Emergence: A Programming Paradigm for Constraint-Shaped Agency

Jeremy McEntire
jeremy@wander.com

February 2026

Abstract

The programmer does not write the program. The programmer writes the conditions under which the program emerges. We propose *Emergence* as a programming paradigm in which the fundamental unit of computation is an autonomous agent that interprets messages within a constraint architecture—schemas, thresholds, scoring functions, and topological rules—and the system's behavior arises from the interaction of agents operating within those constraints. This paradigm recovers Alan Kay's original vision of object-oriented programming, which required a computational substrate with contextual understanding that did not exist until large language model agents. We ground the paradigm in convergent results from cybernetics, modular design theory, organizational theory, and complex systems. We present a working stigmergic mesh as an existence proof and address the paradigm's real limitations—latency, cost, non-determinism, and the genuinely new discipline of designing constraint spaces—with equal honesty.

1 Introduction

Every programming paradigm answers one question: *where does the intelligence live?* In procedural programming, it lives in the sequence of instructions. In functional programming, it lives in the composition of transformations. In object-oriented programming—as practiced, not as conceived—it lives in the class hierarchy.

In each case, the programmer encodes the intelligence directly. The system does what it is told. This works when the problem is fully specifiable in advance. It fails when the problem space exceeds the programmer's capacity to enumerate its states—which is to say, it fails for every interesting problem in every sufficiently complex domain.

The arrival of large language model (LLM) agents creates a genuinely new possibility. For the first time, the computational unit itself possesses contextual understanding. An LLM agent assessing a code review comment does not pattern-match against a list of phrases; it understands what code review is *for*, recognizes when a comment is substantive versus ceremonial, and distinguishes a genuine technical concern from a territorial objection. This capability is not incremental improvement. It is a qualitative change in what a computational unit *can be*.

We call the resulting paradigm **Emergence**. The taxonomy is clean:

> *Procedural programming: you write the steps. Object-oriented programming: you write the entities. Functional programming: you write the transformations. Emergence: you write the constraints, and the behavior writes itself.*

The program is not the behavior. The program is the physics, the constraints, the container, the field equations. The behavior is the emergent property of the system the programmer designed—not the system itself. This shift implies not merely a new tool but a new discipline: the skill of designing constraint spaces that reliably produce

useful behavior is neither programming as currently practiced nor management as currently understood, and its development is as much a part of the paradigm's contribution as its technical architecture (Section 10.3).

2 The Kay Recovery

Alan Kay has spent decades clarifying what he meant—and what he did not mean—by "object-oriented programming." The clarifications have been largely ignored.

In a 2003 email, Kay wrote: "I'm sorry that I long ago coined the term 'objects' for this topic because it gets many people to focus on the lesser idea. The big idea is messaging" [Kay, 2003]. In his Smalltalk work and subsequent reflections, Kay consistently described three properties:

1. Each object is an autonomous entity—"a recursion of the whole computer"—with its own state, behavior, and interpretation of incoming messages.
2. Communication happens exclusively through messages, and the *receiver* decides what a message means.
3. The system's behavior emerges from the interaction of these autonomous message-interpreting entities, not from a central controller.

What the industry built was different. Method dispatch replaced message interpretation: a message became a function call, resolved at compile time or through a vtable, with the caller knowing exactly what would happen. Class hierarchies replaced autonomy: objects became instances of types, their behavior determined by inheritance chains. Design patterns replaced emergence: the Gang of Four catalog [Gamma et al., 1994] codified strategies for managing complexity that the paradigm was supposed to dissolve.

The misimplementation was not stupidity. It was a technological constraint. In 1972—or 1995, or 2015—there was no computational substrate that could *interpret* a message. A function can parse one. A pattern matcher can classify one. A rule engine can apply conditions to one. But none of these *understands* one. The receiver cannot decide what a message means if the receiver has no capacity for meaning.

Kay's vision required a computational unit with genuine contextual understanding. It required exactly what an LLM agent provides.

3 The Agent as Computational Primitive

An LLM agent is not a better function. It is a different kind of computational entity. The distinction matters because it is the distinction Kay was trying to draw.

