Biomimicry in AI-Assisted Coding

Embedded architecture for recursive, self-healing software: how the genome, homeostasis and immune response of living systems become embedded specifications, self-validation and autonomous agents.

Abstract

Software decays. Dependencies drift, invariants erode, and the cost of change climbs until a system is too brittle to touch. Living organisms face the same entropy and answer it with architecture: a genome that specifies how to build and rebuild the body, homeostasis that holds internal conditions steady, an immune system that detects and repairs damage, and redundancy that fails safe. We argue that these are not loose metaphors but concrete design patterns, and that AI-assisted coding is what finally makes them affordable to build. When a system's specification can be transcribed into code, and re-transcribed on demand, the specification becomes a genome; when validation and tests live inside the running system rather than beside it, the system gains a sense of its own state; and when autonomous agents operate the system through the same tools its people use, the loop from sensing to repair can close. The specification, not the ageing code, is the thing we maintain. This paper sets out the mapping from biology to software, shows where it genuinely holds and where it misleads, and grounds it in Nashua 360, our own enterprise platform, which is built this way. The result we are after is not autonomy but resilience: software that notices its own drift, heals the routine, and escalates the rest to a human.

Keywords: biomimicry; AI-assisted coding; self-healing software; embedded architecture; specification-driven development; homeostasis; software immune system; autonomous agents; Model Context Protocol; recursive systems; self-validation; resilience; digital enterprise transformation

1. Introduction

Every non-trivial system tends towards disorder. Lehman observed it in the 1970s and gave it a law: a program in use must be continually adapted, and as it is, its structure degrades unless deliberate work is spent to restore it. Four decades later the diagnosis is unchanged and the symptoms are familiar to anyone who has inherited a codebase: dependencies drift out of support, the invariants the schema was meant to guarantee quietly rot, the tests that once described the system describe an earlier one, and the cost of the next change grows until the safest thing to do with a module is to not touch it. We treat this decay as a fact of nature, and then we are surprised when our systems behave like neglected ones.

Biology has a different relationship with decay. A living organism is under constant assault, from entropy, from injury, from a changing environment, and it does not merely resist; it maintains, repairs and renews itself, for decades, out of the same molecules that are breaking down. It does this not through heroics but through architecture that is present from the first cell: a genome that specifies how every part is built and can be rebuilt, a homeostatic apparatus that holds temperature, pressure and chemistry within tight bands, an immune system that detects damage and foreign material and responds, and redundancy everywhere so that failure is local and recoverable. These are the oldest and best-tested resilience patterns we know of, and software has borrowed only the shallowest of them.

The reason we borrowed so little is that the deep patterns were, until recently, too expensive. A genome is only useful if something can read it and build the body; a self-healing system is only useful if it can afford to notice and act on its own state continuously. Writing and rewriting a system from its specification, keeping a living description of correctness inside the running system, and giving the system agents that can operate it, were all more work than simply shipping and patching. That calculus has changed. AI-assisted coding has made transcription from specification to code cheap and repeatable, and the same models, given tools, can read a system's state and act on it. The biological patterns are suddenly affordable. This paper is about building with them on purpose.

We proceed from the mapping (Section 2) to the three mechanisms that matter: the genome of embedded specifications (Section 3), the homeostasis and immunity of self-validation (Section 4), and the recursive loop in which agents operate and repair the system (Section 5). We are honest about where the analogy breaks and what it must not excuse (Section 6), and we ground the argument throughout in Nashua 360, the enterprise platform we build this way.

2. Why biomimicry, and why now

Biomimicry, in Benyus's framing, is not decoration; it is the discipline of learning from designs that have survived a long selection. Nature has been debugging resilience for several billion years, and the survivors share a small set of structural ideas. Table 1 sets those ideas beside their software equivalents and beside the way our own platform embodies each. The claim of this paper is in that middle column: each biological principle corresponds to a concrete engineering pattern, not a vague inspiration.

The timing argument deserves weight, because the patterns themselves are old. Autonomic computing was proposed by IBM in 2003 as self-configuring, self-healing, self-optimising and self-protecting systems; computer immunology was explored by Forrest and colleagues in the 1990s; Armstrong's Erlang made let it crash and supervision trees a working discipline for self-repair; Cannon named

Table 1. The mapping. Each resilience principle that living systems rely on has a precise engineering counterpart, and AI-assisted coding is what makes the right-hand column affordable to build and keep running.

