Agent-first Driven Development: A Bottom-Up Methodology for Building Software in the Age of AI Agents

Masaomi Hatakeyama
Genomics on Blockchain
14 February 2026

Abstract

The rapid proliferation of AI coding assistants and agent frameworks has produced several competing development methodologies—from unstructured vibe coding to Spec-Driven Development (SDD) and Agent-Driven Development (ADD). However, these approaches share a common assumption: humans write specifications, tests, or instructions, and AI agents implement them. The design origin remains human-centric. This paper proposes Agent-first Driven Development (AFD), a bottom-up methodology where agent tools serve as the primary interface and source of truth for system design. In AFD, developers implement domain logic as agent tools first, validate through agent interaction, extract deterministic operations into backend APIs, and finally add human-facing UIs—yielding a dual-interface architecture that serves both AI agents and human users. AFD introduces six principles, including Zero-Gap Deployment (tool implementation is simultaneously deployment) and Classify by Cognition (separating API-able from Agent-native operations). We contextualize AFD within a comprehensive review of AI-era development methodologies and demonstrate its application through KairosChain, an MCP-based meta ledger for auditable AI skill evolution. AFD fills a methodological gap: while the industry actively converts existing APIs into agent tools (top-down), AFD provides the missing bottom-up direction for greenfield development.

Keywords: agent-first development, MCP, tool-driven design, API extraction, dual-interface architecture, software methodology, KairosChain

---

Figure 1: Agent-first Driven Development (AFD) overview. Existing approaches (SDD, ADD, TDD+Agents) flow top-down from human specifications to AI implementation. AFD inverts this: agent tools are built first, validated through interaction, then progressively extracted into APIs and UIs.

---

1. Introduction

Software development methodologies have historically evolved by redefining the primary artifact around which the development process revolves. Object-Oriented Programming (OOP) placed objects at the center. Test-Driven Development (TDD) made tests the first artifact written [1]. Behavior-Driven Development (BDD) elevated user stories and acceptance criteria. Domain-Driven Design (DDD) organized systems around bounded contexts and ubiquitous language [2]. Each paradigm shift was, in essence, a declaration of a new "first thing to build."

The emergence of Large Language Model (LLM) agents as development partners has created an urgent need for the next such declaration. In 2025, Andrej Karpathy coined "vibe coding" to describe conversational, improvisational code generation with AI [3]. While vibe coding lowered the barrier to entry, research quickly revealed its limitations: unstructured AI code generation led to significant productivity challenges on complex

