectural support for coordination plays a critical role in managing this balance. Persistent agent ecosystems rely on shared communication substrates, contextual signals, and feedback mechanisms that enable agents to align behavior without direct supervision. These substrates include event streams, shared state representations, and interaction protocols that convey not only actions but also intent and outcome. Effective orchestration emerges when agents can reason about the consequences of their actions within a broader system context.

Unlike centralized orchestration, which simplifies reasoning at the cost of flexibility, decentralized coordination distributes decision-making across agents. This distribution enhances scalability and resilience but complicates predictability. Software development models must therefore incorporate tools and practices that allow developers to observe, analyze, and influence emergent behavior over time. Observability becomes a prerequisite for trust in autonomous coordination, enabling developers to detect undesirable dynamics and adjust constraints accordingly.

Another challenge lies in conflict resolution. In persistent ecosystems, agents may pursue objectives that partially overlap or compete. Without appropriate coordination mechanisms, such conflicts can lead to inefficiency or instability. Rather than resolving conflicts through rigid priority schemes, living systems often rely on adaptive resolution strategies informed by context and historical outcomes. Software development models must support the specification of conflict boundaries and acceptable trade-offs, allowing the ecosystem to converge toward stable behavior.

Importantly, orchestration in persistent agent ecosystems does not eliminate the role of human oversight. Developers and operators remain responsible for defining system goals, monitoring behavior, and intervening when necessary. However, their role shifts from direct control to governance. They influence system behavior indirectly by adjusting

Tianjin Daxue Xuebao (Ziran Kexue yu Gongcheng Jishu Ban)/
Journal of Tianjin University Science and Technology
ISSN (Online):0493-2137
E-Publication: Online Open Access
Vol: 58 Issue: 04:2025
DOI: 10.5281/zenodo.18885496

constraints, interaction rules, and evaluation criteria. This governance-oriented approach aligns with the broader shift from control-centric to behavior-centric software development.

In summary, orchestration and coordination in persistent AI-agent ecosystems represent a move away from deterministic workflows toward emergent, interaction-driven behavior. Software development models must evolve to support this transition, providing architectural structures and governance mechanisms that enable autonomy while maintaining coherence. The next section examines how these changes affect the broader software development lifecycle, from design and testing to deployment and maintenance.

8. IMPLICATIONS FOR SOFTWARE DEVELOPMENT LIFECYCLE

Persistent AI-agent ecosystems fundamentally reshape the software development lifecycle by challenging assumptions about determinism, versioning, and control. Traditional lifecycles are organized around discrete phases—design, implementation, testing, deployment, and maintenance—each bounded by human decision points. Living software systems, by contrast, operate continuously and evolve through interaction, requiring development models that accommodate ongoing behavioral change.

Design activities shift from specifying exhaustive functionality to defining behavioral constraints and interaction patterns. Developers must articulate goals, limits, and acceptable outcomes rather than fixed execution paths. This requires new forms of specification that capture long-term tendencies and system-level objectives. As a result, architectural intent and governance become central artifacts of the development process.

Testing practices also evolve. While conventional testing validates deterministic outputs, persistent agent ecosystems demand longitudinal evaluation. Developers rely on simulations, scenario-based testing, and continuous monitoring to assess whether behavior remains within acceptable bounds over time. Testing becomes a persistent activity embedded into system operation rather than a pre-release gate.

Deployment and release management are similarly transformed. In living systems, behavior may evolve without discrete redeployments, reducing the relevance of traditional version boundaries. Software development models must therefore separate capability updates—which remain versioned—from behavioral evolution—which occurs continuously within defined constraints. This separation preserves traceability while enabling adaptive behavior. Maintenance transitions from reactive issue resolution to proactive stewardship. Developers monitor emergent behavior, adjust constraints, and refine interaction rules as the system evolves. Maintenance thus becomes an extension of development rather than a downstream activity. This continuity reinforces the notion that living systems are never “finished,” but are continuously shaped over time.

Overall, persistent AI-agent ecosystems require a lifecycle model centered on governance, observation, and iterative refinement. Software development becomes an ongoing partnership between human designers and autonomous systems, emphasizing alignment and adaptability over static correctness.

Tianjin Daxue Xuebao (Ziran Kexue yu Gongcheng Jishu Ban)/
Journal of Tianjin University Science and Technology
ISSN (Online):0493-2137
E-Publication: Online Open Access
Vol: 58 Issue: 04:2025
DOI: 10.5281/zenodo.18885496

9. DISCUSSION: OPPORTUNITIES AND LIMITATIONS OF LIVING SOFTWARE SYSTEMS

Living software systems offer significant opportunities for building adaptable, resilient, and context-aware applications. Persistent AI-agent ecosystems enable systems to accumulate experience, coordinate dynamically, and respond to changing environments without constant human intervention. These properties can reduce operational overhead and unlock new forms of software behavior that are difficult to achieve with static architectures.