Living system	Software pattern	How Nashua 360 embodies it
Genome and gene expression	An authoritative specification, embedded in the repository and transcribed into code on demand	A single build-conventions document and one specification per module, from which every module is generated and regenerated
Morphogenesis (cells read local context)	New parts match their neighbours rather than an idealised template	The first rule of generation is copy the neighbours: a new module takes the shape of the closest existing one
Homeostasis	Continuous self-measurement that holds invariants within bounds	Live integrity checks that run against the real database, in production, not only in a test lab
Immune system	Detection of damage and anomaly, with a graded response	Tests held as first-class records inside the system and executed on demand; failures are surfaced, not swallowed
Autonomic nervous system	A reflex layer of pre-approved actions, separate from deliberate decisions	An explicit boundary between what agents may do unprompted and what requires a human
Redundancy and fail-safe	Multiple pathways, and a safe default when all fail	Layered fallback for the AI path, and permissions that default to deny when anything is uncertain

homeostasis in 1932. The ideas are excellent and largely unrealised in ordinary software, and the reason is cost. Each of them demands that the system carry extra machinery: a description of itself, a sense of itself, and a means to act on itself. AI-assisted coding lowers the price of all three at once. A specification that a model can turn into code is no longer documentation that rots beside the system; it is an executable genome. A suite of checks that would have been too laborious to write and maintain by hand becomes tractable when a model drafts and repairs it. And an agent that can call the system's own functions can close the loop from noticing a problem to doing something about it. The patterns did not get better; they got affordable.

3. The genome: embedded specifications

The first move is to stop treating the specification as a document and start treating it as a genome. In a living system the genome is not a description of the organism kept in a drawer; it is physically present in every cell, and it is the thing from which the cell is actually built. The software equivalent is a specification that lives in the repository, is authoritative, and is the direct source from which code is generated, rather than a wiki page that describes code written separately and drifts away from it within a sprint.

In Nashua 360 this genome has two parts. One is a single build-conventions document, the developmental rules that hold for every module: how data is owned across databases, how an interface to it is shaped, how permissions gate it, how every change is recorded. The other is a specification per module, the gene for that organ. An AI agent reading these builds the module, and, crucially, can rebuild it, because the specification and the conventions together are sufficient to regenerate the code rather than merely to review it. The description does not document the system; it produces it.

The most biological detail is the first rule of that genome, and it is not an idealised template. It is copy the neighbours: a new module should take the shape of the closest existing one, not of some canonical form held elsewhere. This is exactly how bodies are built. A cell does not consult a master blueprint of the whole organism; it reads its local chemical context and differentiates to match where it is. Morphogenesis is local, and so coherence at the scale of the whole emerges from a rule applied at the scale of the part. For AI-assisted generation this is also the rule that keeps a large system coherent: a model asked to match its neighbours produces something that fits, where a model asked to follow an abstract ideal produces something subtly foreign that the next change has to fight. Genotype and phenotype stay close because the transcription rule is local.

There is a direct dividend for decay. If the genome is authoritative and transcription is cheap, a module that has rotted is not patched in place forever; it can be regenerated from its specification against current conventions, the way tissue is renewed from the same genome that built it. The specification, not the ageing code, is the thing we maintain.

4. Homeostasis and the immune system

A genome builds the body, but it does not keep it alive. Survival is the work of homeostasis and immunity, and both are forms of the same thing: the system continuously sensing its own internal state and acting to keep it within bounds. This is the mechanism software most conspicuously lacks. We test before we ship and then fly blind, trusting that a system correct at release stays correct in use. Organisms do the opposite. They assume that conditions will drift and damage will occur, and they run their sensing and their defences continuously, in the live body, not in a laboratory copy.

The engineering translation is to move validation inside the running system. In Nashua 360 this takes the form of a validator that runs against the real production database and asks, continuously, whether the data is still internally consistent: whether ledgers still balance, whether references still resolve, whether the sequences and reconciliations that the business rules require still hold. These are read-only checks, safe to run in production, and that is the point: they are proprioception, the system's sense of the position of its own parts, not a test performed once on a mannequin. Alongside them sit exercises that behave like a controlled immune challenge: synthetic transactions run against live logic to confirm it still behaves, inside a transaction that is rolled back so the body is unharmed by the test of it. The equivalent in an organism is the constant, low-level activity