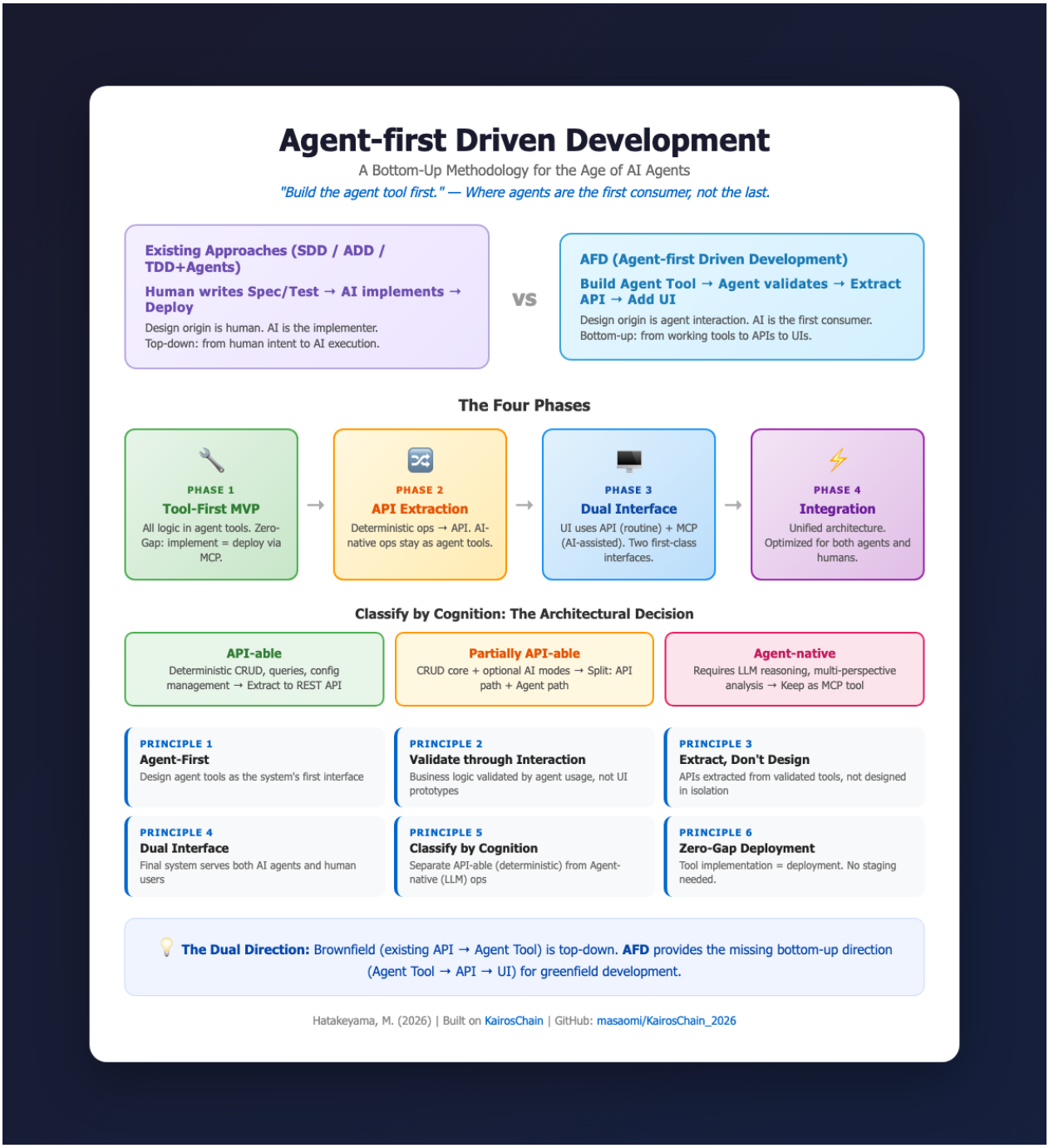


Figure 1: Figure 1: Agent-first Driven Development overview — six principles, four phases, and the Cognitive Classification Pattern

codebases, and generated code suffered from entropy loops where each fix introduced new bugs [4, 5]. The need for structured AI development methodologies became apparent.

Several responses emerged. Spec-Driven Development (SDD) proposed writing specifications as the primary artifact, with AI generating code from specs [6, 7]. Agent-Driven Development (ADD) positioned humans as "Editors" supervising AI implementers [8]. AI-Native Software Engineering (SE 3.0) envisioned intent-centric, conversation-oriented development [9]. TDFlow combined TDD cycles with agentic workflows [10].

Yet these approaches share a fundamental assumption: **the design origin is human**. Humans write specs (SDD), provide editorial direction (ADD), express intent (SE 3.0), or write tests (TDFlow). AI is cast as the *implementer*—a more capable pair programmer, but still responding to human-originated designs.

This paper proposes a different starting point. **Agent-first Driven Development (AFD)** treats AI agents as the *first consumer* of the system, not the last. The development process begins by implementing domain logic as agent tools—structured interfaces that AI agents invoke—and progressively extracts APIs and UIs from validated tool interactions. The agent is not the implementer; the agent is the first user.

This inversion is not merely philosophical. It has concrete architectural consequences: APIs are extracted from empirically validated tool usage patterns rather than designed in isolation; the final system naturally supports both agent and human interfaces; and operations are classified by their cognitive requirements (deterministic vs. AI-dependent), yielding a novel architectural pattern.

The contributions of this paper are threefold:

1. **A systematic review** of AI-era development methodologies (2025–2026), identifying their shared top-down assumption and the gap that AFD addresses.
2. **The AFD methodology**, comprising six principles, a four-phase workflow, and the Cognitive Classification Pattern for architectural decisions.
3. **A case study** demonstrating AFD through KairosChain [11], an MCP-based meta ledger where 20+ agent tools serve as the foundation for progressive API and UI extraction.

---

2. Related Work: AI-Era Development Methodologies

This section reviews the major development methodologies that have emerged in response to AI agents becoming integral to software development. We organize them along a spectrum from unstructured to highly structured approaches, and identify the common pattern they share.

2.1 Unstructured Approaches: Vibe Coding and Signal Coding

Karpathy [3] introduced "vibe coding" in February 2025 as a conversational AI programming paradigm where developers rely on AI agents to generate code through iterative prompting, prioritizing speed and application results over code quality. While effective for rapid prototyping, empirical studies revealed significant limitations. Ge et al. [4] surveyed the vibe coding landscape and documented pain points including unclear specification requirements, debugging challenges, and collaboration difficulties. Industry analyses reported that developers could be significantly slower on complex codebases and generate substantially more code per task when relying on unstructured AI code generation [5].

The "entropy loop" problem proved particularly damaging: initial rapid progress gives way to a brittle, incoherent codebase where each fix introduces new bugs [5]. This led to Signal Coding [12], which proposed

combining rapid AI iteration with structured development practices—code structure, patterns, and tests—to improve AI agent performance. While an improvement, Signal Coding remains a refinement of vibe coding rather than a fundamentally new paradigm.