At the same time, these systems introduce limitations and risks. Emergent behavior complicates predictability, making it challenging to guarantee outcomes in all conditions. Governance mechanisms must therefore balance autonomy with accountability, ensuring that adaptation does not undermine reliability or trust. Additionally, the cognitive load on developers may increase as they reason about long-term behavior rather than discrete functions.

Another limitation concerns evaluation and assurance. Because behavior evolves over time, establishing confidence in system correctness requires sustained observation and analysis. This places new demands on tooling, documentation, and organizational processes. Addressing these challenges is essential for the responsible adoption of living software systems.

Despite these limitations, the opportunities presented by persistent AI-agent ecosystems justify continued exploration. With appropriate architectural constraints and development practices, living systems can remain governable while delivering adaptive capabilities beyond the reach of traditional models.

10. CONCLUSION AND FUTURE RESEARCH DIRECTIONS

This paper examined the transition from monolithic applications to living software systems composed of persistent AI-agent ecosystems. By framing persistence, interaction, and behavioral evolution as core software development concerns, it highlighted the limitations of traditional application-centric models and proposed a new foundation for AI-native development.

The analysis showed that supporting living systems requires changes across development models, architecture, orchestration, and lifecycle practices. Persistent AI agents introduce temporal depth and emergent dynamics that cannot be managed through static workflows alone. Software development must therefore emphasize behavior-centric design, continuous evaluation, and system stewardship.

Future research should explore empirical validation of persistent agent ecosystems, governance mechanisms for emergent behavior, and tooling that supports long-term observability and control. As AI capabilities continue to advance, living software systems provide a promising paradigm for building adaptive, resilient applications that evolve alongside their environments.

Tianjin Daxue Xuebao (Ziran Kexue yu Gongcheng Jishu Ban)/
Journal of Tianjin University Science and Technology
ISSN (Online):0493-2137
E-Publication: Online Open Access
Vol: 58 Issue: 04:2025
DOI: 10.5281/zenodo.18885496

References

1) Bass, L., Clements, P., & Kazman, R. (2021). *Software Architecture in Practice* (4th ed.). Addison-Wesley.

2) Booch, G. (2011). *Object-Oriented Analysis and Design with Applications* (3rd ed.). Addison-Wesley.

3) Newman, S. (2021). *Building Microservices: Designing Fine-Grained Systems* (2nd ed.). O'Reilly Media.

4) Kleppmann, M. (2017). *Designing Data-Intensive Applications*. O'Reilly Media. 5. Fowler, M. (2015). The monolith first. *martinfowler.com*.

5) Wooldridge, M. (2009). *An Introduction to MultiAgent Systems* (2nd ed.). Wiley.

6) Jennings, N. R. (2000). On agent-based software engineering. *Artificial Intelligence*, 117(2), 277–296.

7) Jennings, N. R., Sycara, K., & Wooldridge, M. (1998). A roadmap of agent research and development. *Autonomous Agents and Multi-Agent Systems*, 1(1), 7–38.

8) Weyns, D., Omicini, A., & Odell, J. (2010). Environment as a first-class abstraction in multiagent systems. *Autonomous Agents and Multi-Agent Systems*, 14(1), 5–30.

9) de Lemos, R., Giese, H., Müller, H. A., Shaw, M., Andersson, J., Litoiu, M., Schmerl, B., Tamura, G., Villegas, N. M., Vogel, T., Weyns, D., & Baresi, L. (2013). Software engineering for self-adaptive systems: A second research roadmap. *Software Engineering for Self-Adaptive Systems II*, Springer, 1–32.

10) Brun, Y., Di Marzo Serugendo, G., Gacek, C., Giese, H., Kienle, H. M., Litoiu, M., Müller, H. A., Pezzè, M., & Shaw, M. (2009). Engineering self-adaptive systems through feedback loops. *Software Engineering for Self-Adaptive Systems*, Springer, 48–70.

11) Salehie, M., & Tahvildari, L. (2009). Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems*, 4(2), 1–42.

12) Kephart, J. O., & Chess, D. M. (2003). The vision of autonomic computing. *IEEE Computer*, 36(1), 41–50.

13) Hohpe, G., & Woolf, B. (2003). *Enterprise Integration Patterns*. Addison-Wesley.

14) Richards, M., & Ford, N. (2020). *Fundamentals of Software Architecture*. O'Reilly Media.

15) Omicini, A., Ricci, A., & Viroli, M. (2008). Artifacts in the A&A meta-model for multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 17(3), 432–456.

16) North, M. J., & Macal, C. M. (2007). *Managing Business Complexity: Discovering Strategic Solutions with Agent-Based Modeling and Simulation*. Oxford University Press.

17) Holland, J. H. (1992). Complex adaptive systems. *Daedalus*, 121(1), 17–30.

18) Dobson, S., Denazis, S., Fernández, A., Gaïti, D., Gelenbe, E., Massacci, F., Nixon, P., Saffre, F., Schmidt, N., & Zambonelli, F. (2006). A survey of autonomic communications. *ACM Transactions on Autonomous and Adaptive Systems*, 1(2), 223–259.

19) Weyns, D. (2019). *Software Engineering of Self-Adaptive Systems*. Springer.