A function *computes*: given inputs, it produces outputs according to a fixed algorithm. An agent *assesses*: given a situation, it produces a judgment shaped by its understanding and context. A function that detects scope creep by counting changed lines is doing something fundamentally different from an agent that reads a pull request and recognizes that what was filed as a "bug fix" is actually a feature rewrite touching authentication, database schema, and three API contracts simultaneously. The function applies a rule. The agent understands a situation.

This distinction enables three properties previously unavailable:

Genuine message interpretation. When an agent receives a signal—a pull request, a customer ticket, a deployment log—it does not merely parse the text. It interprets meaning in context. A large change touching authentication logic receives different attention than an identically sized change adding stylesheets, not because of a rule about filenames, but because the agent understands what authentication code is and why changes to it carry risk.

Judgment under ambiguity. Traditional computation requires the programmer to anticipate every case. An agent exercises judgment in situations the programmer never considered, because its training encompasses the breadth of human knowledge about how to reason in context. This is the same capability that allows a

new employee to handle situations not covered by the employee handbook.

Compositional emergence. When multiple agents, each interpreting messages within their own context, interact through a shared environment, the system-level behavior is not the sum of the agents' behaviors. Patterns, correlations, and structures arise that no individual agent was designed to detect. This is the property that Kay described and that class-hierarchy OOP could never deliver.

4 The Constraint Architecture

If the agents provide the flexibility, what provides the reliability? This is the central engineering question of the paradigm, and its answer is the paradigm's central insight: **the program is the boundary, not the behavior.**

In Emergence, the programmer designs the constraint architecture within which agents operate: *rigid at the interfaces, flexible in the interiors.*

4.1 Rigid Boundaries

The boundaries are non-negotiable structural elements that ensure coherence:

Schemas. The data structures agents consume and produce are strictly typed. A signal has a source, timestamp, content, and metadata. An assessment has a confidence score, a set of claims, and a provenance chain. The agent decides what to *put* in these fields; the schema decides what fields *exist*.

Thresholds. Numerical parameters that gate behavior without prescribing it. A vigilance parameter determines how similar a signal must be to existing knowledge before an agent accepts it. An energy budget determines how many signals an agent can process before it must yield. The agent exercises judgment *within* these bounds; the bounds themselves are fixed.

Scoring functions. Mathematical functions that evaluate outputs without understanding them. A familiarity score combines lexical overlap, semantic similarity, structural alignment, and temporal recency into a single number. The agent does not know it is being scored. The scoring function does not know what the agent is thinking. The decoupling is the point: the meta-system shapes selection pressure without micromanaging cognition.

Topological rules. Rules governing how agents connect, communicate, and compete. Which agents see which signals. How routing priority is determined. When agents fork, merge, or decay. These rules define the *structure* of interaction without constraining the *content*.

4.2 Flexible Interiors

Within the boundaries, agents operate with genuine autonomy. They interpret signals, form assessments, develop specializations, and generate insights without procedural instruction. The programmer does not specify *how* an agent should assess whether a team's communication patterns are degrading. The programmer specifies *that* communication health is a property to assess, provides a schema for reporting it, and trusts the agent's contextual understanding to do the rest.

This is the inversion that defines the paradigm. In traditional programming, the interior is rigid (algorithms, control flow, explicit logic) and the boundaries are flexible (APIs change, schemas evolve). In Emergence, the boundaries are rigid and the interiors are flexible. The programmer's job shifts from *writing behavior* to *shaping the space in which behavior occurs.*

4.3 Recursive Constraint Interaction

The constraint architecture is not a thermostat with one feedback loop. It has many degrees of freedom (one per agent per context) and many feedback loops (scoring, routing, lifecycle, consensus) that *interact*. The vigilance threshold affects which signals an agent accepts, which affects its specialization, which affects the topology, which affects which signals it receives, which affects whether the vigilance threshold should change.

This recursive interaction is the mechanism of emergence. The constraint architecture cre-

ates conditions for self-organization; the self-organization produces capabilities that no individual constraint was designed to produce.