2.2 Spec-Driven Development (SDD)

Spec-Driven Development emerged as a more disciplined alternative, reversing traditional development by writing detailed specifications before code. Fowler [6] explored three maturity levels: *spec-first* (specifications guide initial development, then are discarded), *spec-anchored* (specifications persist for ongoing evolution), and *spec-as-source* (humans edit only specifications; AI generates and maintains all code). GitHub's Spec Kit [7], Kiro, and Tessl represent active implementations of this approach.

Enterprise teams adopting SDD with multi-agent systems reported a 34.2% reduction in task completion time and elimination of 40–60% coordination overhead [13]. SDD addresses vibe coding's chaos by providing unambiguous instructions to AI agents, treating specifications as executable blueprints.

However, SDD's core assumption is that **humans write specifications and AI implements them**. The specification is the source of truth, and the direction of design flows from human intent to AI execution.

2.3 Agent-Driven Development (ADD)

Jansen [8] proposed Agent-Driven Development as a structured methodology where AI agents and human developers collaborate through a defined process. The human, termed the "Editor," provides direction, domain expertise, and critical thinking, while agents handle implementation, documentation, testing, and versioning. Jansen emphasized that "trust is not a prerequisite; it's a product of process"—strong DevOps practices (Infrastructure as Code, CI/CD, automated tests) serve as the safety net enabling agent operation.

Notably, Jansen later acknowledged the conceptual convergence between ADD and SDD, writing in November 2025: "This is now known as spec-driven development" [8, comment]. This convergence reveals that ADD and SDD occupy the same fundamental design space: humans design, AI implements.

2.4 AI-Native Software Engineering (SE 3.0)

Hassan et al. [9] proposed SE 3.0 as a vision for intent-centric, conversation-oriented development where AI systems evolve beyond task-driven copilots into intelligent collaborators capable of reasoning about software engineering principles. This academic vision paper anticipated infrastructure changes including AI-native version control, prompts as source code, and AI-native IDEs.

The DORA 2025 report [14] provided empirical grounding, finding that AI primarily functions as an amplifier—magnifying both the strengths of high-performing organizations and the dysfunctions of struggling ones. Andreessen Horowitz [15] identified nine emerging developer patterns for the AI era, including the Model Context Protocol (MCP) as a step toward deeper AI integration.

While SE 3.0 provides a compelling vision, it does not prescribe a concrete development process or methodology.

2.5 TDD with Agentic Workflows

The integration of TDD with AI agents has produced promising results. TDFlow [10], a test-driven agentic workflow, achieved an 88.8% pass rate on SWE-Bench Lite by decomposing software repair into four subagent components governed by patch proposing, debugging, and revision. Multi-agent testing frameworks

[16] demonstrated 60% reduction in invalid tests and 30% coverage improvement compared to single-model baselines.

The Tweag Agentic Coding Handbook [17] articulated why TDD and agents are natural complements: tests provide precise specifications that keep LLMs focused on small, testable goals, reducing hallucination and enabling validation at each step. However, TDD + Agents extends the existing TDD paradigm rather than proposing a new primary artifact.

2.6 Common Pattern and Gap

Across all reviewed methodologies, a common pattern emerges: **the direction of design flows from human to AI**. Humans write specifications (SDD), provide editorial direction (ADD), express intent (SE 3.0), or write tests (TDD + Agents). AI is positioned as the implementer—a sophisticated tool that executes human-originated designs (Table 1).

Table 1: AI-Era Development Methodologies Compared

Methodology	Primary Artifact	Design Origin	AI Role	Direction
Vibe Coding	Conversation	Human	Generator	Human → AI
SDD	Specification	Human	Implementer	Human → AI
ADD	Editorial direction	Human	Implementer	Human → AI
SE 3.0	Intent	Human	Collaborator	Human → AI
TDD + Agents	Tests	Human	Implementer	Human → AI
AFD	**Agent tools**	**Human (tool design)**	**First consumer / validator**	**Tool → API → UI**

A notable gap exists: no current methodology treats agent tools as the design origin and progressively extracts APIs and UIs from validated agent interactions. Furthermore, while the industry actively develops patterns for converting existing APIs into agent tools—a top-down mapping from REST endpoints to MCP tools [18, 19]—the reverse direction for greenfield development remains undefined. AFD addresses this gap.

---

3. Agent-first Driven Development

3.1 Core Philosophy

AFD rests on a single insight: **agent tools can serve as the source of truth for system design**. Just as TDD made tests the first artifact (and thereby the first consumer of the code under test), AFD makes agent tools the first artifact (and thereby the first consumer of the business logic).

The structural parallel to TDD is precise:

```
TDD:  Write Test  → Run (Red)          → Implement (Green)  → Refactor
AFD:  Define Tool → Agent Validates    → Extract API        → Add UI
```

In TDD, writing a test before implementation forces the developer to think about the interface and expected behavior. In AFD, defining an agent tool forces the developer to think about the operation's semantics, input schema, and output contract—all from the perspective of an AI consumer. This constraint produces cleaner,

more composable interfaces because agent tools must be self-describing (with names, descriptions, and JSON schemas) to be usable by LLMs.