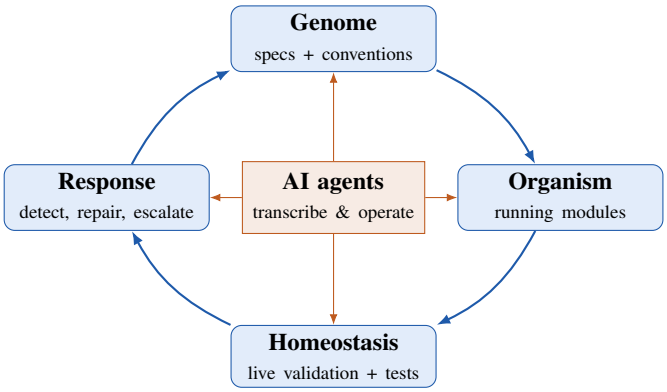


Figure 1. The recursive self-healing loop. A genome of specifications is transcribed into a running organism; a homeostatic layer senses its internal state continuously; and a graded response detects, repairs or escalates. AI-assisted agents sit at the centre because they both transcribe the genome into code and operate the system, which is what lets the loop close on itself.

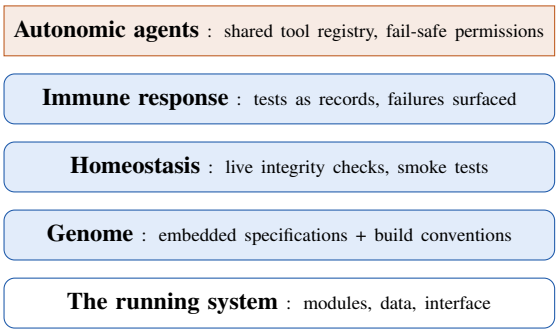


Figure 2. The embedded architecture. The running system sits at the base; above it, and part of the same codebase rather than bolted on beside it, are the genome that builds it, the homeostatic and immune layers that keep it healthy, and the autonomic agents that act through it. Each layer is embedded, which is what makes the whole self-maintaining.

of an immune system that is always probing, not a defence that is mustered only after the breach.

Two properties make this immune rather than merely diagnostic. First, the tests are first-class records inside the system, not scripts in a separate repository: the organism carries its own definition of health, and can be asked at any time to check itself against it. Second, failure is surfaced rather than swallowed. A check that fails does not vanish into a log; it raises a signal that something, a human or an agent, is meant to act on. A system that detects damage and says nothing has an immune system that has itself failed. The goal of this layer is not to prove the system perfect but to ensure that when it drifts, which it will, the drift is noticed early, while it is still cheap to correct.

5. Recursive self-healing

The loop only closes when something can act on what the sensing layer reports, and this is where AI-assisted coding does its most distinctive work. In Nashua 360 the same platform that people operate is also operated by autonomous agents, through exactly the same set of tools. There is a single registry of operations, the system's functional repertoire, and both the people, through the interface, and the agents, through a machine protocol, act through it. An agent can therefore do what the homeostatic layer asks: run the system's own tests, read the results, and, for the routine and well-understood cases, carry out the repair, all by calling the same functions a person would. Figure 1 shows the cycle, and the agents sit at its centre because they are what turns a description and a sense of state into action.

The recursion is the heart of it, and it is worth being precise about. The agents do not stand outside the system operating on it from a privileged position. They act through the system's own repertoire, the same functions, under the same rules, as every other actor. A system whose agents use its own interface to maintain it is self-referential in the way a living thing is: the mechanisms that keep it alive are made of the same stuff as the thing being kept alive. This is von Neumann's old insight about self-reproducing automata, that a system complete enough to describe and build itself can also maintain itself, made ordinary by tools.

That power is exactly why the autonomic boundary matters, and Figure 2 shows where it sits in the stack. A living system does not route every function through the conscious brain; it reserves deliberation for what is novel and consequential, and it hands the rest to reflexes and to the autonomic nervous system, which keeps the heart beating without asking. Software that lets agents act needs the same division, drawn explicitly: a set of actions that are pre-approved and may be taken unprompted, because they are safe, reversible and well understood, and a set that must stop and wait for a human, because they are destructive, ambiguous or touch the system's own safety machinery. In our platform that line is written down, and it is conservative: an agent may generate a module, fix an error it introduced, or run a check; it may not delete data, alter the permission machinery, or change how it is itself governed, without a person. The reflexes are automatic; the irreversible decisions are not.

We are deliberately describing self-healing, not autonomy, and the difference is the escalation path. The honest state of the art, our own included, is that agents heal the routine and must defer the rest. When an agent completes work but part of it fails, the safe behaviour is not to barrel on or to silently roll back a half-finished change; it is to stop, mark the work as partial, and raise it for human review. An organism does the same: minor damage is repaired without awareness, but a serious injury recruits the whole system and, in a social animal, calls for help. A self-healing system that cannot tell the difference between a scratch and a wound is more dangerous than one that heals nothing, and the escalation path is what keeps the recursion safe.