4.4 Well-Formedness

If the paradigm is to be engineering rather than aspiration, constraint architectures must have well-formedness conditions. A constraint architecture is well-formed when it satisfies three properties:

1. **Completeness**: every agent output passes through at least one scoring function. No agent can produce assessments that bypass evaluation. This is the analog of type safety: the system guarantees that all behavior is bounded, even if the specific behavior is not predicted.
2. **Liveness**: the lifecycle rules guarantee that the system cannot freeze. There must exist conditions under which new agents are created (fork), idle agents are removed (decay), and redundant agents are consolidated (merge). A system that can only grow or only shrink is not well-formed.
3. **Dissipation**: the energy budget is finite and monotonically decreasing absent external input. No agent can process signals indefinitely. This prevents monopoly (one agent capturing all signals) and guarantees that the system must continuously earn its computational expenditure.

These conditions do not specify what the system will do. They specify what the system *cannot* do: produce unscored outputs, freeze into a fixed topology, or consume resources without bound. The formal development of these conditions into a full design calculus—constraint typing, compositional well-formedness, convergence guarantees—is future work. But the conditions themselves are already enforced in the existence proof and are sufficient to distinguish a well-formed constraint architecture from an ad hoc collection of agents.

5 Theoretical Convergence

The Emergence paradigm converges with results from four independent traditions. The convergence is not coincidental—each tradition discovered, through its own methods, the same structural truth about where rigidity and flexibility must live.

5.1 Ashby's Law of Requisite Variety

Ashby's Law [Ashby, 1956] states that a regulator must have at least as much variety as the system it regulates. But Ashby's formulation is more specific than the popular summary suggests. The law constrains the *channel capacity* between regulator and regulated—the bandwidth through which the regulator observes and acts upon the system. In traditional programming, the channel capacity is determined by the programmer's ability to enumerate states. Every `if` branch, every `case` statement, every validation rule is an explicit expansion of the channel. This works until the state space exceeds enumerability.

Emergence achieves requisite variety through a different channel. The agents' contextual understanding *already encompasses* the variety of the problem space. The constraint architecture does not need to enumerate states; it needs to *bound the agents that are already navigating them.* The channel capacity comes from the agents, not the program. This is why the paradigm applies precisely where traditional programming fails: in domains where the state space exceeds any programmer's capacity to enumerate—organizations, ecosystems, markets, complex social systems.

5.2 Beer's Viable System Model

Stafford Beer's Viable System Model (VSM) [Beer, 1972] identifies the 3–4 homeostat as the central tension in any viable system: System 3 (internal stability) must be dynamically balanced against System 4 (environmental intelligence). Beer's specific contribution is not the observation that stability and adaptability must coexist—that is platitude. It is that *System 4 must be a genuine model of the environment, not a summary of internal reports.* Organizations fail, Beer argues, because System 4 degrades into a mirror of System 3: the "intelligence" function reports what operations already believe rather than what the environment actually contains.

This is precisely the failure mode that Emergence is designed to avoid. The agents in the flexible interior are not summarizing internal state; they are interpreting environmental signals with contextual understanding independent of the system's prior beliefs. The lifecycle rules—fork, decay, merge—are the homeostat, and the rigid boundaries are System 3. But the specific contribution of Emergence to Beer's framework is a System 4 that cannot be captured by System 3, because the agents' understanding comes from outside the system's own training data.

5.3 Baldwin and Clark's Design Rules

Baldwin and Clark [Baldwin and Clark, 2000] demonstrated that modular systems outperform integral systems when design rules—the interfaces between modules—are well-specified. Their framework goes further than the popular summary "modularity is good." They provide a formal option-pricing model showing that modularity creates *option value*: each module can be independently varied, creating a combinatorial space of possible designs that can be explored in parallel. The value of this optionality grows with the number of modules and the uncertainty of the design space.

In Emergence, agents are modules and constraints are design rules. Each agent independently develops specialization within the rigid interface definitions. The system explores the space of possible specializations in parallel—an agent that absorbs deployment signals develops deployment expertise while another, absorbing customer tickets, develops support expertise. Neither was programmed to specialize. The option value is not financial but epistemic: the system discovers what specializations are useful rather than requiring the programmer to predict them.