The key difference from existing methodologies is the **direction of extraction**:

- **SDD/ADD**: Human-written spec → AI-generated code → Deploy
- **API-to-MCP** (industry trend): Existing API → Agent tool mapping [18, 19]
- **AFD**: Agent tool → Validated interaction → Extracted API → Human UI

AFD and the API-to-MCP pattern are complementary: the former serves greenfield development, the latter serves brownfield migration. Both directions are necessary for a complete AI-era development ecosystem.

3.2 The Six Principles of AFD

We propose six principles that define the AFD methodology:

Principle 1: Agent-First. Design agent tools as the system's first interface. Before any API endpoint or UI component exists, domain operations should be expressible as agent tools with defined names, descriptions, input schemas, and output contracts.

Principle 2: Validate through Interaction. Validate business logic through actual agent interaction, not through UI prototypes or specification review. When an AI agent uses a tool to perform a domain operation, the interaction reveals whether the tool's interface is well-designed, whether the operation's granularity is appropriate, and whether the output contract is sufficient.

Principle 3: Extract, Don't Design. Do not design APIs in isolation. Instead, extract API endpoints from validated agent tools. Tools that have been empirically validated through agent interaction encode real usage patterns. An API extracted from such tools reflects actual needs rather than speculative requirements.

Principle 4: Dual Interface. The final architecture serves both AI agents and human users. Rather than treating AI as an implementation detail hidden behind a UI, AFD produces systems with two first-class interfaces: agent tools (accessed via protocols like MCP) for AI-assisted operations, and conventional APIs/UIs for routine human operations.

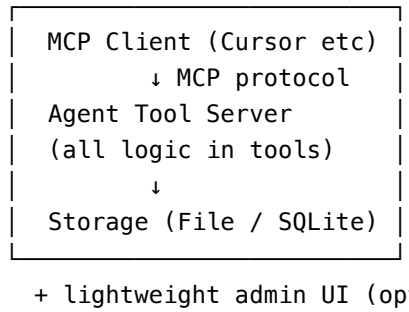
Principle 5: Classify by Cognition. Classify operations by their cognitive requirements. Some operations are deterministic (CRUD, queries, standard management)—these are *API-able*. Others fundamentally require LLM reasoning (semantic analysis, multi-perspective evaluation, natural language pipeline generation)—these are *Agent-native*. This classification drives the Phase 2 extraction decision and the final architectural boundary.

Principle 6: Zero-Gap Deployment. Tool implementation is simultaneously deployment. This principle is most fully realized in MCP-based implementations, where defining a tool as a server-side handler immediately makes it accessible to any MCP-compatible client (Cursor, Claude Code, etc.) via STDIO transport, and to remote teams via Streamable HTTP transport. There is no separate "deploy" step between building and validating. The degree of zero-gap varies by framework—MCP achieves it most directly, while other agent frameworks may require additional configuration—but the aspiration holds: minimize the distance between implementing a tool and making it available for agent validation.

3.3 The Four Phases

AFD proceeds through four phases, each with defined exit criteria. The phases are adapted and generalized from the MCP-to-SaaS Development Workflow [20], which was developed through practical experience building KairosChain [11].

Phase 1: Tool-First MVP **Goal:** Validate core functionality by implementing all domain logic as agent tools.

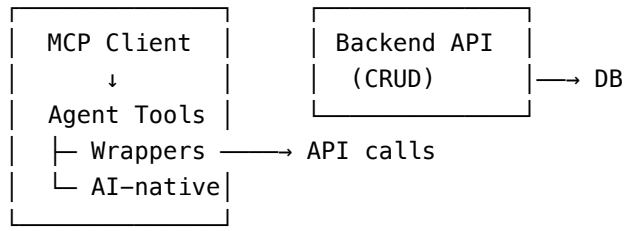


All business logic lives in agent tools, accessed via MCP clients or equivalent agent frameworks. Thanks to Zero-Gap Deployment, each implemented tool is immediately available to MCP-compatible clients via STDIO, and to remote teams via Streamable HTTP—eliminating the gap between implementation and deployment that plagues traditional MVP workflows.

Key activities: - Implement domain operations as agent tools with defined input schemas - Validate with real users via AI agent clients - Add a lightweight admin UI for basic system visibility - Identify which tools are pure CRUD vs. AI-dependent

Exit criteria: - Core operations are working and validated by agent interaction - Tool boundaries are well understood - CRUD vs. AI-dependent classification is emerging

Phase 2: API Extraction **Goal:** Separate deterministic business logic into a backend API. Agent tools that wrap CRUD operations become thin API wrappers; tools that inherently require AI reasoning remain as agent-native tools.