6. Where the analogy breaks

A metaphor is a tool, and a tool used past its purpose does harm, so we are explicit about the limits. Software is not alive, and three differences matter. First, evolution optimises for reproduction, not for correctness or for the user's interest, and a system that heals towards the wrong goal heals faster into the wrong place: the objective still has to be set by people, and the genome still has to be written by people who understand the domain. Second, biological

self-repair tolerates approximation, a healed tissue is good enough, whereas a ledger that is approximately balanced is simply wrong; the homeostatic bounds for software are often exact, and the immune layer must respect that there are invariants with no acceptable drift at all. Third, an organism's autonomy is bounded by a body, while software agents act at the speed and reach of a network, so the blast radius of an unsupervised action is incomparably larger. This is why the governance layer, the permissions, the audit trail, the human gate on the irreversible, is not an add-on to the biological design but a load-bearing part of it.

There is also a failure mode specific to the cheapness that makes all this possible. When regeneration is almost free, the temptation is to regenerate rather than to understand, to let the system churn its own tissue in response to every signal. Living systems guard against this with cost and with inertia: renewal is metabolically expensive and tightly regulated, which is a feature, not a limitation. Software should borrow that restraint on purpose, through the autonomic boundary and through a bias towards the smallest sufficient repair, rather than discovering the need for it after a system has rewritten itself into something no one recognises.

Finally, none of this removes the engineer; it relocates them. The work moves up a level, from writing and patching the code to authoring the genome, setting the homeostatic bounds, defining the immune responses, and drawing the autonomic line. That is more leverage and more responsibility at once, which is the honest shape of every real advance in this field, and the subject of our companion argument on the AI coding chain: the hard part was never generating the code, but verifying it and deciding what it should be.

7. Conclusion

Software decays because we build it like an artefact and then are surprised when it behaves like one. Living systems do not decay in the same way, because they are built to maintain themselves, from a genome that can rebuild the body, through homeostasis and immunity that sense and defend it, to a nervous system that reserves deliberation for what truly needs it. These are concrete, proven architectures, and AI-assisted coding has at last made them affordable to adopt: a specification becomes an executable genome, validation becomes a living sense of the system's own state, and agents acting through the system's own tools close the loop from sensing to repair. The aim is not to remove people or to pretend software is alive. It is resilience: a system that notices its own drift, heals the routine, escalates the wound, and keeps the irreversible decisions firmly in human hands. We build Nashua 360 this way, and we think most enterprise software will, in time, be built this way too. Talk to us about what that would mean for yours.

References

[1] Benyus, J. M. (1997). *Biomimicry: Innovation Inspired by Nature*. William Morrow.

[2] Cannon, W. B. (1932). *The Wisdom of the Body*. W. W. Norton. The naming of homeostasis.

[3] Lehman, M. M. (1980). *Programs, Life Cycles, and Laws of Software Evolution*. Proceedings of the IEEE, 68(9).

[4] Kephart, J. O. and Chess, D. M. (2003). *The Vision of Autonomic Computing*. IEEE Computer, 36(1).

[5] Forrest, S., Hofmeyr, S. A. and Somayaji, A. (1997). *Computer Immunology*. Communications of the ACM, 40(10).

[6] Armstrong, J. (2003). *Making Reliable Distributed Systems in the Presence of Software Errors*. PhD thesis, KTH. The origin of let it crash and supervision trees.

[7] von Neumann, J. (1966). *Theory of Self-Reproducing Automata* (edited by A. W. Burks). University of Illinois Press.

[8] Taleb, N. N. (2012). *Antifragile: Things That Gain from Disorder*. Random House.

[9] Gall, J. (1975). *Systemantics: How Systems Work and Especially How They Fail*. Quadrangle. Gall's law on evolving working systems.

[10] Waddington, C. H. (1957). *The Strategy of the Genes*. Allen and Unwin. Canalisation and the epigenetic landscape.

[11] Anthropic (2026). *Model Context Protocol specification*. https://modelcontextprotocol.io

[12] Anthropic (2026). *Claude Code documentation*. https://code.claude.com/docs

[13] Nashua B.V. (2026). *The AI Coding Chain*. Nashua research papers. https://nashua.nl/en-eu/white-papers/ai-coding-chain