5.4 Kauffman's Edge of Chaos

Kauffman's work on Boolean networks [Kauffman, 1993] identified the regime between order and chaos where adaptive computation occurs. In frozen networks, perturbations die out and the system cannot adapt. In chaotic networks, perturbations propagate everywhere and the system cannot maintain structure. At the edge of chaos, perturbations propagate just far enough to enable adaptation without dissolution.

Kauffman showed that the position in this spectrum is controlled by two parameters: the connectivity of the network (how many other elements each element influences) and the bias of the Boolean functions (how likely each element is to be in a given state). The Emergence constraint architecture controls both: topological rules govern connectivity (which agents see which signals), and thresholds govern bias (how likely an agent is to accept, reject, or ignore). The lifecycle rules—fork, merge, decay—continuously adjust these parameters, maintaining the system at the edge without requiring the programmer to calculate the critical regime directly.

5.5 The Convergence

Four traditions, developed independently over six decades, arrived at the same prescription: rigid constraints at the boundaries, flexible behavior in the interiors, with feedback loops that maintain the balance. Ashby specified the information-theoretic requirement. Beer described the organizational architecture. Baldwin and Clark proved the economic value. Kauffman identified the computational regime. Even Postel's Law—"be conservative in what you send, liberal in what you accept" [Postel, 1980]—is the same insight applied to protocol design: rigid outputs, flexible inputs. Emergence generalizes this principle from protocols to entire systems: rigid boundaries, flexible agents.

The convergence across independent formalisms is the strongest evidence that the paradigm captures something real—not a framework imposed on recalcitrant material, but a pattern the material itself exhibits.

6 Related Work

The Emergence paradigm did not arise in a vacuum. Several research traditions have explored adjacent territory, and the differences are as instructive as the similarities.

BDI Agents. The Belief-Desire-Intention architecture [Rao and Georgeff, 1995] gave agents internal mental states—beliefs about the world, desires about outcomes, intentions about plans. This was a genuine advance over reactive agents, but the beliefs, desires, and intentions were still programmer-specified data structures. A BDI agent does not *understand* its beliefs; it *stores* them. The architecture provided the right structure but lacked the substrate: a computational unit that could actually form beliefs from evidence rather than retrieving them from a database.

Swarm Intelligence. Ant colony optimization, particle swarm optimization, and related approaches [Bonabeau et al., 1999] demonstrated that simple agents following simple rules can produce complex collective behavior. This is genuine emergence, and it inspired the stigmergic coordination in our existence proof. But swarm agents are *simple*—they follow fixed behavioral rules without contextual understanding. Emergence extends the principle to agents with genuine interpretive capacity, enabling coordination in domains (language, judgment, assessment) where simple rules are insufficient.

Autonomic Computing. IBM's autonomic computing initiative [Kephart and Chess, 2003] proposed self-managing systems organized around the MAPE-K loop (Monitor, Analyze, Plan, Execute, Knowledge). The architecture anticipated many elements of Emergence: self-organization, feedback loops, adaptive behavior. But the MAPE-K loop assumes that the Analyze and Plan phases can be implemented as traditional algorithms. In domains where analysis requires judgment rather than computation, the loop stalls—the system can monitor, but it cannot *understand* what it monitors.

Self-Adaptive Systems. The SEAMS community has explored systems that modify their own behavior at runtime, typically through models@runtime and feedback control [Cheng et al., 2009]. These systems are the closest precursor to Emergence, but they rely on explicit models of the adaptation space—the programmer must specify what can change and how. Emergence removes this requirement: the agents' contextual understanding replaces the explicit adaptation model.

In each case, the limitation was the same: the agents lacked understanding. The architectures were right. The substrate was wrong. Emergence inherits the architectural insights of these traditions and pairs them with the first computational substrate capable of delivering on their promise.

7 The Chronicler Principle

The paradigm is easiest to understand through analogy.