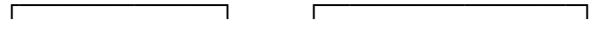


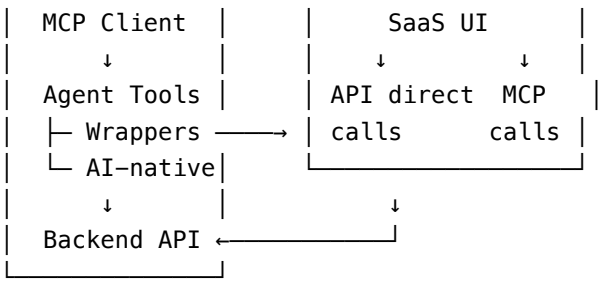
For each existing tool, ask: **"Does this tool fundamentally require LLM reasoning, or is it a deterministic operation?"** This question drives the Cognitive Classification (Section 3.4).

Key activities: - Extract Provider/Manager layers into API service endpoints - Rewrite API-able tools as thin wrappers that call the backend API - Keep Agent-native tools unchanged - For partially API-able tools, split: deterministic paths call API, AI paths remain in tool

Exit criteria: - Backend API serves all deterministic operations - Agent tools for CRUD are now API wrappers - Agent-native tools remain functional - Both MCP clients and API clients can perform deterministic operations

Phase 3: Dual Interface **Goal:** Add a user-facing UI that leverages both the backend API (for routine operations) and agent tool calls (for AI-assisted operations).

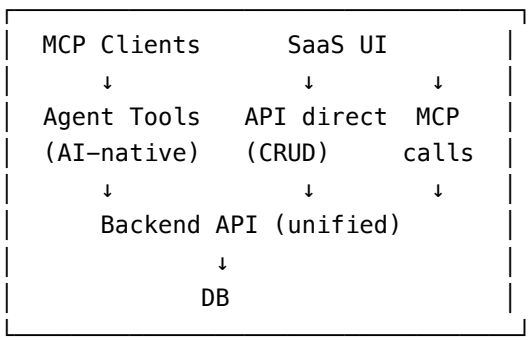




The UI implements two interaction patterns: **Pattern A (API Direct)** for routine CRUD operations with no AI involvement, and **Pattern B (MCP Tool Calls)** for operations where AI reasoning is essential. This dual-interface architecture is a direct consequence of the Classify by Cognition principle.

Exit criteria: - UI is functional for both routine and AI-assisted operations - The boundary between API-direct and MCP-call is clear in the UX

Phase 4: Integration and Optimization **Goal:** Unify the architecture, optimize performance, and establish operational boundaries.



Key activities: Optimize performance, consolidate authentication, add monitoring, document architecture, consider scaling strategies.

3.4 The Cognitive Classification Pattern

A central architectural decision in AFD is the classification of operations by their cognitive requirements. This classification serves a role analogous to the distinction between value objects and entities in DDD, or the separation of commands and queries in CQRS—it is a design vocabulary specific to AFD that drives architectural decisions.

API-able operations are deterministic: given the same inputs, they produce the same outputs regardless of context. Examples include CRUD operations, status queries, resource listings, and configuration management. These operations can be fully expressed as REST endpoints.

Partially API-able operations have a deterministic core with optional AI-enhanced modes. For example, a knowledge update operation might be a simple CRUD write (API-able) but could also accept natural language content that requires AI processing (Agent-native). The recommended pattern is to split: deterministic paths call the API, while AI paths remain in the agent tool.

Agent-native operations fundamentally require LLM reasoning and cannot be meaningfully reduced to deterministic API calls. Examples include multi-perspective analysis, semantic search with contextual reasoning,

natural language pipeline generation, and skill evolution proposals. Forcing these into REST conventions creates awkward asynchronous polling patterns and loses the conversational context that makes them valuable.

This three-tier classification emerges naturally from Phase 1 tool validation and directly informs the Phase 2 extraction boundary.

3.5 Anti-Patterns

AFD defines several anti-patterns that practitioners should avoid:

Big Bang API Design. Designing the entire backend API upfront before validating with agent tools. APIs designed in isolation often mismatch real usage patterns. Let agent tool usage inform API design.

Force Everything into API. Trying to make every agent tool into an API endpoint, including Agent-native ones. AI-native operations do not fit REST conventions well. Maintain the Agent-native / API-able distinction.

Premature SaaS UI. Building the SaaS UI before the API is stable. API changes cascade into UI rewrites. Phase 2 must reach exit criteria before Phase 3 begins.

Human-First Fallacy. Starting design from the human UI and adding agent capabilities as an afterthought. This produces systems where AI feels bolted on rather than integral—the opposite of the AFD philosophy.

Agent as Afterthought. Retrofitting agent tools onto an existing API without rethinking the interface from the agent's perspective. Existing API designs reflect human UI needs, not agent interaction patterns.

Tool Sprawl. Creating large numbers of fine-grained tools without considering composition and granularity. Agent tools should represent meaningful domain operations, not individual database queries.