Consider a system designed to write a novel. The traditional approach programs a plot generator: act structure, character arcs, conflict escalation, prose templates. The programmer encodes the intelligence of storytelling into explicit rules.

The Emergence approach builds a *world*: a physics model that enforces conservation laws, an economy model that enforces scarcity, a character model that enforces psychological consistency, a geography model that enforces spatial constraints. Each model is an agent bound to a context, interpreting events within its domain. No model knows about stories. No model has a plot. But when a character model reports that a character's motivation has shifted, and the economy model reports that a resource has become scarce, and the geography model reports that two factions now share a border—a story emerges from the interaction.

A chronicler agent observes the interactions, recognizes narrative structure in the convergence of model outputs, and produces prose. The chronicler does not invent events. It *witnesses* them. The story is not written; it is *discovered.*

This is not metaphor. It is the literal architecture. Replace "physics model" with "schema validator" and "character model" with "domain expert agent" and "chronicler" with "consensus layer," and you have the architecture of any Emergence system. The programmer builds the world. The agents inhabit it. The behavior is what hap-

pens when the world's rules interact.

8 Existence Proof: A Stigmergic Mesh

Theory without implementation is speculation. We have built a working system that instantiates the Emergence paradigm: a stigmergic mesh for ambient structure discovery in organizations [McEntire, 2026].

The system ingests work artifacts—pull requests, issue tickets, deployment logs, chat messages—as signals. No one formulates a query. No one specifies what to look for. The system processes the ambient exhaust of organizational work and surfaces structural patterns that no one asked about.

8.1 Architecture

The mesh consists of worker agents, each bound to a context representing a region of specialization. Signals arrive and are routed via breadth-first search through the worker topology. The first worker whose familiarity score exceeds a vigilance threshold accepts the signal; the remaining workers never see it. Accepted signals update the worker's context through one of three learning modes: full storage (indexed), compressed summary, or lossy weight shift. Workers that accept too many signals fork. Workers that accept too few decay. Redundant workers merge. The topology self-organizes.

The rigid boundaries are strict: immutable typed signals, assessment schemas with required confidence scores, a five-component familiarity function, exponential energy decay with activity-based recharge. The flexible interiors are genuine: when an agent assesses a pull request, it brings the full weight of its contextual understanding to bear on what the change means, what risks it carries, and what patterns it participates in.

8.2 What Emerged

During live deployment against a real engineering organization (dozens of repositories, multiple chat channels, several project tracking teams), the system produced detections that no individual component was designed to generate:

- A policy intervention flagged a pull request that modified security-sensitive code buried inside a large feature branch—not because of a rule about filenames, but because the agent recognized that a sensitive surface was being modified incidentally, inside a change scoped as something else, and that similar modifications were accumulating faster than the budgeted rate.
- A churn detector surfaced that a repository had exceeded its weekly diff budget, flagging not the volume itself but the *pattern*: a single large PR pushing cumulative churn past a threshold, suggesting scope creep in what had been filed as a contained change.
- Workers self-organized into specializations that no one configured. Over successive runs, individual workers developed affinities for infrastructure concerns, product-facing behavior, and cross-team coordination—not because they were assigned these domains, but because the signals they accepted reinforced familiarity in those directions. Workers that received no signals decayed and were pruned. Workers that received too many forked, creating new specialists.
- Cross-source correlations emerged. The same organizational pattern—a team under sustained pressure—manifested differently across version control (shorter review comments, faster merges), project tracking (tickets reclassified from feature work to defects), and chat (reduced channel activity). No individual source showed the pattern clearly. The convergence across sources—detected by independently specialized agents whose outputs were correlated by attention mechanisms—surfaced it.

The pattern across these detections is consistent: *temporal accumulation invisible to any single actor.* Each individual action—a pull request, a code review, a ticket reclassification—is visible to its author and unremarkable in isolation. The trajectory across actions, across authors, across sources, is visible to no one. That is the mesh's niche: it detects the structural patterns that arise

from the interaction of individually reasonable decisions, which is precisely the class of problem that traditional query-based systems cannot address because no one knows to ask.

Equally important is what the system *suppressed.* The majority of ingested signals were filtered as retrieval rather than discovery: factual restatements of what an individual author did, adding no structural insight. The filtering machinery—bot detection, duplicate suppression, retrieval/discovery classification—is not auxiliary to the emergence; it is what makes the emergence legible. Without it, the genuine structural findings would be buried in noise and no one would read any of them.

The system operates with over one thousand passing tests and several automated detectors for early indicators of organizational dysfunction. A full ingestion cycle costs under one dollar and completes in minutes.

9 What Changes

9.1 The Programmer's Role

The programmer becomes an architect of constraint spaces. The skills shift from algorithm design to boundary design: what schemas to enforce, what thresholds to set, what scoring functions to use, what topological rules to apply. The question changes from "what should the system do?" to "what should the system *not* be able to do?"—and then trusting that useful behavior emerges within those limits.

This is not deskilling. Designing constraint architectures that produce useful emergent behavior is at least as difficult as writing explicit algorithms. The difficulty shifts from "how do I make the system do X?" to "how do I create conditions under which X naturally arises?"

9.2 Testing and Verification

Emergence systems are not fully deterministic. You do not get the same novel twice. You do not get the same mesh findings twice. You get *structurally consistent* results—bounded by the same constraints, shaped by the same scoring functions, governed by the same lifecycle rules—but the specific trajectories vary because the agents are flexible.

This is a feature, not a defect—but only if the constraints are designed well enough that every trajectory is acceptable. That is the engineering skill: not writing behavior, but designing the space of acceptable behaviors so thoroughly that whatever emerges within it is useful. This is what makes Emergence engineering rather than hope.

Testing must therefore shift from input-output verification to *property verification*: does the system maintain its invariants? Do the constraints hold? Does the topology self-organize into useful specializations? Do cross-agent correlations converge on genuine patterns rather than noise? The discipline is not less rigorous; it is differently rigorous.

9.3 Failure Modes

A traditional system fails by producing wrong outputs. An Emergence system fails by producing *incoherent* outputs—agents that contradict each other, emergent patterns that are artifacts rather than structure, consensus that converges on noise.

Debugging shifts accordingly. Where traditional debugging traces execution paths, Emergence debugging examines constraint interactions: which boundaries are too tight (suppressing useful behavior), which are too loose (permitting incoherent behavior), which feedback loops are amplifying noise. In the stigmergic mesh, this means monitoring energy landscapes, tracking spectral properties of the agent topology, and examining whether worker specializations are converging or drifting.

9.4 The Configuration Trap Dissolved

A pervasive pattern in traditional software is the *configuration trap*: flexibility implemented as configuration options accumulates until the combinatorial space of configurations becomes the dominant source of complexity.

Emergence dissolves this trap by placing flexibility in the agents rather than in configuration.

The constraints are few and well-defined. The flexibility comes from contextual interpretation, which does not require configuration—it requires understanding, which the agents already possess.

10 Limitations

The Emergence paradigm has real limitations that must be acknowledged honestly. Some are fundamental; some are contingent on the current state of technology.

10.1 Latency

Every agent assessment is an LLM inference call. In the stigmergic mesh, processing a single signal through reflection, evaluation, and assessment takes seconds, not microseconds. A backfill of several dozen signals from a version control system takes several minutes. This is not a network optimization problem; it is a fundamental property of the computational substrate. LLM inference is slow in the same way that human judgment is slow—the richness of contextual interpretation has a temporal cost.

This means Emergence systems operate on a different timescale than traditional software. They are not suitable for real-time control, latency-sensitive APIs, or tight feedback loops. They are suitable for problems where the cost of missing a pattern exceeds the cost of detecting it minutes or hours later—organizational intelligence, risk monitoring, strategic assessment. The paradigm does not replace traditional computing; it occupies a different niche.

10.2 Cost

Agent assessments cost money. Each LLM call consumes tokens, and tokens have a price. The stigmergic mesh spends on the order of one dollar per session processing dozens of signals across multiple sources. At continuous operation, this is a meaningful operational cost. The constraint architecture must therefore include explicit economic boundaries: token budgets, energy decay functions that limit processing, three-tier learning modes that range from full analysis (expensive) to lossy impression (cheap). Cost management is not an operational afterthought; it is a first-class constraint in the architecture.