---

4. Case Study: KairosChain

4.1 Overview

KairosChain [11] is a meta ledger that records the evolution of AI skills on a private blockchain, implemented as a Model Context Protocol (MCP) server. It combines Pure Agent Skills—executable skill definitions using Ruby DSL and Abstract Syntax Trees—with the Minimum-Nomic principle [21], where rules can be amended but the amendment history cannot be erased. KairosChain's development followed the AFD pattern before AFD was formally articulated, making it a natural case study for retrospective analysis.

4.2 Phase 1 in Practice: Zero-Gap Deployment

KairosChain implements over 20 MCP tools as `BaseTool` subclasses, each with a defined `name`, `description`, `input_schema`, and `call` method. These tools were the first—and initially the only—interface to the system. Users interacted with KairosChain exclusively through MCP clients (Cursor, Claude Code).

The Zero-Gap Deployment principle was demonstrated concretely: each new tool, upon implementation, was immediately accessible to MCP clients via STDIO transport. When Streamable HTTP transport was added (via `POST /mcp`), the same tools became accessible to remote team members with no additional configuration. The blockchain layer simultaneously recorded all tool invocations, creating an audit trail that accumulated evidence for future API extraction decisions.

This eliminated the traditional MVP deployment gap: there was no separate "staging" or "demo" environment to configure. The development environment *was* the deployment environment, accessed through the same MCP protocol that production users would employ.

4.3 Cognitive Classification Applied

Applying the Cognitive Classification Pattern to KairosChain's tools reveals a clear three-tier structure:

API-able tools (deterministic CRUD): - `knowledge_update` (create/delete knowledge entries) - `chain_status` (query blockchain state) - `chain_history` (retrieve transaction history) - `token_manage` (token management operations) - `resource_list` / `resource_read` (resource browsing)

Partially API-able tools: - `skills_evolve` — the `apply` and `reset` modes are deterministic (API-able), but the `propose` mode requires LLM analysis of existing skills (Agent-native) - `context_save` — saving context is CRUD (API-able), but generating context summaries requires AI (Agent-native)

Agent-native tools: - `skills_audit` — invokes a Persona Assembly where multiple AI personas evaluate skills from different perspectives - `skills_promote` — requires multi-perspective analysis to determine whether knowledge should be elevated in the layer hierarchy - Semantic search with contextual reasoning

This classification directly maps to the Phase 2 extraction boundary: API-able tools become thin wrappers around REST endpoints; Agent-native tools remain as MCP tools.

4.4 Extractable Architecture

KairosChain's existing codebase already contains the architectural seams that AFD Phase 2 would exploit:

Existing Abstraction	AFD Phase 2 Role
`BaseTool` interface	Tool implementations swappable without affecting clients
`Storage::Backend` (FileBackend, SqliteBackend)	Add `ApiBackend` for API integration
Provider/Manager layer (`KnowledgeProvider`, `ContextManager`, `Chain`)	Directly extractable as API service layer
L0 skill-tool DSL (`execute` block)	Repoint to API calls by updating DSL definition
`ToolRegistry` dynamic registration	Swap tool implementations at runtime

These seams were not designed with AFD in mind—they emerged naturally from the tool-first approach. This suggests that AFD's bottom-up process inherently produces architectures amenable to progressive extraction.

4.5 GenomicsChain Application

The AFD methodology extends to GenomicsChain, a planned decentralized genomic data analysis platform:

- **Phase 1:** MCP tools for NFT minting, pipeline execution, dataset management—immediately usable by researchers via Cursor/Claude Code
- **Phase 2:** REST API for dataset CRUD, pipeline status, NFT management; MCP tools remain for AI-guided pipeline selection, semantic data search
- **Phase 3:** Web UI for researchers (browse datasets, run pipelines, manage NFTs); AI features for natural language pipeline configuration and automated quality control

- **Phase 4:** Production platform with unified authentication, monitoring, and scaling

---

5. Discussion

5.1 Relationship to Existing Methodologies

AFD does not oppose existing methodologies; it complements them. The relationship is best understood through the lens of **direction**:

- **Brownfield systems** (existing codebase with APIs): Use the API-to-MCP pattern [18, 19] to map existing endpoints to agent tools (top-down)
- **Greenfield systems** (new development): Use AFD to build agent tools first, then extract APIs and UIs (bottom-up)
- **Within AFD Phase 1**: TDD remains valuable for testing individual tool implementations. SDD can inform tool specification. ADD's editorial workflow applies to reviewing agent-generated code.

AFD is thus a *meta-methodology* for the overall development lifecycle, within which other methodologies operate at specific phases.