10.3 The Space Between: A New Discipline

The hardest limitation is human, not technical. Emergence requires a skill that does not yet have a name, a training pipeline, or a professional identity. It is not programming—the practitioner does not write algorithms or control flow. It is not management—the practitioner does not direct agents through conversation or feedback. It is something in between: closer to designing a game's rules than to playing it, closer to landscape architecture than to construction, closer to creating an ecosystem than to building a machine.

The practitioner must think in terms of selection pressures rather than instructions, in terms of fitness landscapes rather than control flow, in terms of what behaviors a constraint *space* permits rather than what a specific agent will *do*. The feedback loops are slower—you cannot step through a debugger; you observe emergent behavior across many signals and many cycles. The diagnostic intuition is ecological rather than mechanical: "the system isn't specializing because the vigilance threshold is too low" is a different kind of reasoning than "the function returns null because the input is empty."

This is arguably the paradigm's most important implication. Every previous paradigm shift changed what programmers *wrote*—structured blocks instead of gotos, objects instead of procedures, pure functions instead of mutable state. Emergence changes what programmers *are*. The practitioner becomes something closer to a constitutional designer—defining the rules under which autonomous entities operate, the invariants that must hold regardless of what those entities do, and the feedback mechanisms that keep the system within its viable envelope. The analogy to governance is not decorative: the questions are the same. What freedoms do the agents need? What constraints prevent pathological outcomes? How do you design accountability without destroying autonomy?

This discipline will develop. Training pipelines will emerge. But in the near term, the paradigm asks practitioners to operate in a genuinely unfamiliar cognitive mode—the space between writing code and herding cats, where the work product is neither an algorithm nor a directive but a set of conditions under which useful behavior reliably arises.

10.4 Non-Determinism

Emergence systems do not produce identical outputs from identical inputs. This requires a precise distinction: the *constraints* are deterministic; the *trajectories* are not. The scoring functions produce the same scores for the same inputs. The energy budgets decay at the same rates. The schemas enforce the same shapes. What varies is the path the agents take through the space those constraints define.

This means the debugging methodology is different but not absent: you debug the constraints, not the behavior. If the system produces pathological output, the question is not "what did agent A do wrong?" but "what constraint failed to prevent this trajectory?" The constraints guarantee what *will not* happen. The agents discover what *will*.

For many engineering contexts, nondeterministic trajectories are disqualifying. The paradigm does not apply where exact reproducibility is required. It applies where *structural consistency* is sufficient: where the question is not "did the system produce output X?" but "did the system produce an output within the acceptable set?" This is a genuine restriction on the paradigm's scope, not a limitation to be engineered around.

10.5 State and Memory

Emergence systems must learn, but LLM agents are stateless between calls. All continuity must be externalized—into context summaries, weight vectors, learned term lists, communication graphs, finding registries. This externalization is lossy. The stigmergic mesh preserves semantic state (what agents have learned) but loses topological state (how many agents exist, their specialization profiles, their neighbor relationships) on restart. The system reconverges, but the reconvergence path may differ from the original. This is the "dice in a cube" problem: the agent can store what it has learned, but not the emergent structure of its relationships with other agents.

11 Objections

11.1 "This is just multi-agent systems."

Multi-agent systems have existed for decades (Section 6). The novelty is not the architecture; it is the nature of the agents. Traditional multi-agent systems use agents that follow rules, match patterns, and execute programmed behaviors. The constraint architecture in such systems must compensate for the agents' lack of understanding. In Emergence, contextual understanding is a first-class capability. The constraint architecture does not compensate for a deficit; it channels a surplus.

11.2 "You cannot rely on LLM judgment."

If LLM judgment is unreliable in all contexts, then LLMs are useless. If it is reliable in some contexts and unreliable in others, the engineering question is how to create the contexts in which it is reliable. That is what the constraint architecture does: schemas ensure structural validity, scoring functions calibrate judgments, consensus mechanisms prevent single-agent errors from propagating, and lifecycle rules ensure that consistently wrong agents decay. The paradigm does not assume perfect agents. It assumes *bounded* agents operating within a structure that compensates for individual limitations—the same assumption that underpins every functioning human organization.

11.3 "Emergent behavior is unpredictable."

Yes. That is the point. If behavior were fully predictable, the system would not be discovering anything—it would be computing something the programmer already knew. The question is not

whether behavior is predictable but whether it is *bounded*—whether the constraint architecture prevents pathological behavior while permitting useful surprise. This is the same question governing any adaptive system, from immune responses to market economies to scientific communities.

11.4 "This will be commoditized into frameworks."

It will. And the commoditization will produce a useful litmus test. When the paradigm attracts "emergence frameworks" that are orchestration with randomness, the name itself provides the criterion: *did behavior emerge, or was it scripted?* If you can read the source and predict the output, it is not Emergence. It is procedural programming with extra steps. A system in which the programmer specified the behavior, however indirectly, is not exhibiting emergence. A system in which the programmer specified the constraints, and behavior arose from agents operating within those constraints, is.

12 Conclusion

Programming paradigms arrive when a new capability makes a new relationship between programmer and machine possible. Structured programming arrived when hardware supported abstraction. Object-oriented programming arrived when GUIs demanded encapsulation. Functional programming arrived (the second time) when multicore demanded immutability.

Emergence arrives because large language models have created computational units with contextual understanding. For the first time, the receiver of a message can decide what it means. For the first time, the computational primitive can exercise judgment. For the first time, the programmer can specify *what matters* and trust the system to figure out *how to assess it.*

This was Kay's vision. Not classes and inheritance. Not design patterns and factories. Autonomous entities interpreting messages, coordinating through shared environments, producing behavior that emerges from their interaction rather than from central specification.

The programmer does not write the program. The programmer writes the conditions under which the program emerges. The constraints are the artifact. The computation is what arises within them.

The name carries an honest admission. Emergence is not deterministic. The same constraints, the same agents, the same signals will not produce identical trajectories. They will produce structurally consistent trajectories—bounded by the same constraints, shaped by the same scoring functions, governed by the same lifecycle rules—but the specific paths will vary. This is the admission that distinguishes the paradigm from its predecessors and the commitment that makes it engineering: design the constraints well enough that every trajectory is acceptable, and the nondeterminism becomes a feature. Design them poorly, and it becomes noise.

That is the skill. That is the discipline. And for the first time, the technology exists to practice it.

References

W. Ross Ashby. *An Introduction to Cybernetics.* Chapman & Hall, 1956.

Carliss Y. Baldwin and Kim B. Clark. *Design Rules: The Power of Modularity.* MIT Press, 2000.

Stafford Beer. *Brain of the Firm.* Allen Lane / The Penguin Press, 1972.

Eric Bonabeau, Marco Dorigo, and Guy Theraulaz. *Swarm Intelligence: From Natural to Artificial Systems.* Oxford University Press, 1999.

Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, Jeff Magee, et al. Software engineering for self-adaptive systems: A research roadmap. In *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 1–26. Springer, 2009.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements*

of Reusable Object-Oriented Software. Addison-Wesley, 1994.

Stuart A. Kauffman. *The Origins of Order: Self-Organization and Selection in Evolution*. Oxford University Press, 1993.

Alan Kay. Re: Prototypes vs classes was: Re: Sun's hotspot, 2003. Email to the Squeak mailing list, archived at `http://lists.squeakfoundation.org/pipermail/squeak-dev/2003-July/066092.html`.

Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36 (1):41–50, 2003.

Jeremy McEntire. Ambient structure discovery via stigmergic mesh. Working paper, 2026.

Jon Postel. DoD standard internet protocol. Technical Report RFC 760, USC/Information Sciences Institute, 1980. The robustness principle: "be conservative in what you do, be liberal in what you accept from others".

Anand S. Rao and Michael P. Georgeff. BDI agents: From theory to practice. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 312–319, 1995.