5.2 Generalizability Beyond MCP

While this paper uses MCP as the concrete agent protocol, AFD's principles are not MCP-specific. The methodology applies to any system where AI agents invoke structured tool interfaces:

- **OpenAI Function Calling**: Tools defined as function schemas with `name`, `description`, and `parameters`
- **LangChain / LangGraph Tools**: Python-decorated functions with type annotations and docstrings
- **Semantic Kernel**: Plugins with annotated kernel functions
- **Custom agent frameworks**: Any system implementing tool invocation with structured input/output

The essential requirement is that the agent interface be *self-describing*—tools must declare their name, purpose, input schema, and output contract in a way that enables AI agents to select and invoke them. This self-describing property is what makes agent tools superior to conventional APIs as a design starting point: they force explicitness about semantics that APIs often leave implicit.

5.3 Auditable Evolution as AFD Infrastructure

KairosChain's blockchain-based audit trail [11] provides infrastructure that supports AFD across all phases:

- **Phase 1 evidence**: Tool invocation histories, recorded on-chain, provide empirical evidence of which tools are used, how often, and in what patterns. This evidence informs Phase 2 API extraction decisions.
- **Phase 2 verification**: When tools are refactored into API wrappers, the audit trail enables before/after comparison to verify that behavior is preserved.
- **Continuous validation**: The Minimum-Nomic principle [21] ensures that the system can evolve (tools can be added, modified, or decomposed) while maintaining a complete record of how and why changes were made.

This suggests that auditable evolution infrastructure—while not required for AFD—significantly strengthens its practice by providing objective evidence for extraction decisions.

5.4 Limitations and Applicability Conditions

AFD is most effective when: - The domain involves both deterministic operations and AI-assisted operations - The system will eventually serve both AI agents and human users - Rapid validation through agent interaction is valuable - The development team has experience with agent tool design

AFD is less appropriate when: - The system is a pure CRUD application with no AI component - Agent tool design skills are unavailable on the team - The target users will never interact through AI agents - Regulatory requirements mandate traditional software development processes

Additionally, AFD's reliance on agent interaction for validation introduces a dependency on LLM quality: if the AI agent cannot effectively exercise the tools, validation quality suffers. This limitation is expected to diminish as LLM capabilities continue to improve.

A practical concern arises at scale: when the number of tools exceeds LLM context window limits, agents may struggle to select appropriate tools. Current MCP implementations list all available tools in the system prompt, creating a ceiling of roughly 50–100 tools before performance degrades. Mitigation strategies include tool namespacing, hierarchical tool discovery, and dynamic tool loading—areas where MCP's `ToolRegistry` pattern provides a foundation but further research is needed.

Furthermore, this paper's case study (KairosChain) has only completed Phase 1, with Phases 2–4 described prospectively based on architectural analysis rather than empirical execution. A full validation of AFD's four-phase lifecycle awaits future work.

5.5 The Dual Direction Thesis

An important observation is that the software industry simultaneously needs *both* directions:

```
Brownfield:  Existing API  →  Agent Tool (top-down mapping)
Greenfield:  Agent Tool    →  API → UI  (bottom-up extraction, AFD)
```

Current industry focus is heavily weighted toward the top-down direction—converting existing APIs to MCP tools [18, 19]. AFD provides the complementary bottom-up direction. A mature AI development ecosystem requires both, just as software engineering requires both top-down design and bottom-up implementation strategies.

5.6 Future Metrics

AFD would benefit from quantitative metrics to measure its effectiveness. We propose the following candidates for future validation:

- **Tool Coverage**: The proportion of domain operations expressible as agent tools (analogous to test coverage in TDD)
- **Extraction Rate**: The proportion of Phase 1 tools successfully extracted into API endpoints in Phase 2
- **Phase Transition Lead Time**: The calendar time required to move between phases, measured from exit criteria achievement
- **Dual-Interface Parity**: The degree to which agent and human interfaces provide equivalent functional access
- **Cognitive Classification Stability**: How often the API-able / Agent-native classification of a tool changes after initial assignment

Empirical validation of these metrics across multiple AFD projects would strengthen the methodology's evidence base.

---

6. Conclusion

Agent-first Driven Development proposes a fundamental reorientation of the software development process for the AI agent era. Where TDD declared "write the test first" and SDD declared "write the spec first," AFD declares: **"build the agent tool first."**

By treating AI agents as the first consumer rather than the last implementer, AFD produces architectures that naturally support dual interfaces (agent and human), enables empirically-grounded API extraction, and introduces the Cognitive Classification Pattern as a novel architectural vocabulary for separating deterministic from AI-native operations.

The Zero-Gap Deployment principle—where tool implementation is simultaneously deployment—eliminates the traditional gap between building and validating, enabling rapid iteration cycles that rival vibe coding's speed while maintaining the structural discipline of SDD.

KairosChain demonstrates that AFD's bottom-up approach naturally produces architectures with clean extraction seams, and that auditable evolution infrastructure can provide objective evidence for extraction decisions across phases.

Future work should validate AFD through complete Phase 1–4 lifecycle execution across multiple projects, develop quantitative metrics (tool coverage, extraction rate, phase transition lead time), and address scalability challenges when tool counts exceed LLM context window limits.

As AI agents become ubiquitous development partners and end-user interfaces, the question is no longer whether to design for agents, but *when* in the development process to do so. AFD's answer is clear: from the very beginning.

References

[1] K. Beck, *Test-Driven Development: By Example*. Boston: Addison-Wesley, 2002. ISBN: 978-0321146533

[2] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Boston: Addison-Wesley, 2003. ISBN: 978-0321125217

[3] A. Karpathy, "Vibe coding," X post, February 2025. [Online]. Available: https://x.com/karpathy/status/188619218480814

[4] Y. Ge et al., "A Survey of Vibe Coding with Large Language Models," arXiv preprint arXiv:2510.12399, 2025.

[5] "The Vibe Coding Trap: Why Unstructured AI Code Generation Fails," Strategy Radar AI, 2025. [Online]. Available: https://www.strategyradar.ai/blog/vibe-coding-trap

[6] M. Fowler, "Understanding Spec-Driven-Development: Kiro, spec-kit, and Tessl," MartinFowler.com, 2025. [Online]. Available: https://martinfowler.com/articles/exploring-gen-ai/sdd-3-tools.html

[7] GitHub Blog, "Spec-driven development with AI: Get started with a new open source toolkit," 2025. [Online]. Available: https://github.blog/ai-and-ml/generative-ai/spec-driven-development-with-ai-get-started-with-a-new-open-source-toolkit

[8] R. H. Jansen, "Agent Driven Development (ADD): The Next Paradigm Shift in Software Engineering," DEV Community, July 2025. [Online]. Available: https://dev.to/remojansen/agent-driven-development-add-the-next-paradigm-shift-in-software-engineering-1jfg

[9] A. E. Hassan, G. A. Oliva, D. Lin, B. Chen, and Z. M. Jiang, "Towards AI-Native Software Engineering (SE 3.0): A Vision and a Challenge Roadmap," arXiv preprint arXiv:2410.06107, 2024.

[10] K. Han, S. Maddikayala, T. Knappe, O. Patel, A. Liao, and A. B. Farimani, "TDFlow: Agentic Workflows for Test Driven Development," arXiv preprint arXiv:2510.23761, 2025.

[11] M. Hatakeyama, "KairosChain: Pure Agent Skills with Self-Amendment for Auditable AI Evolution," Technical Note, Zenodo, January 2026. DOI: 10.5281/zenodo.18289164

[12] "Why Vibe Coding Fails – and How Signal Coding Fixes It," SEP, 2025. [Online]. Available: https://sep.com/blog/vibe-coding-evolved/

[13] Augment Code, "Spec-Driven AI Code Generation with Multi-Agent Systems," 2025. [Online]. Available: https://www.augmentcode.com/guides/spec-driven-ai-code-generation-with-multi-agent-systems

[14] Google, "DORA 2025 State of AI-assisted Software Development Report," 2025. [Online]. Available: https://research.google/pubs/dora-2025-state-of-ai-assisted-software-development-report/

[15] Andreessen Horowitz, "Emerging Developer Patterns for the AI Era," a16z.com, 2025. [Online]. Available: https://a16z.com/nine-emerging-developer-patterns-for-the-ai-era

[16] S. Naqvi, M. Baqar, and N. A. Mohammad, "The Rise of Agentic Testing: Multi-Agent Systems for Robust Software Quality Assurance," arXiv preprint arXiv:2601.02454, 2026.

[17] Tweag, "Test-Driven Development," Agentic Coding Handbook, 2025. [Online]. Available: https://tweag.github.io/agentic-coding-handbook/WORKFLOW_TDD/

[18] Scalekit, "How to map an existing API into MCP tool definitions," 2025. [Online]. Available: https://www.scalekit.com/blog/map-api-into-mcp-tool-definitions

[19] S. Shines, "Mapping an existing API to MCP tools," DEV Community, 2025. [Online]. Available: https://dev.to/saif_shines/mapping-an-existing-api-to-mcp-tools-57l1

[20] M. Hatakeyama, "MCP-to-SaaS Development Workflow," KairosChain L1 Knowledge, 2026. [Online]. Available: https://github.com/masaomi/KairosChain_2026/blob/main/KairosChain_mcp_server/knowledge/mcp_to_saas_de

[21] M. Hatakeyama and T. Hashimoto, "Minimum Nomic: A Tool for Studying Rule Dynamics," *Artificial Life and Robotics*, vol. 13, no. 2, pp. 500–503, 2009. DOI: 10.1007/s10015-008-0605-6

DOI and Citation

This paper is available on Zenodo.

DOI: 10.5281/zenodo.18649254

Recommended citation:

Hatakeyama, M. (2026). Agent-first Driven Development: A Bottom-Up Methodology for Building Software in the Age of AI Agents. Preprint. Zenodo. https://doi.org/10.5281/zenodo.18649254

Version 1.1 — 15 February 2026 (reference corrections)